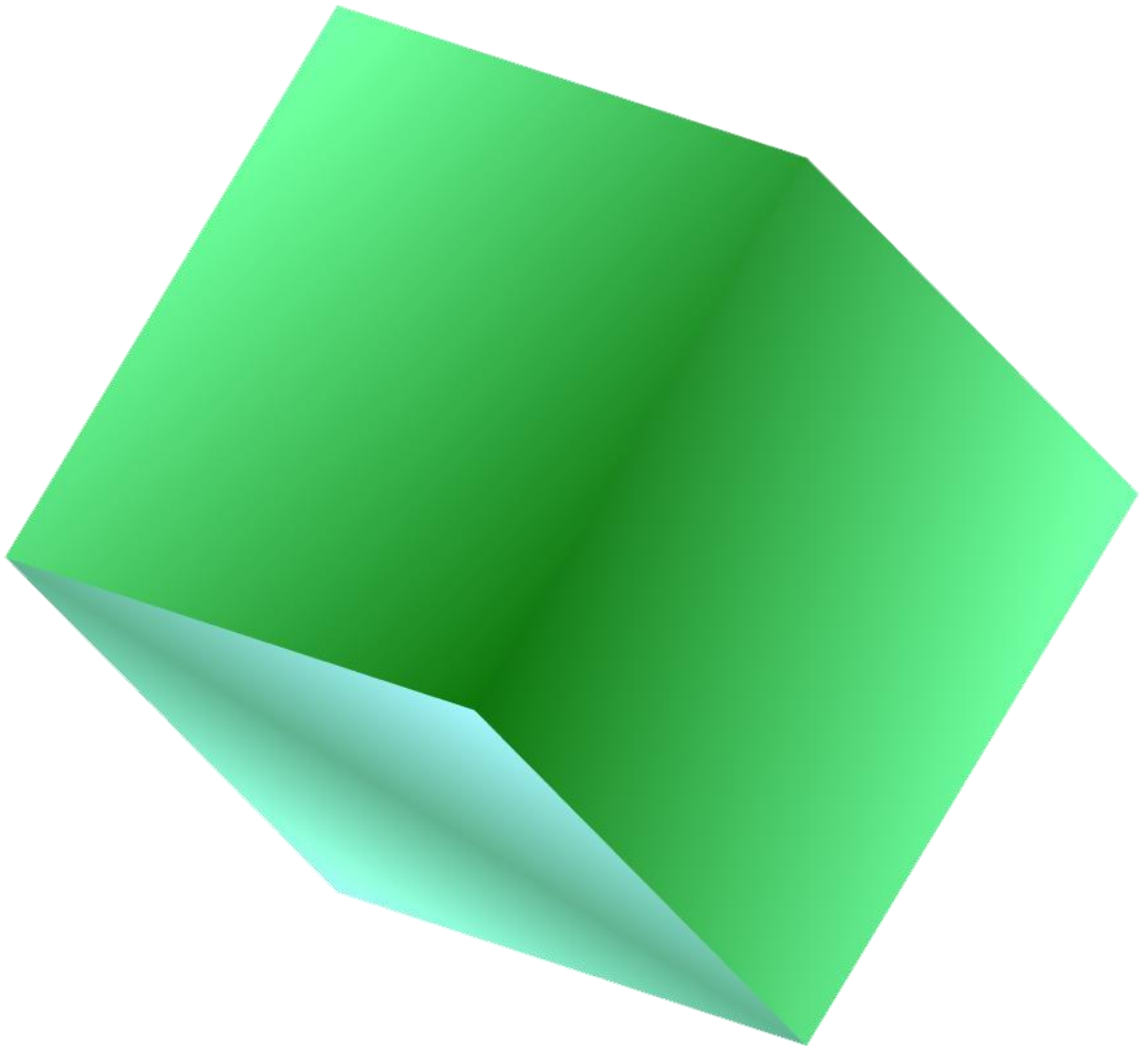
Interactive Graphics Report

Building and Animating a 3D Cube in WebGL



"LA SAPIENZA", UNIVERSITY OF ROME

5 maggio 2018

Autore: Simone Faricelli 1647406

INTERACTIVE GRAPHICS REPORT

BUILDING AND ANIMATING A 3D CUBE IN WEBGL

INTRODUCTION - WEBGL

The idea of bringing the real-time 3D graphics inside a Web page is not a new feature. In fact, since the birth of WWW, we have seen a string of different implementations, in order to pursue that purpose: applet Java, VRML, Flash and many others. With the arrival of HTML5, we are likely to reach a standardization for the real-time 3D rendering inside a browser, due to the use of WebGL and the tag <canvas> as tools to build and animate three-dimensional models.

WebGL is in fact a Javascript Application Programming Interface (API) based on OpenGL, compatible with any web browser, for rendering 2D and 3D graphics through the use of GPU-acceleration. It is basically composed by a control code side, written in JS, and a shader code side, written in OpenGL ES Shading Language (GLSL).

SHADERS

The shader is a set of instruction that tells the GPU essentially what to do, during the rasterization phase, with the data we passed to it. In order to make it possible, we must provide two shaders, respectively called Vertex Shader and Fragment Shader.

VERTEX SHADER

The Vertex Shader's main task is to map the object's world coordinates into clipping coordinates. In other words, when it is invoked (once for each vertex), it will transform a set of coordinates that model the vertex in the space into another set of coordinates that, instead, model the vertex in the bidimensional space (or window) of the <canvas>. The output of this process is stored inside the built-in variable gl_Position, which is eventually passed to the rasterizer. All the calculations are computed with respect to the projection type, position of the virtual camera, lights and textures or material properties.

