Ahsanullah University of Science and Technology (AUST)

Department of Computer Science and Engineering

LABORATORY MANUAL

Course No. : CSE4204

Course Title: Computer Graphics Lab

For the students of 4th Year, 2nd semester of

B.Sc. in Computer Science and Engineering program

Table of Contents

Content	Page no.
Course Objective	2
Preferred Tools	2
Reference Books	2
Administrative Policy of the Laboratory	2
Lab 1 – Introduction to Graphics API and Shading Language	3
Lab 2 – Different Qualifiers of GLSL Data Types and Interactive Computer Graphics	8
Lab 3 – Index Buffer and Transformation Matrices	17
Lab 4 – Perspective and Camera Transformation	22
Lab 5 – Texture Mapping and Animation	26
Lab 6 – Surface Shading	31
Mid Term Examination	34
Lab Final Examination	34
Project	34

Course Objective

Course objectives for the course are -

1. To learn graphics APIs, such as OpenGL/ WebGL; to write a simple program to basic primitives with coordinates and colors; to understand the basic concepts of shading language and its important parts: vertex and fragment shader; to understand the basic graphics pipeline.
2. To learn basics about GLS, such as – different types of data used in GLSL such as attribute, uniform and varying; implementing mouse interaction in WebGL.
3. To apply index buffer and to understand its necessity; to work with 3D objects and to use 3D transformation matrices
4. To be able to apply the theoretical knowledge of viewing and projection in WebGL.
5. TO apply the theoretical knowledge of texture in WebGL and implement basic animation in WebGL.
6. To implement the theoretical knowledge of lighting and surface shading in WebGL and implement the Phong shading using GLSL.

Preferred Tools

1. Language: GLSL (OpenGL Shader Language) and JavaScript
2. API: Modern WebGL (Modern OpenGL ES)
3. Editor: Notepad++, VS Code

Reference Books

Introduction to Computer Graphics (*6 January 2018, Version 1.2*) by David J. Eck.
[Link: http://math.hws.edu/graphicsbook/index.html]

Administrative Policy of the Laboratory

1. Class assessment tasks must be performed by students individually, without help of others.
2. Viva for each program will be taken and considered as a performance.
3. Plagiarism is strictly forbidden.

Lab 1 – Introduction to Graphics API and Shading Language

1.1. Objective:

Objectives of this topics are – (1) to learn about graphics APIs, such as OpenGL/ WebGL; (2) to write a simple program to basic primitives with coordinates and colors; (3) to understand the basic concepts of shading language and its important parts: vertex and fragment shader; (4) to understand the basic graphics pipeline.

1.2. The Graphics API – WebGL:

OpenGL is a family of computer graphics APIs that is implemented in many graphics hardware devices. There are several versions of the API, and there are implementations, or "bindings" for several different programming languages. Versions of OpenGL for embedded systems such as mobile phones are known as OpenGL ES. a graphics API that was introduced in 1992 and has gone through many versions and many changes since then. WebGL is a version for use on Web pages. OpenGL can be used for 2D as well as for 3D graphics, but it is most commonly associated with 3D.

WebGL is A 3D graphics API for use on web pages. WebGL programs are written in the JavaScript programming language and display their images in HTML canvas elements. WebGL is based on OpenGL ES, the version of OpenGL for embedded systems, with a few changes to adapt it to the JavaScript language and the Web environment.

1.3. The Graphics Pipeline:

OpenGL 1.1 used a fixed-function pipeline for graphics processing. Data is provided by a program and passes through a series of processing stages that ultimately produce the pixel colors seen in the final image. The program can enable and disable some of the steps in the process, such as the depth test and lighting calculations. But there is no way for it to change what happens at each stage. The functionality is fixed.

OpenGL 2.0 introduced a programmable pipeline. It is a processing pipeline in which some of the processing stages can or must be implemented by programs. Data for an image passes through a sequence of processing stages, with the image as the end product. The sequence is called a "pipeline." Programmable pipelines are used in modern GPUs to provide more flexibility and control to the programmer. The programs for a programmable pipeline are known as shaders and are written in a shader programming language such as GLSL. It

became possible for the programmer to replace certain stages in the pipeline with their own programs. This gives the programmer complete control over what happens at that stage. In OpenGL 2.0, the programmability was optional; the complete fixed-function pipeline was still available for programs that didn't need the flexibility of programmability. WebGL uses a programmable pipeline, and it is mandatory. There is no way to use WebGL without writing programs to implement part of the graphics processing pipeline.

