Interactive Graphics - Homework 1

by Edoardo Piroli, 1711234

First Point:

The first point asked to define:
1. The viewer position: I've defined it using the lookAt(eye, at, up) function. This function abstracts some of the complexity of moving the camera, in fact it returns a matrix which performs rotations and translations to move the camera from the default position(the origin, pointing in the negative z direction) to the eye position looking toward at and tilted accordingly to up.
I've made the at and up values constants. In fact I've found reasonable, in this case, to always look at the origin (0, 0, 0) hence where the vertices are located, and to do so without being tilted, therefore with up = (0, 1, 0). eye is defined as follows: (radius*sin(theta)*cos(phi), radius*sin(theta)*sin(phi), radius*cos(theta)
The default value of eye is (0, 0, 6) determined by radius = 6.0, theta = 0.0 and phi = 0.0
In order to allow to edit the eye position within the application I've implemented 6 buttons: increase and decrease theta, increase and decrease phi and increase and decrease radius.
Increase and Decrease Radius will multiply or divide its current value by 2.
Increase and Decrease Phi or Theta will add or subtract from the respective variable a constant amount.
2. A projection: I've decided to apply an orthogonal projection, which is a special case of parallel projections, whose peculiarities are: the centre of projection placed infinitely far away from the objects, projectors that never converge and perpendicular, orthogonal, to the projection plane.
I've defined the orthogonal projection via the predefined ortho(left, right, bottom, top, near, far) function. This function returns a matrix which is used to project the vertices to the the viewing volume, in fact the viewer will only be able to see objects within the parallelepiped defined by the vertices (left, bottom, -near) and (right, top, -far). The matrix returned by ortho is a 4x4 matrix because it uses homogeneous coordinates and it stores the depth information in order to be able to apply hidden surface removal later in the pipeline.
near and far are defined as the distances from the camera to the respective planes. It is possible to change both of them using increase, or decrease, Z button which multiply by 1.1, or 0.9, both of them.
This method prevents near from becoming bigger than far. The default values are: far = 8.0, near = 1.0
It is also possible, as requested, to change the values of left, right, top and bottom using the buttons Wider, Narrower, Higher and Shorter which apply a multiplication by either 1.1 or 0.9 to the respective parameters(left and right for Narrower and Wider, top and bottom for Higher and Lower).

Second Point:

The second point asked to include:
1. A uniform scaling matrix: I've defined the scaling matrix using the scalem(x, y, z) function.
I apply, by default, a 0.5 uniform(in all 3 directions) scaling factor, but as requested it is possible to change this factor in the application, I've made it possible, using a slider, to go from -2 to 2 in order to show that negative values also apply a reflection about the origin to the vertices.
It is not possible, and not requested, to apply a non-uniform scaling directly through the application.
2. A translation matrix: I've defined the translation matrix using the translate(x, y, z) function.
By default the vertices aren't translated, or better yet, they are translated by 0 along each of the axes.
It is possible to apply translations along any of the them using the 3 available sliders which go from -1 to 1.

Third Point:

The third point asked to define an orthographic projection which I had already defined in point 1.
Although this point also asked to make it possible to change the near and far planes through one slider each.
I've made it possible, in fact both of their values can be changed from -10.0 to 10.0 with a 0.01 precision.
In order to avoid that the far value could become smaller than the near one I've realised 2 event listeners which prevent that. For example if the far slider is moved to a value lower than the current near value, then it sets near to the value far - 0.01. This increases the robustness of the application.
Since I had already defined the Increase and Decrease Z buttons, which I left in the application, I decided to link the sliders and the buttons, in fact if the buttons are clicked also the sliders change their current values, of course with the buttons it is possible to go below -10 and/or above 10 this is not supported by the sliders which will stop at the nearest supported value.

Fourth Point:

The fourth point asked to split the window vertically, representing the objects in both of them: using the previously defined orthographic projection for the first and a perspective projection for the second.
Perspective projections, as opposed to orthogonal ones, have the centre of projection at a finite distance and objects which are further away are perceived smaller; this effect is obtained dividing the x and y coordinates by the z coordinate, which represents the depth.
I've defined the projection matrix for the perspective projection using the perspective(fovy, aspectRatio, near, far) function which defines the viewing volume as a frustum(a truncated pyramid) and clips out the vertices which are not comprised in it.
fovy is the angle, which goes from the top to the bottom plane along the y axis, which defines the field of view; the aspectRatio is needed to preserve the shape of the represented objects which otherwise wouldn't be preserved on a rectangular projection plane, it is defined as width/height of the projection plane.
Perspective projections in WebGL are actually performed as if they were orthogonal, on distorted, in order to preserve the different effect, objects; this allows for a simpler, hence more efficient, pipeline.
The operation which transforms a perspective projection is called perspective normalisation transformation.
In order to split the window into 2 different parts I've used 2 viewports and the gl.scissor(x, y, width, height) function.
In the left part of the canvas I've projected the vertices via the orthogonal projection whereas in the right part I've projected them using the perspective projection.
All of the sliders works for both of them with the obvious exception of Higher, Shorter, Wider and Narrower.