FRAGMENT SHADER

The Fragment Shader, instead, is responsible for assigning a four-dimensional RGBA colour to each fragment, coming from the rasterizer, which is the actual pixel rendered in the canvas window. As for the vertex shader, each fragment is stored inside the built-in variable gl_FragColor.

SCRIPT

A WebGL script can be roughly divided into two phases (and functions): Init and Render.

INIT

In order to pass the vertex arrays to the GPU and render them, we have to put these data inside a buffer. The declaration of these objects is made through the function gl.createBuffer() , and then then set-up of this buffer as the current buffer is made through the other function gl.bindBuffer() so that, anytime a function will put some data into a buffer, the bind one will be used until another one is bound. The last step of the initialization is clearly, the placement of the data inside the buffer. This is done through the function gl.bufferData().

As for the data buffer, the shaders also need to be initialized. To do that we create a container called program object through the function initShaders("vertex", "fragment"), which will hold the two shaders together and it will be used every time we need to pass variables or arrays to the shaders.

Last step is the initialization of the canvas windows inside the html.

RENDER

Finally we can build and show our 3D model through the function gl.drawArrays() and continuously refreshing it, we can handle object, light and virtual camera's movement. In order to animate the object without passing and rendering different vertices periodically, the render function will call itself recursively.

TECNIQUES AND FEATURES

Let's analyse and discuss the solutions adopted in this homework.

APPLICATION OR SHADER

Speaking of Rotation, Translation and Scaling, or more in general their concatenation, of ModelView Matrix, it's know that there are different ways to switch from world coordinates to clipping coordinates.

The first approach consists in sending the angle values to the vertex shader, letting it compute the calculation to get the new vertex position. This is indeed the most efficient method because every time we have to calculate the new position of the object, we are not sending new data from the CPU to GPU, avoiding then the bottleneck effect due to the transfer of big amount of data. This problematic could show up instead in the second approach.

In the second approach, the one we adopted, we form in the application a new ModelView Matrix for each Transformation, then we send it to the vertex shader and apply it to the vertex data there. As written above, this is surely not the best method in terms of efficiency but it probably is the easiest way for a programmer to code a WebGL Animation thanks to the use of JS libraries.

TRANSFORMATION MATRIX

In order to include a scaling and translation matrix controlled by slider, I started adding in the html code the tag <input type="range">, which is the slider that controls the value of each variable. In the JS file, the value is then stored in a variable through the use of the JS Event Listener (document.getElementById().**oninput**) which is invoked every time we Drag and Move the slider. No need to release the cursor to apply the changes. The parseFloat function is then added to guarantee the data type of the value.

At this point, instead of declaring and filling two new matrices to compute the transformation of the coordinates, I used the premade functions, inside the MV library, mult(modelViewMatrix, scalem(s, s, s)) and mult(modelViewMatrix, translate(x, y, z)) to multiply the ModelView Matrix by the uniform scaling matrix of factor "s" and by the translation matrix of factors x along x axis, y along y axis and z along z axis.

PROJECTION

Now it's time to position and orient the camera. To do that we need, at first, another change of coordinates to represent the object in the camera frame, so we have to multiply the ModelView Matrix by the function lookAt(eye, at, up) from the MV library. Eye, At and Up are three-dimensional vectors which represent respectively the position of the camera in the space, the point the camera is focusing and the orientation of the upper side of the camera.

The second step is the application of the projection transformation, orthographic or perspective, to the vertices, obtaining the clipping coordinates for the objects within the clipping volume, which is the volume, delimited by the near and far planes, that the camera "can see". We're now introducing a new transformation matrix, that will be later multiplied by the ModelView Matrix, named Projection Matrix.

ORTOGRAPHIC

Is a projection in which the center of projection is indefinitely far from the objects being viewed, resulting in projectors that are parallel rather than converging at the center of projection. The Projection Matrix, and therefore the clipping volume, is then defined through the function ortho(left, right, bottom, top, near, far) where the first four variables are the lateral delimitation planes of the volume, arbitrarily chosen, and the last two, are the variables (planes) mentioned above.

PERSPECTIVE

Is the projection we get with a camera whose lens has a finite focal length or, in other words, the center of projection is finite. Here the Projection Matrix is defined through the function perspective(fovy, aspect, near, far) where the fovy is the angle between the top and bottom planes of the clipping volume, the aspect is the ration between width and height of the display area. As before, near and fare are the planes mentioned before.

LIGHT AND SHADING

There are many ways to implement lighting, but we're using the directional lighting, because of is the simplest. Directional lighting assumes the light is coming uniformly from one direction. The sun on a clear day is often considered a directional light. It's so far way that its rays can be considered to be hitting the surface of an object all in parallel.

In order to represent the light, is possible to choose between two different kind of shading: Gouraud and Phong. In Gouraud shading we define the normal at a vertex to be the normalized average of the normales of the polygons that share the vertex. Phong can be instead seen as an improvement of the Gouraud shading. In fact instead of interpolating vertex intensities, here we interpolate normals across each polygon. For this reason, Phong shading requires that the lightning model be applied to each fragment. We're therefore implementing Phong shading though the fragment shader.

CONCLUSION

For a matter of design, I eventually added some CSS to the HTML file.