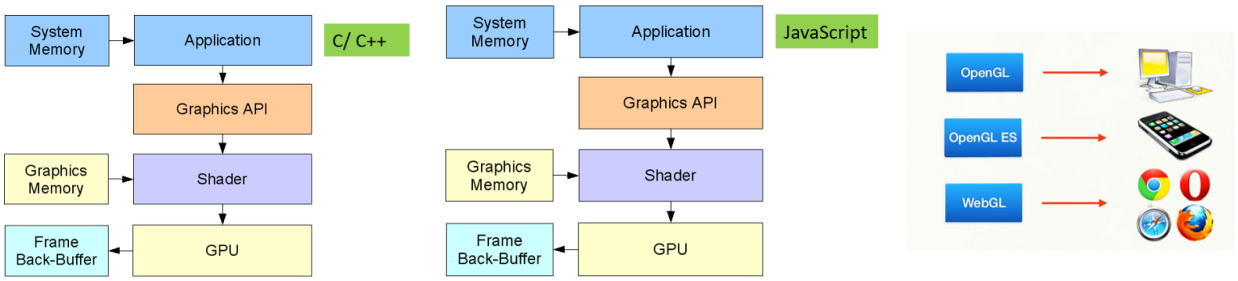


***Figure 1.1**: Source: Difference between OpenGL, OpenGL ES and WebGL. (source: https://ict.senecacollege.ca/~chris.szalwinski/archives/gam670.071/content/shadr_p.html)*

The programs that are written as part of the pipeline are called shaders. For WebGL, you need to write a vertex shader, which is called once for each vertex in a primitive, and a fragment shader, which is called once for each pixel in the primitive. Aside from these two programmable stages, the WebGL pipeline also contains several stages from the original fixed-function pipeline. For example, the depth test is still part of the fixed functionality, and it can be enabled or disabled in WebGL in the same way as in OpenGL 1.1.

1.3.1. Vertex Shader: A shader program that will be executed once for each vertex in a primitive. A vertex shader must compute the vertex coordinates in the clip coordinate system. It can also compute other properties, such as color.

1.3.2. Fragment Shader: A shader program that will be executed once for each pixel in a primitive. A fragment shader must compute a color for the pixel, or discard it. Fragment shaders are also called pixel shaders.

1.4. A Graphics Program:

There are two sides to any WebGL program:
Part – 1: written in JavaScript
Part – 2: written in GLSL, a language for writing "shader" programs that run on the GPU.

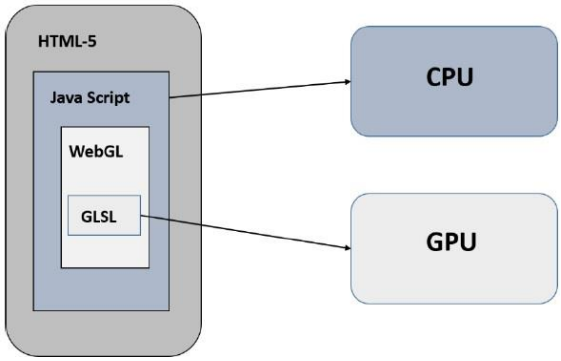


Figure 1.2: *Parts of a WebGL program. (source: https://www.tutorialspoint.com/webgl/webgl_quick_guide.htm)*

1.4.1. Standard steps for a WebGL program: A basic WebGL program follows the basic steps as listed below.

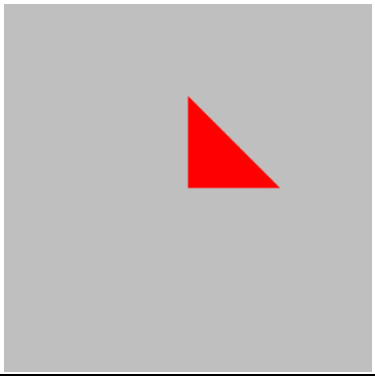
Step 1 – Prepare the canvas and get WebGL rendering context
Step 2 – Create and compile Shader programs
Step 3 – Associate the shader programs with buffer objects
Step 4 – Define the geometry and store it in buffer objects
Step 5 – Drawing the required object

Code: 1.1. Sample Code for drawing a triangle

```
<!-- saved from url=(0065)http://math.hws.edu/graphicsbook/source/webgl/simple-texture.html --
><!-- modified by Mohammad Imrul Jubair -->

<html>
<title>LAB-1: Intro</title>
<canvas id="webglcanvas" width="500" height="500"></canvas>

<script>

    var canvas = document.getElementById("webglcanvas");
    var  gl = canvas.getContext("webgl");

    var vertexShaderSource =
       `attribute vec3 a_coords;
        void main() {
            gl_Position = vec4(a_coords, 1.0);
        }`;

    var fragmentShaderSource =
       `void main() {
            gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
        }`;
```

```
    var vsh = gl.createShader( gl.VERTEX_SHADER );
    gl.shaderSource( vsh, vertexShaderSource );
    gl.compileShader( vsh );

    var fsh = gl.createShader( gl.FRAGMENT_SHADER );
    gl.shaderSource( fsh, fragmentShaderSource );
    gl.compileShader( fsh );

    var prog = gl.createProgram();

    gl.attachShader( prog, vsh );
    gl.attachShader( prog, fsh );
    gl.linkProgram( prog );
    gl.useProgram(prog);

    var a_coords_location = gl.getAttribLocation(prog, "a_coords");

    var coords = new Float32Array( [0.0, 0.0, 0.0,
                                    0.0, 0.5, 0.0,
                                    0.5, 0.0, 0.0] );

    var a_coords_buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);
    gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
    gl.vertexAttribPointer(a_coords_location, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(a_coords_location);
    gl.clearColor(0.75, 0.75, 0.75, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, 3);

</script></html>
```

Output:



1.5. Task:

a) Write a program to draw quad using two triangles.
b) Write a program that utilizes different drawing primitives to draw a multiple triangle.