Fifth Point:

The fifth point asked to introduce a light source, replace the colours with the properties of a material and assign to each vertex a normal.
I've added a light source in the position (1.0, 1.0, 2.0), with the fourth component alpha = 1.0, meaning it is a point source, hence an ideal source of light which spreads equally in every direction.
I've used standard values for the other 3 components of the light:
-lightAmbient = (0.2, 0.2, 0.2, 1.0)
-lightDiffuse = (1.0, 1.0, 1.0, 1.0)
-lightSpecular = (1.0, 1.0, 1.0, 1.0)
As of the material I've replaced the colours with the properties of the emerald found here:
http://www.barradeau.com/nicoptere/dump/materials.html
Also I've assigned a normal to each vertex in the quad function.
It's not possible to perfectly evaluate analytically the light effects since they are continuously absorbed and reflected by all the objects in the scene, this is the reason, in computer graphics, we use models which approximate the effects of light.
In particular the model of reflection I've used is the Phong Blinn reflection one which describes the way a surface reflects light as a combination of ambient light, a small amount scattered throughout the whole scene, with diffuse reflection, of rough surfaces, and specular reflection, of shiny ones. This differs from the Phong reflection model because there is one more approximation done using the half way vector h, which is the normalized vector halfway between v and h, instead of calculating a new reflection and view vector for each vertex.

Sixth Point:

The sixth point asked to implement the Gouraud and the Phong shading models, with a button switching between them.
The button is the Gouraud/Phong one and is implemented as a uniform boolean variable in the shaders which determine which of the 2 models it should use to determine the colours.
By default the implemented shading model is the Gouraud one.
The Gouraud shading model works at the vertex level, and I've implemented it in the vertex shader.
It evaluates the colour, using the material properties and the vertex n, v and l, for each vertex, and linearly interpolating(task performed by the Rasterizer) the shades for every fragment; of course this method is an approximation, hence non-photorealistic: in fact if the light source is sufficiently far away and the viewer is distant, or there aren't specular reflections, this method will shade every vertex with a constant colour.
Also the Gouraud shading model is very efficient, in fact is way less expensive to evaluate a linear interpolation between already evaluated colours than to evaluate the colour for each pixel; although this

efficiency comes at cost in terms of fidelity, in fact if we were to apply this shading model to a mesh with a localised light effect in the centre this wouldn't be rendered correctly.
This is noticeable even in the application I've implemented looking at the differences with the phong model on the centre of the most illuminated face.
Another drawback of using the Gouraud shading model is that it might render Mach bands, exaggerating the contrast between edges of slightly different shades of grey.
The Phong shading model, instead, works at the fragment level, and I've implemented it in the fragment shader. In particular it linearly interpolates normal vectors across the surface of the polygon from its vertex normals. The surface normal is interpolated and normalised at each pixel.
In fact, it is less efficient but more accurate than the Gouraud model since the reflection model must be computed at each pixel instead of each vertex. Although the Phong model looks smoother when rendering polygon mesh with high curvatures than the Gouraud one.

Seventh Point:

The seventh point asked to add a procedural texture on each face of the cube, evaluating the colour as a combination of the light and the texture.
Textures are just patterns and can be single or multi-dimensional; those I've applied are 2-dimensional.
The textures are usually generated from an image, in my case I've just generated them "manually" with 2 for loops. This kind of textures is called Texture mapping.
Textures are stored like arrays, whose elements are called texels; the texture map associates a texel with each point on a geometric object, that is mapped to screen coordinates to be displayed.
I've applied 2 textures, which is just the same of applying only 1, although in some applications there might be the need for successive applications of textures in order to obtain more involved graphical effects, e.g. rendering the shadow of an object whose shades are already determined by a texture mapping.
My final texture looks like a checkerboard black and green, the second colour is determined by the properties of the object previously defined.