# Interactive Graphics HomeWork#1 Documentation

by Herson John Nolasco Ruiz

April 24, 2019

## 1 Viewer and Projection

The **viewer position** is added in polar coordinates through the variable **eye**. The variable eye can be manipulated through sliders in order to modify the $\boldsymbol{\theta}$ angle, $\boldsymbol{\phi}$ angle and the **radius** that is the distance from the origin.
The angles $\theta$ and $\phi$ are set by default to 0 and can be modified from -90 to +90.
The radius is set to 3 by default and can be modified from 0.05 to 10.
The model-view matrix is computed through the **"lookAt"** function that takes in input the variable eye previously defined and two more variable "**up**" and "**at**" fixed in the initialization.
The projection selected for the first point is the **"perspective projection"** because I want to provide the sense of "depth" that cannot be achieved by the "orthogonal projection".
To implement the perspective projection and compute the projection matrix I used the **"perspective"** function defined in the MV.js file. This function takes in input 4 parameters:

- **fovy** is the field-of-view (y-axis) by default set to 45 degrees.
  This variable can be modified in the range [10, 120 degrees]. When this parameter gets smaller the objects in the scene get bigger.
- **aspect** ratio parameter is the width divided by the height of the canvas window. Set by default to 0.5. The user can adjust this parameter in the range [0.1, 2].
- **near** and **far** distances that affect what is clipped from the scene. By default near is set to 0.3 and far to 12. The variable near can be modified in the range [0.1, 3] and far in the range [3, 15]. The first variable should always be less than the second.

There is also a "Reset" button that sets to default all the parameters for viewing position.

## 2 Scaling and Translation

To implement the **uniform scaling** and compute the **scaling matrix**, I used the **"scalem"** function defined in the MV.js file.
The input of the scalem function are three parameters that indicate the scaling factor for the three axis (x, y, z). To get a uniform scaling the three variables are set equally to a single parameter called "scalingFactor".
The default value of the scaling factor is 0.5. In order to manipulate the scaling factor there is a Scaling Slider that goes from 0.2 to 1.3.
To implement the **translation** and compute the **translation matrix** we use the "**translate**" function defined in the MV.js file.
The input are three parameters that indicate the translation through the three axis (x, y, z).
These three parameters can be manipulated through three sliders named "translateX", "translateY", translateZ". The range of values accepted are from -1.5 to 1.5 and the default value is 0 for "translateX" and "translateZ" and 0.5 for "translateY".

# 3 Orthographic projection

The **orthographic projection** is implemented through the **"ortho"** function defined in MV.js file. The view volume is a parallelepiped and the sides of the clipping volume are the four planes x = left, x = right, y = ytop, y = bottom.
The near (front) clipping plane is located a distance near from the origin, and the far (back) clipping plane is at a distance far from the origin.
These values are the input of the "ortho" function mentioned before. The values of the first four parameters are fixed ( left = -0.5, right = 0.5, ytop = 1.0, bottom = -1.0).
The values of "near" and "far" can be manipulated through two sliders:

- **"near"** is set by default to 0.3 and can be modified in the range [0.1, 2.8].
- **"far"** is set by default to 10 and can be modified in the range [3, 15].

# 4 Splitting the window

In order to split the windows vertically the idea is to use two viewports one for the perspective projection and another one for the orthogonal projection.
The size of the two viewports are **(canvas.width/2, canvas.height)** in order to split equally the canvas.
In order to create distinct viewports and to draw different renders we need to pass different matrices to the render function.
So in the index.html file, in the main function of the vertex-shader we define the gl_position as the product of all the matrices:
**gl_Position = translationMatrix*scalingMatrix*perspectiveMatrix*orthogonalMatrix* modelViewMatrix*vPosition**.
To render the two different projection:

- **perspective**(left-viewport): we compute the perspective matrix and we set the orthogonal matrix to the identity matrix using the function mat4().
- **orthogonal**(right-viewport): we compute the orthogonal matrix and we set the perspective matrix to the identity matrix using the function mat4().

We can modify the "near" and "far" parameters for the perspective and orthogonal projections using the sliders defined before.

# 5 Lighting

I choosed to use a single point light source with these specifications:
**var lightPosition = vec4(1.0, 2.0, 3.0, 1.0);**
**var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0);**
**var lightDiffuse = vec4(1.0, 1.0, 1.0, 1.0);**
**var lightSpecular = vec4(1.0, 1.0, 1.0, 1.0);**

The material that I wanted to represent with the object is the "Polished Gold" and the specifications for this material are the following:
**var materialAmbient = vec4(0.24725, 0.2245, 0.0645, 1.0);**
**var materialDiffuse = vec4(0.34615, 0.3143, 0.0903, 1.0);**
**var materialSpecular = vec4(0.797357, 0.723991, 0.208006, 1.0);**
**var materialShininess = 83.2;**

The normals of the vertices are stored in the **normalsArray**[] in the .js file.

# 6 Gouraud and Phong Shading models

In the previous section we implemented the Gouraud model because basically we computed the lighting equation in the vertex shader. In order to implement the Phong model I added the computation for the lighting equation in the fragment shader.
In the vertex shader and in the fragment shader we have an **if-else** controlled by a variable called "**changeShading**" that is set by default to "true".
If the variable is set to "**true**", the **Gouraud model** is loaded and basically we have the computation of the color for the vertices made by the vertex shader that pass it to the fragment shader.
Instead, if the variable is set to "**false**" the **"Phong" model** is loaded and the vertex shader passes the normal and position data to the fragment shader that computes the color by interpolation.

# 7 Procedural texture

The procedural texture that I wanted to implement for the cube is the "**Checkerboard texture mapping**" such that the texture image is used for each side of the cube.
The final pixel color is the combination of the variable "**fColor**" that is the is the sum of the ambient, diffuse, and specular contributions of the lighting model and the "**2D sampler**".
The checkerboard texture is implemented through the variables "image1" and "image2", the first creates the checkerboard pattern using the floats and the second that converts the floats to unsigned bytes.
In the "configureTexture" function, we set some parameters to deal with the "aliasing" problem:

- For the "**magnification**" problem, we used the value of the nearest point sampling.
- For the "**minifaction**" problem, we used a technique called "**mipmapping**" that use point filtering with the best mipmap.