Lab 2 – Different Qualifiers of GLSL Data Types and Interactive Computer Graphics

2.1. Objective:

Objective of the topics is to learn (1) different types of data used in GLSL such as attribute, uniform and varying; (2) implementing mouse interaction in WebGL.

2.2. Data Flow in the Graphics Pipeline:

The JavaScript side of the program sends values for attributes and uniform variables to the GPU and then issues a command to draw a primitive. The GPU executes the vertex shader once for each vertex. The vertex shader can use the values of attributes and uniforms. It assigns values to *gl_Position* and to any varying variables that exist in the shader. After clipping, rasterization, and interpolation, the GPU executes the fragment shader once for each pixel in the primitive. The fragment shader can use the values of varying variables, uniform variables, and *gl_FragCoord*. It computes a value for gl_FragColor. The following diagram summarizes the flow of data:

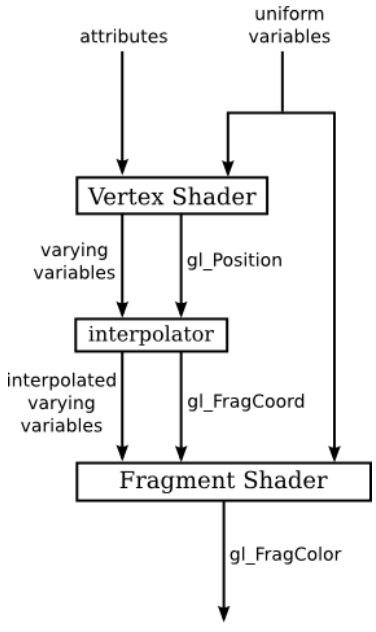


Figure 2.1: Data flow in Graphics Pipeline. (source: http://math.hws.edu/graphicsbook/c6/s1.html#webgl.1.5)

2.3. Attribute

An attribute can take a different value for each vertex in a primitive. The basic idea is that the complete set of data for the attribute is copied in a single operation from a JavaScript array into memory that is accessible to the GPU. Unfortunately, setting things up to make that operation possible is non-trivial (see code 1.1).

2.4. Uniform and Varying

Global variable declarations in a vertex shader can be marked as attribute, uniform, or varying. A variable declaration with none of these modifiers defines a variable that is local to the vertex shader. Global variables in a fragment can optionally be modified with uniform or varying, or they can be declared without a modifier. A varying variable should be declared in both shaders, with the same name and type. This allows the GLSL compiler to determine what attribute, uniform, and varying variables are used in a shader program. This qualifier forms a link between a vertex shader and fragment shader for interpolated data. It can be used with the following data types – *float, vec2, vec3, vec4*, *mat2, mat3, mat4*, or arrays. (see code 2.1 and 2.2). The summary on attribute, uniform and varying qualifiers are listed in the Figure 2.2.

Sr.No.	Qualifier & Description
1	**attribute** This qualifier acts as a link between a vertex shader and OpenGL ES for per-vertex data. The value of this attribute changes for every execution of the vertex shader.
2	**uniform** This qualifier links shader programs and the WebGL application. Unlike attribute qualifier, the values of uniforms do not change. Uniforms are read-only; you can use them with any basic data types, to declare a variable. **Example** − uniform **vec4** lightPosition;
3	**varying** This qualifier forms a link between a vertex shader and fragment shader for interpolated data. It can be used with the following data types − float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays. **Example** − varying **vec3** normal;

Figure 2.2: Attribute, uniform and varying qualifiers (source: https://www.tutorialspoint.com/webgl/webgl_quick_guide.htm)

