3D Viewing

Learning Outcomes

At the end of this session, the students should be able to:

1. modify the camera and perform viewing transformations;
2. apply model transformations; and
3. implement projection transformation.

Content

Viewing Transformation

This requires indicating of viewpoint. In WebGL, this can be done by rotating the camera with *lookAt()* function and applying this to the **view matrix** or **camera transformation matrix**.

The following must be indicated as content of the **view matrix**:

1) **Position of the camera/viewpoint/eye point**
2) **Reference point/center point**
3) **Up vector (where the surface is facing)**

```
const viewMatrix = glMatrix.mat4.create();
glMatrix.mat4.lookAt(viewMatrix, viewpoint, eye/center point, up vector);
```

Model Transformation

Modeling transformation involves translating, rotating, and scaling the object based on the intended position in the scene. The model matrix is computed and set to perform these transformations on every point of the object into the correct space.

Here are some sample transformations that can be performed:

```
// setup model matrix;
    const modelMatrix = glMatrix.mat4.create();
    glMatrix.mat4.rotateZ(modelMatrix, modelMatrix, glMatrix.glMatrix.toRadian(0));
    glMatrix.mat4.translate(modelMatrix, modelMatrix, [-0.25,-0.25,-0.25,0]);
```

Projection Transformation

The projection transformation deals with specification of the view frustum. This requires the declaration and specification of the projection matrix. There are two types of projection: parallel and perspective. In parallel projection, object retains the same shape and size regardless of distance from viewpoint. While in perspective projection, objects become smaller as they move farther. This is because the near and far planes of the frustum are not equal. The far plane in the perspective view volume is larger than the near plane making the objects in the far plane appear smaller.

```
const projectionMatrix = glMatrix.mat4.create();

var left = -1;
var right = 1;
var bottom = -1;
var top = 1;
var near = 1;
var far =10;
glMatrix.mat4.ortho(projectionMatrix, left, right, bottom, top, near, far);
```

Modification of Shader Program and Pointer Declaration

The vertex shader program must be modified to anticipate the transformations performed before rendering the objects in the scene. Take note that the u_transformation_matrix mentioned in this code is from the previous topic about affine transformations (translate, scale, rotate).

```
//For view, model, and projection transformations
   uniform mat4 u_model_matrix;
   uniform mat4 u_view_matrix;
   uniform mat4 u_projection_matrix;

   void main() {
     //Apply the transformations to the object to be rendered
     gl_Position = u_transformation_matrix * u_view_matrix * u_model_matrix * u_projection_matrix *
                   a_position;
   }
```

Pointers to access the declared transformation matrices must be declared and loaded.

```
gl.uniformMatrix4fv(uModelMatrixPointer, false, new Float32Array(modelMatrix));
gl.uniformMatrix4fv(uViewMatrixPointer, false, new Float32Array(viewMatrix));
gl.uniformMatrix4fv(uProjectionMatrixPointer, false, new Float32Array(projectionMatrix));
```

This part is implemented in **drawScene()** function before executing the drawing functions.

gl.DEPTH_TEST must also be enabled when drawing objects in three-dimensional space. In the sample code provided, there are two objects (squares) to be drawn. The two objects overlap. The object with a covered section compared to the object fully drawn is dependent on the sequence of the function calls to the **draw** function.

In the main function:

```
const square1_vertices = [
  -0.75,-0.5,-0.5,1,
  -0.75, 0.5,-0.5,1,
  0.25, 0.5,-0.5,1,
  0.25,-0.5,-0.5,1
];
const color1 = [0, 1, 0, 1]; //GREEN

const square2_vertices = [
  -0.5,-0.5,-0.25,1,
  -0.5, 0.5,-0.25,1,
  0.5, 0.5,-0.25,1,
  0.5,-0.5,-0.25,1
];
const color2 = [0, 0, 1, 1]; //BLUE
```

In the drawScene function, **square1** with **color1** (green) is drawn first and then **square2** with **color2** (blue). Make sure to enable the pointer declared for accessing the position of the vertices.

```
// enable attribute array
gl.enableVertexAttribArray(aPositionPointer);

...

// load shader variables
gl.uniform4fv(colorLocation,color1);

gl.bindBuffer(gl.ARRAY_BUFFER, square1Buffer);
gl.vertexAttribPointer(aPositionPointer, 4, gl.FLOAT, false, 0, 0);
gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);

gl.uniform4fv(colorLocation, color2);
gl.bindBuffer(gl.ARRAY_BUFFER, square2Buffer);
gl.vertexAttribPointer(aPositionPointer, 4, gl.FLOAT, false, 0, 0);
gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
```

Exercise5

A ***programming exercise*** that integrates all the concepts in WebGL outlined in this material *(e.g. view, model, projection, and transformation matrices, buffer object, functions)* written as an ***html file***.

Submission

Submit your well-documented code on the provided assignment in Google Classroom. Name your file as **SurnameEx5.html** (ex. **DelaCruzEx5.html**)

Required competencies: viewing, model, projections transformations, 3D space, function declaration and calls, buffer objects

Instructions: Given the sample code **WebGL_4.html,** modify the code to apply viewing, model, and projection transformations on the object you have created in the previous exercise. The affine transformations that you were able to implement (translate, rotate, and scale) must still be present in this exercise and should still be working on your 3D object. Instead of a 2D animal, create depth by rendering a 3D version of the triangulated animal assigned to you. Provide user-friendly interface for performing affine transformations and view/mode/projection transformations. There must be an interface (SLIDERS) for changing the X,Y,Z components for the **viewpoint, eye point,** and **up** vector. There must also be interface for easily changing from parallel to perspective and vice versa as well as specifying the left, right, top, bottom, near, and far planes for the projection view volume. For the model transformation, no interface is needed to be specified but you must provide a transformation to the model matrix in the code different from the sample provided in this handout and in the sample code.

References

[1] **WebGL model view projection**. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection

[2] WebGLRenderingContext.depthFunc(). Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/depthFunc