Interactive Graphics - 2019 - HW1

> Student: Ion Tataru; Student ID: 1595331;

Task 1

Add the viewer position (your choice), a projection (your choice) and compute the ModelView and Projection matrices in the JavaScript application. The viewer position and viewing volume should be controllable with buttons, sliders or menus.

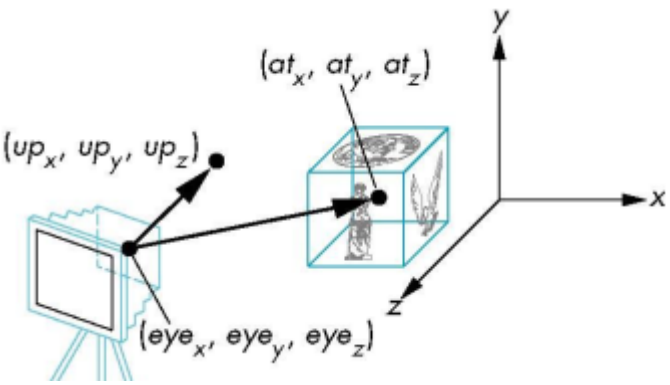
Solution T1

The viewer position related to the `eye` position expressed in polar coordinates, that is controlled by the *user* with the:

- `radius` - the distance from the origin;
- `theta` and `phi` angles - characterize the degree of obliqueness;

The **ModelViewMatrix** is computed by: `modelViewMatrix = lookAt(eye, at, up);`

- `at` - the point where the camera is looking at;
- `up` - the point that specifies the way your camera is rotated;
- `LookAt(eye, at, up)`



The viewer volume is controlled by the following parameters:

- Orthographic Projection case: `right`, `left`, `top`, `bottom`, `near`, `far`.
  -

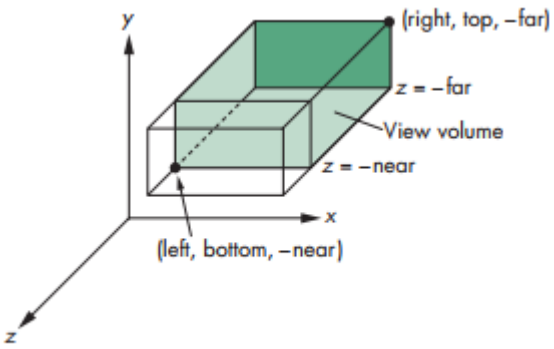


FIGURE 5.22 Orthographic viewing.

- The **Orthographic Projection Matrix** is computed as following: `projectionMatrix = ortho(left, right, bottom, ytop, zNear, zFar);`
  - `ytop` - to avoid the conflict with the window object member name (`top`) ;

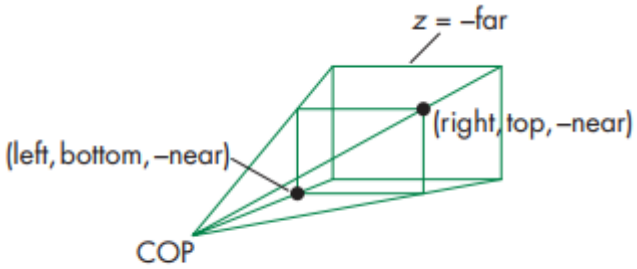
- Orthographic Projection case:
  - `fovy` - field of view along Y axes; The angle between the top and bottom planes of the clipping volume;
  - `aspect` - ratio (*width divided by height*) of the projection plane;
  - `near` and `far`;
  - 

    FIGURE 5.40 WebGL perspective.
  - The **Perspective Projection Matrix** is computed as following: `projectionMatrix = perspective(fovy, aspect, zNear, zFar);`

We send the model-view and the projection matrixes to the vertex shader (VS) with:
`gl.uniformMatrix4fv(modelViewMatrixLoc, false, flatten(modelViewMatrix));`
`gl.uniformMatrix4fv(projectionMatrixLoc, false, flatten(projectionMatrix));`

> Note: The splitting procedure for the two different projections are explained bellow, in Task 4;

> Note: By default, orthographic projection is used;

Task 2

Include a scaling (uniform, all parameters have the same value) and a translation Matrix and control them with sliders.

Solution T2

The translation is controlled by the 3 range sliders (*Translate-X*, *Translate-Y*, *Translate-Z*), one for each axes.

The scaling is controlled by the *Scale Factor* range slider.