Code: 2.1. Sample Code for drawing a triangle using Uniform color

```
<!-- saved from url=(0065)http://math.hws.edu/graphicsbook/source/webgl/simple-texture.html --
> <!-- modified by Mohammad Imrul Jubair -->

<html>
<title>LAB-2: Uniform</title>
<canvas id="webglcanvas" width="500" height="500"></canvas>

<script>
    var canvas = document.getElementById("webglcanvas");
    var  gl = canvas.getContext("webgl");

    var vertexShaderSource =
       `attribute vec3 a_coords;
        void main() {
            gl_Position = vec4(a_coords, 1.0);
        }`;

    var fragmentShaderSource =

       `precision mediump float;
        uniform vec3 u_color;
        void main() {
            gl_FragColor = vec4(u_color, 1.0);
        }`;

    var vsh = gl.createShader( gl.VERTEX_SHADER );
    gl.shaderSource( vsh, vertexShaderSource );
    gl.compileShader( vsh );

    var fsh = gl.createShader( gl.FRAGMENT_SHADER );
    gl.shaderSource( fsh, fragmentShaderSource );
    gl.compileShader( fsh );

    var prog = gl.createProgram();

    gl.attachShader( prog, vsh );
    gl.attachShader( prog, fsh );
    gl.linkProgram( prog );
    gl.useProgram(prog);

    var a_coords_location = gl.getAttribLocation(prog, "a_coords");

    var coords = new Float32Array( [0.0, 0.0, 0.0,
                                    0.0, 0.5, 0.0,
```

```
                                    0.5, 0.0, 0.0] );

    var a_coords_buffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);
    gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
    gl.vertexAttribPointer(a_coords_location, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(a_coords_location);

    var u_color_location = gl.getUniformLocation(prog, "u_color");
    var color = new Float32Array( [0.5, 0.7, 0.3] );
    gl.uniform3fv(u_color_location,  color);

    gl.clearColor(0.75, 0.75, 0.75, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, 3);

</script></html>
```

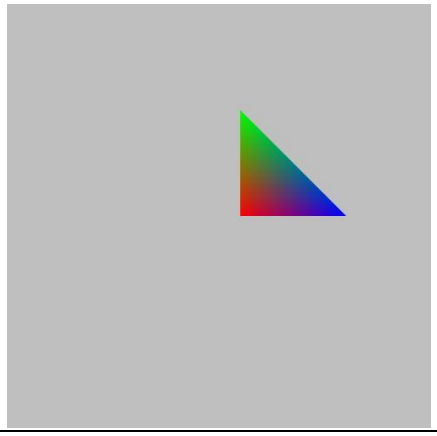
Code: 2.2. Sample Code for drawing a triangle using varying (interpolated) color

```
<!-- saved from url=(0065)http://math.hws.edu/graphicsbook/source/webgl/simple-texture.html --
> <!-- modified by Mohammad Imrul Jubair -->

<html>
<title>LAB-2: Varying </title>
<canvas id="webglcanvas" width="500" height="500"></canvas>

<script>
    var canvas = document.getElementById("webglcanvas");
    var  gl = canvas.getContext("webgl");

    var vertexShaderSource =
       `attribute vec3 a_coords;
        attribute vec3 a_colors;
        uniform float u_shift;
        varying vec3 v_color;

        void main() {
            gl_Position = vec4(a_coords.x + u_shift, a_coords.y, a_coords.z, 1.0);
            v_color = a_colors;
        }`;

    var fragmentShaderSource =
       `precision mediump float;
```

```
    varying vec3 v_color;
    void main() {
        gl_FragColor = vec4(v_color, 1.0);
    }`;

var vsh = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource( vsh, vertexShaderSource );
gl.compileShader( vsh );

var fsh = gl.createShader( gl.FRAGMENT_SHADER );
gl.shaderSource( fsh, fragmentShaderSource );
gl.compileShader( fsh );

var prog = gl.createProgram();

gl.attachShader( prog, vsh );
gl.attachShader( prog, fsh );
gl.linkProgram( prog );
gl.useProgram(prog);

var a_coords_location = gl.getAttribLocation(prog, "a_coords");

var coords = new Float32Array( [0.0, 0.0, 0.0,
                                0.0, 0.5, 0.0,
                                0.5, 0.0, 0.0] );

var a_coords_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);
gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
gl.vertexAttribPointer(a_coords_location, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(a_coords_location);

var u_shift_location = gl.getUniformLocation(prog, "u_shift");
var shift = 0.0;
gl.uniform1f(u_shift_location, shift);

a_colors_location = gl.getAttribLocation(prog, "a_colors");
var colors = new Float32Array( [1.0, 0.0, 0.0,
                                0.0, 1.0, 0.0,
                                0.0, 0.0, 1.0] );

a_colors_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, a_colors_buffer);
gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);
gl.vertexAttribPointer(a_colors_location, 3, gl.FLOAT, false, 0, 0);
```

```
    gl.enableVertexAttribArray(a_colors_location);
    gl.clearColor(0.75, 0.75, 0.75, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, 3);

    canvas.onmousedown = function ()
    {
    shift = shift + 0.1;
    gl.uniform1f(u_shift_location, shift);

    gl.clearColor(0.75, 0.75, 0.75, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
    gl.drawArrays(gl.TRIANGLES, 0, 3);
    };
</script>
</html>
```

Output:



2.5. Mouse Interaction:

The *mousedown* event is fired at an Element when a pointing device button is pressed while the pointer is inside the element. We can utilize this to call draw functions to render on the canvas based on clicking. The following code segment from code 2.2 is responsible for invoking draw function for mouse click.

```
canvas.onmousedown = function ()
{
shift = shift + 0.1;
gl.uniform1f(u_shift_location, shift);
```

```
gl.clearColor(0.75, 0.75, 0.75, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);
};
```

2.6. GLSL Control Statements and Built-in functions

2.6.1. Control Statements: GLS is a C-like language. We can use our basic concepts of control statements (e.g. if-else, switch-case, loop, etc). A helpful resource is: https://www.shaderific.com/glsl-statements. An example of IF-ELSE in vertex shader is provided below.

