HOMEWORK 1 DOCUMENTATION

The goal of the first homework is to allow the user to manipulate certain aspects of a scene that shows a cube, including the position of the camera, the type of projection, the shading and so on.
The cube is visualized on a 512x512 canvas at the center of the page, while the options available to the user are shown on either side. Below there is a short explanation for all the tasks requested by the assignment.

1. Changing the Viewer Position

The Viewer Position indicates where our camera is located in the world space: in order to change this position we can compute the *modelView* matrix with the *lookAt* function, which makes the process of setting up the camera a very intuitive one. In fact this function accepts three parameters: the position of the camera itself, where it should look at (in our case the origin of the world space) and how it should be oriented. By adding some sliders in the HTML file, we can manually control the values of the variables that alter that very first parameter of the function, in a way that ensures a smooth movement of the camera across the three axes, while leaving the other two parameters unchanged.
The parameters that we can change are the *radius* and the angles *theta* and *phi*, while the formula used to determine the new position is:
x = *radius**sin(*theta*)*cos(*phi*); y = *radius**sin(*theta*)*sin(*phi*); z = *radius**cos(*theta*).
Once the new *modelView* matrix is computed, it is then sent to the vertex-shader.

2. Changing the Viewing Volume and the type of Projection

The Viewing Volume is what the camera is able to see, while every object or part of it that falls outside is clipped out: this volume is composed by everything there is between two planes, one that is near the camera and one that is far away. In order to change the volume, we can manually tweak the distance of these two planes in relation to the camera thanks to a pair of sliders defined in the HTML file. This solution works regardless of the type of projection we want to use, since the concept of viewing planes is shared between the perspective and orthogonal projections.
When the program is started up, the cube is visualized by default with the perspective projection, where all projectors converge at the center of the projection and whose matrix is computed with the *perspective* function, which, besides the position of the planes, needs a FOV (field of view) parameter and an aspect ratio. Since our canvas has the same length for its height and width, a good aspect ratio that doesn’t deform the image is 1 (which is

equal to height/width).
However, with a button defined in the HTML file, we can switch to the orthogonal projection, where all projectors are parallel between each other and whose matrix is computed with the *orthogonal* function, which, besides the position of the planes, needs four other parameters in order to determine their size. In our case, the planes are 2x2.
It must be noted that the Viewing Volume is computed differently between the two projections, even if we use the same values for the planes. This means that an object can be fully visible with one type of projection, while it's fully clipped out when we switch to the other one.
Once the new projection matrix is computed, it is sent to the vertex-shader.

3. Scaling and Translating

We can change the position of the cube across the world space, so that we can alter its size or translate it, by multiplying the positions of the vertexes after they are sent to the vertex-shader with two matrixes, one that dictates the scaling while the other the translation.
The order of the multiplication is critical and it's:
New Position: Scaling*Translation*Old Position.
In our case, it's possible to use only one matrix that combines the effects of a scaling and translation, which I called *transformations*, and that is equal to:
{scale, 0.0, 0.0, 0.0,
0.0, scale, 0.0, 0.0,
0.0, 0.0, scale, 0.0,
translation, translation, translation, 1.0}*.
The values that dictate how much we need to scale or translate are obtained with a pair of sliders defined in the HTML file, and then sent to the vertex-shader. The values are also applied uniformly on the three axes.
*: OpenGL uses a column-major order.

4. Splitting the Canvas

A good way to showcase the differences between the two types of projection is to split the canvas and render the same object twice, so that we can simultaneously see the cube from the same angle but with a different type of projection enabled on each side.
Since I wanted to keep the proportions of the cube intact, I decided to split the canvas in four parts: the perspective projection is shown on the upper left corner, while the orthogonal view is shown on the bottom right one. The downside is that the other two corners are left empty.
In order to draw the same cube twice on the same canvas we first change which part of the canvas we are working on thanks to the *viewPort* function, then we compute the right

projection matrix and finally we draw the cube, and then we repeat the same process with the other type of projection. It's important to note that the canvas isn't cleared when we switch to the second drawing process, so that the first cube isn't erased.
The option to split the canvas is enabled by a button defined in the HTML file.

5. Shading

We can enable shading by introducing a light source in the scene and by describing the properties of the cube's material: both have an ambient, diffuse and specular components, while the materials also have a shininess factor. Once all these values are initialized, we calculate the product between each pair of components and send the results to the vertex-shader. We can then compute the shading in the vertex-shader itself (Gouraud Model), or pass those values to the fragment-shader through varying variables (Phong Model) so that we can obtain a better result at the cost of a higher computational cost.
Regardless of the approach, for each vertex we must compute four vectors: the ones that link the vertex to the viewer (*pos*), to the light source (*light*), to the perfect reflector and the normal of the plane in which the vertex resides (*N*). Each of these vectors is multiplied with the *modelView* matrix, so that we can obtain their eye coordinates, which assume that the observer is at the origin of the system, and then normalized.
Because our light isn't directional, meaning that it has a known location that can be measured, we can find the *light* vector by computing the difference between the position of the light and the one of the vertex, while we can find the *pos* vector by simply using the inverse of the vertex coordinates, since the viewer is a the origin.
For the normals we can take three vertexes for each side of the cube, compute (*p2-p1*)x(*p1-p0*), and then assign the result to each vertex of that side and send it to the vertex-shader.
We can skip the calculation of the reflection vector by using instead the halfway vector (*H*), which is computed by the normalized sum of the *light* and *pos* vectors.
The first three vectors can be computed in the vertex-shader regardless of the chosen solution, while the halfway vector must be calculated in the fragment-shader if the Phong Model is enabled.
Once we obtain all the pieces, the final formula for the color of the vertex (Gouraud) or of the fragment (Phong) is:
ambientProduct + diffuseProduct*($light\bullet N$) + specularProduct*$(N\bullet H)^{\text{shininess}}$.
The model can be chosen thanks to a series of buttons defined in the HTML file, and then a value is sent to the vertex-shader so that it can find out what model has been selected.

6. Texture Mapping

We can enable texture mapping by creating an image in the javascript file, binding it to a texture, linking each vertex of the cube with a point of the texture, and then send those

values and the texture itself to the vertex-shader, so they can reach the fragment-shader through varying variables. Once they do so, the color of the texel is multiplied with the one of the associated fragment: the result is the image applied on top of every side of the cube.
When binding the image to the texture we also need to define its mipmap, that is how the image gets progressively scaled back when viewed in lower resolutions in order to avoid artifacts or aliasing.
Instead of using a procedural image, in our case a checkerboard, we can apply an already existing one: however the browser may reject it for security reasons. In my case, the image I selected does work, even though I don't know why there are patches of pitch black on its perimeter.
The user can enable texture mapping through a button defined in the HTML file, and then a value is sent to the vertex-shader so that it can find out what kind of texture has been selected.