Sapienza Università di Roma
Interactive Graphics

20/04/19

Progress Report
Homework 1

Lorenzi Flavio
mat.1662963
2018-2019

The work done to complete this homework was achieved thanks to the study of the concepts studied in class and on the book, but also thanks to surfing the internet to scrupulous research of in-depth studies on the topics addressed. I will discuss my work point by point:

Point 1

In the html file I declared the two matrices "**projection**" (to guarantee perspective to the cube) and "**view**" (to guarantee the displacement of the viewing angle); these will implement the variable gl_position used for the initial position of the cube.
In the js file then I initialize four variables: two with **gl.getUniformLocation** take as input what we have defined above and, other two (mvMatrix, pMatrix) that instead define the pointers for the calls **gl.uniformMatrix4fv**. The mvMatrix matrix is defined by three vectors of 3 elements each: **eye** describes where the viewer is; **at** is the direction where one looks; **up** is the normal to the eye.
Eye is the most important and is defined by the formula **(radius * Math.sin (phi), radius * Math.sin (theta), radius * Math.cos (phi))**, where the last term guarantees the right initial perspective and depth.
Instead through pMatrix I created a perspective projection characterized by four position variables (the field of view, the aspect defined by canvas, near and far values).
Everything was easily implemented through the sliders " ... " that show us how the perspective changes by changing the value of the main angles implemented in the formula described above, and how the image changes depending on our field of view .
NB: the perspective projection taken in this first point as parameters: **perspective (fovy, aspect, 0.1, 1000.0)** with near and far like constants. After these values have been made parametric, due to the specifications required in Point 4.

Point 2

For the **scaling** (uniform) and **translation** (on xyz) of the cube I acted very similarly: adding (multiplying) the two new matrices to the html document, I created and set them (exactly as in point 1 for position and view). Only that they have been implemented with 3 different unknown values: in the translation matrix the elements [12], [13], [14] define the translation on x,y and z of the cube, so changing these we get the displacement; for scaling instead the elements [0], [5], [10] are required, which are all set in the Scale Slider to guarantee the required uniformity.

Point 3

For the orthographic projection I created a new variable in the html file called **ortoProjection** (NB: to guarantee both types of projections I left the perspective projection by default).
Then I performed the usual operations with gl.getUniformLocation and uniformMatrix4fv and I guaranteed the new projection that takes the following values as input: ortho (left, right, bottom, top, near, far) that are the different coordinates declared at the beginning.
Finally I created sliders to control the shifting of the two near and far planes (this time parametric): obviously when near overcomes Far, or Far is too small the cube disappears.
NB: initially it is set to 5 in the sliders, then it must be increased to get a good view by changing the parameters and angles of the projection.
NB: In the perspective view objects which are far away are smaller than those nearby. In the orthographic view, all objects will appear at the same scale. So perspective viewpoints give us more informations about depth (we use perspective views in real life).

Point 4

First of all in the html file I deleted ortoProjection, as I chose to use only one projection variable for both types of projection.
Also in javascript I have unified the projections distinguishing the two types through **perspective()** and **orto()**.
Through **gl.scissor(0, height / 2, width / 2, height / 2)** & **gl.viewport (0, height / 2, width / 2, height / 2)** I moved everything done so far into a box to the left of the screen, then shifted to make place to another box with another cube with the following param:
gl.scissor (width / 2, height / 2, width / 2, height / 2) & **gl.viewport (width / 2, height / 2, width / 2, height / 2)**;
this allowed me to create two identical cubes, but one dependent on the ortho projection, the second from the perspective.
As required the near-far sliders work for both, while obviously the fov changes only for the perspective type.