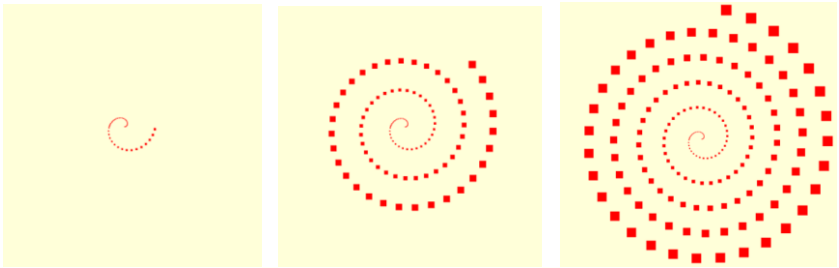
```
attribute vec3 a_coords;
attribute vec3 a_colors;
uniform float u_shift;
varying vec3 v_color;

void main() {
    if (u_shift < 0.7)
        gl_Position = vec4(a_coords.x - u_shift,
                           a_coords.y,
                           a_coords.z,
                           1.0);
    else
        gl_Position = vec4(a_coords.x,
                           a_coords.y,
                           a_coords.z,
                           1.0);
    v_color = a_colors; }
```

2.6.2. Built-in Functions: GLSL provides a significant number of built-in functions and you should be familiar with them. Please see this page (https://www.shaderific.com/glsl-functions) to get idea regarding some of these functions. An example of a vertex shader code segment that uses a built-in function *clamp()* is provided below.

```
void main() {
    gl_Position = vec4(clamp(a_coords.x - u_shift, -0.5, 1.0),
                       a_coords.y,
                       a_coords.z,
                       1.0);
```

2.7. Task:

2.7.1. Task 1: Create a 2D spiral. For each click, the spiral will keep increasing. The outer points will be bigger than the inner points depending on the distance from the center. Note that, you have to send 2D data to the GPU from CPU. The following figure shows different situation of the canvas after several mouse clicking (from left to right).



Hints:

- Use *gl_PointSize* in the vertex shader to fix the size of the point. (Example: https://www.tutorialspoint.com/webgl/webgl_drawing_points.htm)
- Use GLSL *distance()* function to calculate the distance between a vertex and the center in the vertex shader.
- To generate the vertices for a 2D spiral in CPU, you can use JavaScript's Math library to apply the formula, e.g. *Math.cos()*. Use *push()* function to build up an array of vertices of the spiral using a loop.
- Appy optimistically while using/ reusing vertex buffer.

2.7.2. Task 2: Create a 2D flower based on *Rhodonea* curve. Use *GL_POINTS* in your draw call. For each click, the arrangement of the petals will keep changing. Also, the color of the points will be alternated between green and red for each click. Note that, you have to send 2D data to the shader from CPU.

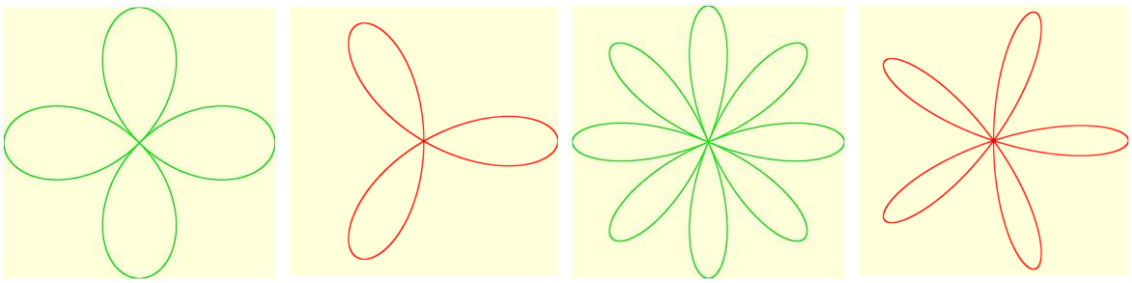


Figure: Situation of the canvas after several mouse clicking (from left to right).

Hints:

- Rose in math: https://en.wikipedia.org/wiki/Rose_(mathematics)

- You can track odd/ even clicking in the shader to alternate color.
- To generate the vertices for a 2D spiral in CPU, you can use JavaScript's Math library to apply the formula, e.g. *Math.cos().* Use *push()* function to build up an array of vertices of the spiral using a loop.
- Appy optimistically while using/ reusing vertex buffer.