The Translating and Scaling matrixes are declared in the vertex-shaders, after the scale and translation vectors parameters are passed to the shaders (the components of the scale vector are all the same (*uniform scaling*)):

```
            uniform vec3 t;    // translation vector
            uniform vec3 s;    // scale factor vector

            // This matrices are column-major !!!
            mat4 T = mat4(  1.0,  0.0,  0.0, 0.0,
                            0.0,  1.0,  0.0, 0.0,
                            0.0,  0.0,  1.0, 0.0,
                            t.x,  t.y,  t.z, 1.0 );     // Translation Matrix


            mat4 S = mat4(  s.x,  0.0,  0.0, 0.0,
                            0.0,  s.y,  0.0, 0.0,
                            0.0,  0.0,  s.z, 0.0,
                            0.0,  0.0,  0.0, 1.0 );     // Scaling Matrix

            /** . . . **/

            void main(){
                /** . . . **/
                gl_Position = T * S * projectionMatrix * modelViewMatrix * vPosition;
                /** . . . **/
            }
```

We can observe that Scale (`S`) and Translation (`T`) are the last transformation applied to the model.

Task 3

Define an orthographic projection with the planes near and far controlled by sliders.

Solution T3

See Task1.

Task 4

Split the window vertically into two parts. One shows the orthographic projection defined above, the second uses a perspective projection. The slider for near and far should work for both projections. Points 5 to 7 use the splitted window with two different projections.

Solution T4

We use `gl.viewport` and `gl.scissor` to split the canvas in two parts, one for each of the two projections views. With `glScissor` we define a rectangle, called *the scissor box*, in window coordinates. The first two arguments, *x* and *y*, specify the lower left corner of the box. *width* and *height* specify the width and height of the box.

```
var render = function() {
    function renderScene(X, Y, width, height, projectionMatrix) {
        gl.enable(gl.SCISSOR_TEST);       // Only pixels that lie within the scissor
        gl.viewport(X, Y, width, height); // box can be modified by drawing; commands;
        gl.scissor(X, Y, width, height);  // Define the scissor box;
        /**                    . . .                          **/
        /** . . . usual render() content instructions . . . **/
        /**                    . . .                          **/
    }

    // Draw the left scene ( ORTHOGRAPHIC projection ):
    {
        projectionMatrix = ortho(left, right, bottom, ytop, zNear, zFar);
        gl.clearColor(1, 1, 1, 1);
        renderScene(0, 0, width/2, height, projectionMatrix);
    }
    // Draw the right scene ( PERSPECTIVE projection ):
    {
        const projectionMatrix  =  perspective(fovy, aspect, zNear, zFar);
        gl.clearColor(1, 1, 1, 1);
        renderScene(width/2, 0,  width/2, height, projectionMatrix);
    }
    requestAnimFrame(render);
}
```

So, the two scenes' programs receives just different projection matrixes.

> The same near and far sliders are used to set both projections.

Task 5

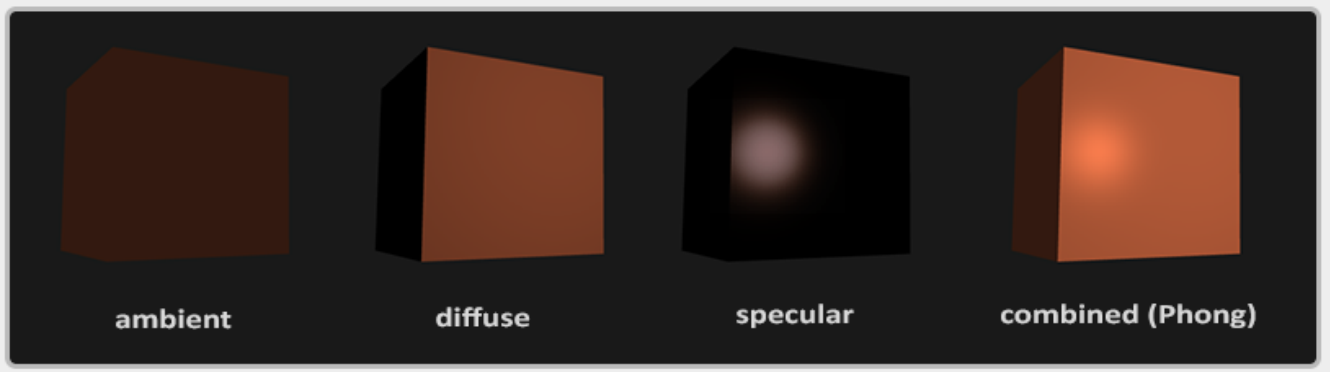
Introduce a light source, replace the colors by the properties of the material (your choice) and assign to each vertex a normal.