Point 5

In the js file I have defined 7 vectors: 4 represent the properties of light (position included) and 3 for the material (expressed in RGBA); in addition there is for material properties the shininess coefficient (set to 100.0).
So I defined for both file (html and js) **ambient**, **diffuse** and **specular** (NB: I also define them later for the products carried out as light*material). Then exactly as in the previous points, with the commands gl.uniform4fv and gl.getUniformLocation allow the passage of information with the html file passing the location as a parameter and creating the output for rendering.
Into the function **quad** I replace each colorsArray with the normals defined above: this allows us to insert with the push function a normal for each vertex, following the formula: N = (b-a)*(c-b).
Below then I insert a new **nBuffer** replacing cBuffer which takes as input normalsArray just defined and obviously instead of vColor I create **vNormal**, passed in **enableVertexAttribArray** which specifies the index that identifies the vertex to be enabled.
In the html file I have defined all material/light properties vectors; in the main() cycle I have executed the following steps in the Vertex Shader: - Fixed the position of the light also defining the vectors NN, H, E; -Transform the normals to the vertices in cordinate eye defining N; - Calculation of the terms that will be used in the equation **fColor = ambient + diffuse + specular** multiplied by the relative coefficients **Kd** and **Ks**.
Taking as material properties a predefined vector (slide), the result was a monochrome purple cube. Then I change it replacing with another predefined vector to obtain a new color similar to yellow.

Point 6

In the html file I put a boolean variable (flag): the goal is to have a button that change the current shading between the two studied types (**Phong and Gouraud**).
The main differences between these are as following: 1) while Gouraud uses the normal average for each vertex, the Phong uses the vertex normalization; 2) while the first uses a modified Phong model to interpolate the shades of light on the vertices, the second uses the interpolation of vertex normals on the edges.
The main important thing is the following: “Phong Shading means the technique, which does the light calculations per fragment while Gouraud Shading does the light calculations per vertex: here the calculated light is (either in a perspective correct manner or linearly) interpolated for all the fragments covered by the primitive. *This increases the performance, but gives a big loss of quality, especially on large primitives and strong specular highlights (the light is not linear distributed on a surface).*
So we need to implement both vertex shader and fragment shader.

In the vertex shader I put an if-else cycle { } in which: if flag == true I will use vertex normals implemented for the computation of fColor = ambient + diffuse + specular; else flag == false I will start passing parameters with new vertexes implemented in the fragment shader: here I will compute fColor = ambient + diffuse + specular, in the same way: while Gouraud is computed in the vertex shader, the Phong is left to the fragment shader.
Into the js I put flag = true and in the render() I put **gl.uniform1i(gl.getUniformLocation(program, "flag"), flag)** to guarantee the passage of info with the html, where I will create a button for switching the shadows: in the js with the function **document.getElementById ("switching").onclick** that allows changing the flag every time the button is pushed, and consequently I will get the desired shading.

Point 7

To complete homework I need to implement a procedural texture on each face of the cube: there are a lot of techniques to do that and a lot of possible results (I take the chessboard making into my example using a mathematical description).
Into the vertex shader I fix **attribute vec2 vTexCoord** == **varying vec2 fTexCoord**
it will output texture coordinates to be rasterized.
Than textures are applied during fragments shading by a sampler (**uniform sampler2D texture**) that returns a texture color from a texture object. Here the main important thing is the following multiplication (texture specifications are combined with the color): **gl_FragColor = fColor * texture2D(texture, fTexCoord)**.
Into the js file taking inspiration from the example on site 2 I create a chessboard (black + material color) creating two images (**Arrays**): one as an initial pointer (support), the other for final creation.
Then I initialized the configureTexture (image) function for creating the texture object: it was later called with the image2 parameter inside. For linking with shaders I created **vTexCoord** thanks to gl.getAttribLocation takes the parameter initialized in the html; instead, through Buffer, I store the **texCoordsArray** data.
NB: In comparison to the generation of an image texture (for example) using a custom image (.jpeg), the procedural texture is preferred because of the low storage cost, unlimited texture resolution and easy texture mapping.

Recent changes

-In the html I added translate and scale matrices inside the vertex normals vec N, so that even in the translation or scaling phase the light remains still and therefore the reflection changes accordingly.
-In js file I changed (pure customization) the procedural texture section in this way:
I put texSize = 32; in the Image1 I changed a value in **var c = (((i & 0x8) == 1) ^ ((j & 0x8) == 0))** (first value was 0 not 1) and in the R(GBA) value in **image1[i][j] = [1, c, c, 1]** set to 1 to change color to red from black; finally I changed the texCoord() vectors to change the chessboard into custom (yellow-red) lines.

Final considerations

The work done to complete this homework is based solely on what was learned on the book; the main advantage was the speed with which I managed to immerse myself in the basic computer graphics procedures, being aware better results could be achieved thanks to more experience in this field or just using different libraries such as Three.js and Babylon.js.