2.7.3. Task 3: Create a 2D scenario (model) using your creativity. The model has to be created using 2D triangle mesh. Apply per-vertex color on your model. Integrate a keyboard interaction having at least one GLSL control statement (and/or built-in function) inside the shader.

Note:

- Your mesh must have at least 45 vertices in total.
- You can use gl.TRIANGLES and/or gl.TRIANGLE_STRIP and/or gl.TRIANGLE_FAN.

Lab 3 – Index Buffer and Transformation Matrices

3.1. Objective:

Objective of the topic is (1) to learn about index buffer and its necessity; (2) to work with 3D objects and (3) to use 3D transformation matrices.

3.2. Index Buffer:

The index buffer contains integers, three for each triangle in the mesh, which reference the various attribute buffers (position, colour, UV coordinates, other UV coordinates, normal, etc.). It's a little bit like in the OBJ file format, with one huge difference: there is only ONE index buffer. This means that for a vertex to be shared between two triangles, all attributes must be the same.

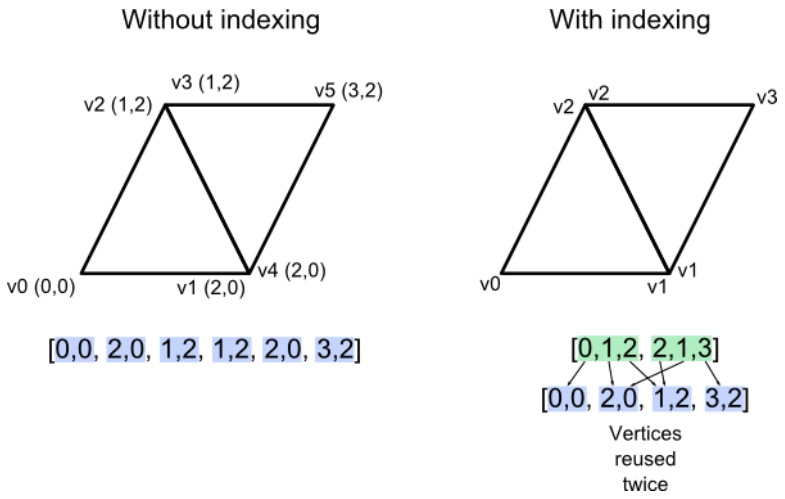


Figure 3.1: *Concept of the index buffer (source: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing/)*

Here is an example:

Code: 3.1. Implementing index buffer

```
var 17coords = new Float32Array( [
            -0.5, -0.5,  0.0, //v0
             0.5, -0.5,  0.0, //v1
             0.5,  0.5,  0.0, //v2
            -0.5,  0.5,  0.0  //v3
                           ] );

var colors = new Float32Array( [
```

```
                    1.0, 0.0, 0.0, //color at v0
                    0.0, 1.0, 0.0, //color at v1
                    0.0, 0.0, 1.0, //color at v2
                    1.0, 1.0, 0.0 //color at v3
                                ] );

    var indices = new Uint8Array([0, 1, 2, 0, 2, 3]);
    var bufferInd = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, bufferInd);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
```

In the above example, we defined the geometry and colors in the *coords* and *colors* arrays. We define the indices in and store it into the buffer so that GPU can render accordingly (shown in highlights)

3.2. 3D Transformation Matrices:

Basic transformations are: scaling, shearing, rotation, reflection and translation. Transformation is applied through the concept of linear transformation. Which is an operation of taking a vector and produces another vector by a simple matrix multiplication. These matrices are called transformation matrices.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y \\ a_{21}x + a_{22}y \end{bmatrix}$$

Some examples, we can use the following matrix to perform scaling in 2D.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Code: 3.2. Implementing 3D Rotation Matrix

```
<!-- saved from url=(0065)http://math.hws.edu/graphicsbook/source/webgl/simple-texture.html --
>
<!-- modified by Mohammad Imrul Jubair -->
<html>
<title>LAB-3: Transformation Matrix </title>
<canvas id="webglcanvas" width="500" height="500"></canvas>
<script>

    var canvas = document.getElementById("webglcanvas");
    var  gl = canvas.getContext("webgl");
```