Solution T5

We add a point light source (directional, 4^{th} component = 0.0):

```
var lightPosition     = vec4(-1.0,  1.0,  1.0,  0.0 );  // directional source in the
                                                        // direction (-1.0, 1.0, 1.0).
var lightAmbient      = vec4( 0.2,  0.2,  0.2,  1.0 );
var lightDiffuse      = vec4( 0.8,  0.8,  0.8,  1.0 );
var lightSpecular     = vec4( 1.0,  1.0,  1.0,  1.0 );

/*Here we define a small amount of gray ambient reflectivity, yellow diffuse
properties, and white specular reflections.*/
var materialAmbient   = vec4( 1.0,  0.0,  1.0,  1.0 );
var materialDiffuse   = vec4( 1.0,  0.8,  0.0,  1.0);
var materialSpecular  = vec4( 1.0,  0.8,  0.0,  1.0 );
var materialShininess = 32.0;                       // specular component's shininess
```



The normals for each vertexes are computed and pushed in the `normalsArray` buffer in the `quad()` function;

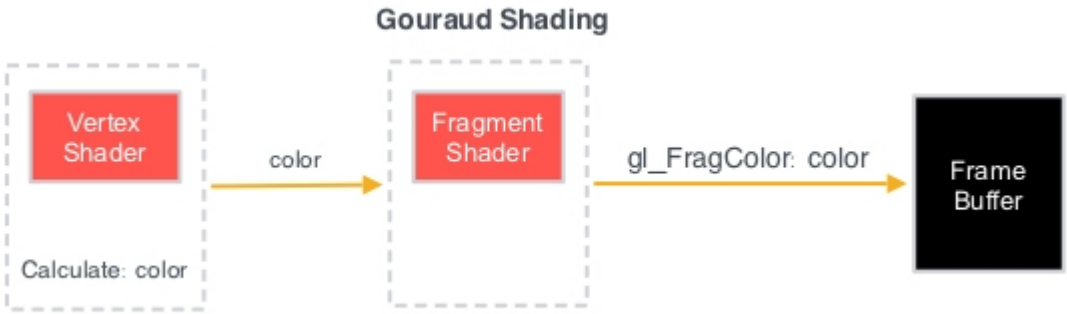
Task 6

Implement both the Gouraud and the Phong shading models, with a button switching between them.

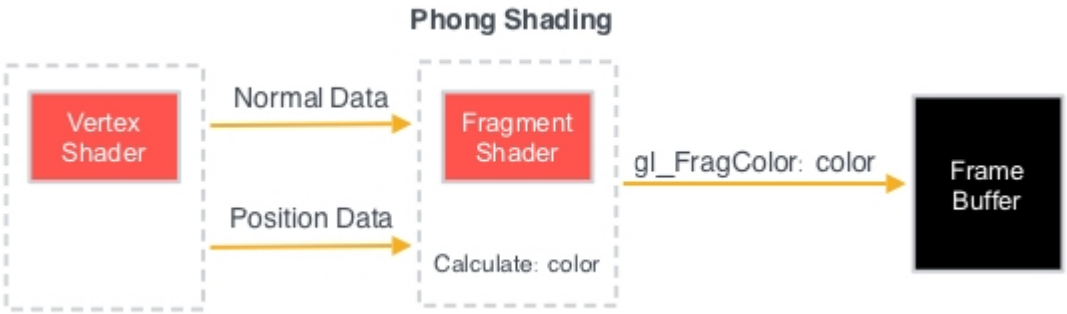
Solution T6

The switching is done by selecting the desired shading model to switch on. A select input element is used instead of the button initially (at this point I had no time to change/fix it).

The Gouraud Shading Model is less realistic, but more efficient in terms of calculations (*per-vertex color computation*). the vertex shader must determine a color for each vertex and pass the color as an **out** variable to the fragment shader.



On the other side, Phong Shading Model much more realistic, but less efficient in terms of computations (*per-fragment color computation*). The vertex shader provides the normal and position data as **out** variables to the fragment shader. The fragment shader then interpolates these variables and computes the color. CPU - GPU data transferring overloading is possible for complex object models.



Implementation:

In the HTML file we have two pairs of shaders for the two Shading Models.

The `function setShadingModel(shadingModel)` is called whenever a different Shading Model is selected (*Phong Model - by default*). Also it is called in the `init()` function with the `defaultShadingModel`. This function is responsible for the selecting the right shaders to use for the selected Shading Model.

Task 7

Add a procedural texture (your choice) on each face, with the pixel color a combination of the color computed using the lighting model and the texture.

Solution T7

The *Procedural texture mapping* requires much less memory as compared to *image based texture mapping*. There is no image to download or store in RAM or in the GPU's memory. Instead, it is calculated at rendering time for each individual fragment. If the calculations are complex, rendering speeds become slower (try to switch the greatest texture size - `Texture Size` range slider).

The basic steps *procedural texture mapping* that was followed in our case (checkboard texture with different sizes):

1. When building the model:

    1. Assign an appropriate *texture coordinate*, (s,t), to each vertex of a triangle. (This can be skipped if the geometry of a model is used for texture mapping inputs.)
2. JavaScript pre-processing for a canvas rendering:
3. JavaScript setup each time a model is rendered using a procedural texture map;
    1. Select the correct *shader program* with `gl.useProgram()`.
4. Shader program
    1. In the *vertex shader*, create a `varying` variable that will interpolate the *texture coordinates* across the surface of a triangle. (Or interpolate some other property of the model across the face.)
    2. In the *fragment shader*, use the *texture coordinates* (or some other interpolated value) to **calculate** a color.