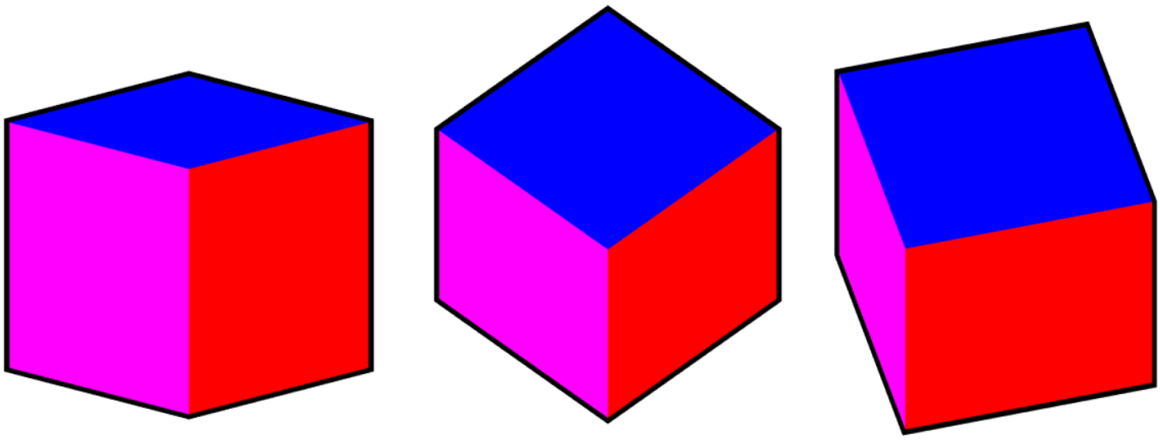
```
var vertexShaderSource =
   `attribute vec3 a_coords;
    attribute vec3 a_colors;
    uniform mat4 u_RotY;
    uniform mat4 u_RotX;
    varying vec3 v_color;

    void main() {
        gl_Position = u_RotX*u_RotY*vec4(a_coords, 1.0);
        v_color = a_colors;
    }`;

var fragmentShaderSource =
   `precision mediump float;
    varying vec3 v_color;
    void main() {
        gl_FragColor = vec4(v_color, 1.0);
    }`;

var vsh = gl.createShader( gl.VERTEX_SHADER );
gl.shaderSource( vsh, vertexShaderSource );
gl.compileShader( vsh );

var fsh = gl.createShader( gl.FRAGMENT_SHADER );
gl.shaderSource( fsh, fragmentShaderSource );
gl.compileShader( fsh );

var prog = gl.createProgram();

gl.attachShader( prog, vsh );
gl.attachShader( prog, fsh );
gl.linkProgram( prog );
gl.useProgram(prog);

var a_coords_location = gl.getAttribLocation(prog, "a_coords");

var coords = new Float32Array( [
                                    // Coordinates……
                                ] );

var a_coords_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, a_coords_buffer);
gl.bufferData(gl.ARRAY_BUFFER, coords, gl.STATIC_DRAW);
gl.vertexAttribPointer(a_coords_location, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(a_coords_location);
```

```
var a_colors_location = gl.getAttribLocation(prog, "a_colors");
var colors = new Float32Array( [
                                \\Colors……
                                ] );

var a_colors_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, a_colors_buffer);
gl.bufferData(gl.ARRAY_BUFFER, colors, gl.STATIC_DRAW);
gl.vertexAttribPointer(a_colors_location, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(a_colors_location);

var indices = new Uint8Array([
                                \\ Indices……
                            ]);
var bufferInd = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, bufferInd);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);


var u_rotateY_location = gl.getUniformLocation(prog, "u_RotY");
var thetaY = 45;
var rad = thetaY*Math.PI/180;
var rotateYMatrix = new Float32Array( [
       Math.cos(rad),  0.0,  -Math.sin(rad),  0.0,
       0.0,            1.0,   0.0,            0.0,
       Math.sin(rad),  0.0,   Math.cos(rad),  0.0,
       0.0,            0.0,   0.0,            1.0] );

gl.uniformMatrix4fv(u_rotateY_location, false, rotateYMatrix);

var u_rotateX_location = gl.getUniformLocation(prog, "u_RotX");
var thetaX = 45;
var rad = thetaX*Math.PI/180;
var rotateXMatrix = new Float32Array( [
         1.0,  0.0,            0.0,            0.0,
         0.0,  Math.cos(rad),  Math.sin(rad),  0.0,
         0.0, -Math.sin(rad),  Math.cos(rad),  0.0,
         0.0,  0.0,            0.0,            1.0] );
gl.uniformMatrix4fv(u_rotateX_location, false, rotateXMatrix);

gl.clearColor(1.0, 1.0, 1.0, 1.0);
gl.enable(gl.DEPTH_TEST);
gl.enable(gl.CULL_FACE);
```

```
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.drawElements(gl.TRIANGLES, 3*12, gl.UNSIGNED_BYTE, 0);

</script>
</html>
```

3.7. Task:

Create 3D cube using index buffer. Provide different colors for different faces. You have to introduce border for the object as shown in the diagrams below. For each left click, the cube will be scaled up and for right click, it will be scaled down. For pressing right and down arrow keys, the cube will rotate (+ve) along Y and X axis respectively.



Hints:

- To draw the border, you can call the draw functions two times. One will draw the cube with black color, but being slightly scaled up. For the next call, the cube will be drawn with faces' colors, but not being scaled. This difference between scaling factors will be appeared as a border in the canvas.
- Two handle the colors for two different draw calls, you can use control statements inside shaders that will switch between different gl_FragColor.
- Be careful while using gl.COLOR_BUFFER_BIT for second draw call. Also, z-coordinates can be ignored for drawing the border.

Lab 4 – Perspective and Camera Transformation

4.1. Objective:

Objective of the topic is to apply the theoretical knowledge of viewing in WebGL

4.2. Projection Transformation Chain:

The stages of operations that need to be applied to view a vertex from perspective view volume to the viewport work as a chain, which are listed below.

1. Modeling transform: M_m
2. Camera Transformation: M_{cam}
3. Perspective: P
4. Orthographic Projection: M_{orth}
5. Viewport Transform: M_{vp}

This stage is applied as composite transformation which is –

$$\mathbf{p}_s = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}}\mathbf{p}_o$$

$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{M}_{\text{cam}}\mathbf{M}_{\text{m}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

We can implement this using the following vertex shader code 4.1.

Code: 4.1. Vertex shader source code with Model, View and Projection Matrix

```
var vertexShaderSource =

       `attribute vec3 a_coords;
        attribute vec3 a_colors;
        uniform mat4 u_RotY;
        uniform mat4 u_RotX;
        uniform mat4 u_Scale;
        uniform mat4 u_Trans;
        uniform mat4 u_Basis;
        uniform mat4 u_Eye;
        uniform mat4 u_Pers;
        varying vec3 v_color;

        void main() {
```

```
            mat4 M = u_Trans*u_RotX*u_RotY*u_Scale;
            mat4 V = u_Basis*u_Eye;
            mat4 P = u_Pers;
            mat4 MVP = P*V*M;
            gl_Position = MVP*vec4(a_coords, 1.0);
            v_color = a_colors;
        }`;
```

Here M, V and P are the modeling, view and perspective matrices respectively. Sample code for implementing modeling matrix is already shown in the previous lab (section – 3). The perspective and camera matrices are already covered in the theory class. However, the code for defining these two matrices is shown below (code 4.2. and 4.3.).

Code: 4.2. Implementing Projection Matrix

```
var aspect = 1.0;
var fov = 75.0;
var far = 10.0;
var near = 1.0;

var pa = 1.0/(aspect*Math.tan((fov/2)*Math.PI/180));
var pb = 1.0/(Math.tan((fov/2)*Math.PI/180));
var pc = -(far+near) / (far-near);
var pd = -(2.0*far*near) / (far-near);

    var persMat = new Float32Array( [pa,   0.0,  0.0,  0.0,
                                     0.0,  pb,   0,    0.0,
                                     0.0,  0.0,  pc,  -1.0,
                                     0.0,  0.0,  pd,   0.0] );
```

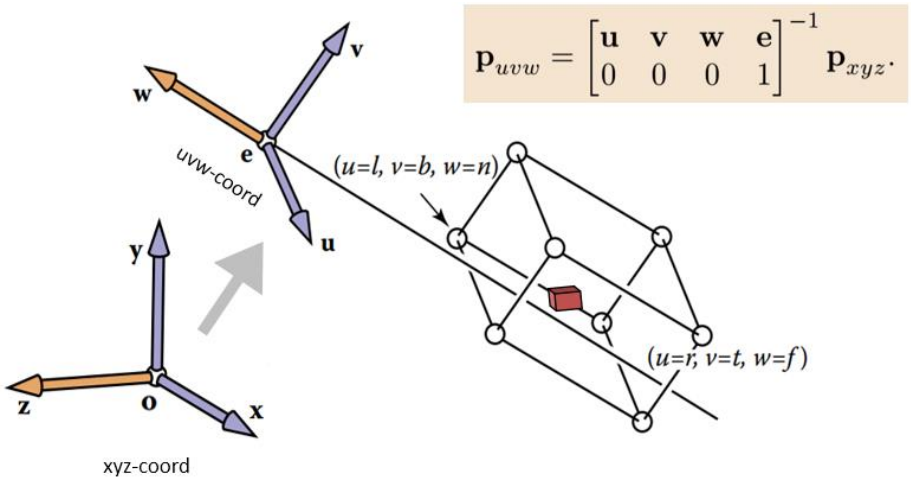
Here, *fov*, *near*, *far* and *aspects* are field-of-view, distance of near and far clipping plane and aspects ratios to define the viewing frustum.

Code: 4.3. Implementing Camera Matrix

```
var basisMat = new Float32Array([ 1, 0, 0, 0,
                                  0, 1, 0, 0,
                                  0, 0, 1, 0,
                                  0, 0, 0, 1]);

    var xe = 0.9;
    var ye = 0.0;
```

```
var ze = 0.1;

var eyeMat = new Float32Array([1,   0,   0,   0,
                               0,   1,   0,   0,
                               0,   0,   1,   0,
                              -xe,  -ye, -ze,  1]);
```

Here, *basisMat* and *eyeMat* defines the basis vectors and viewpoints positions of the frame coordinate system for the camera transformation respectively as shown the theory classes. Figure 4.1. also shows the concept.



$$\begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

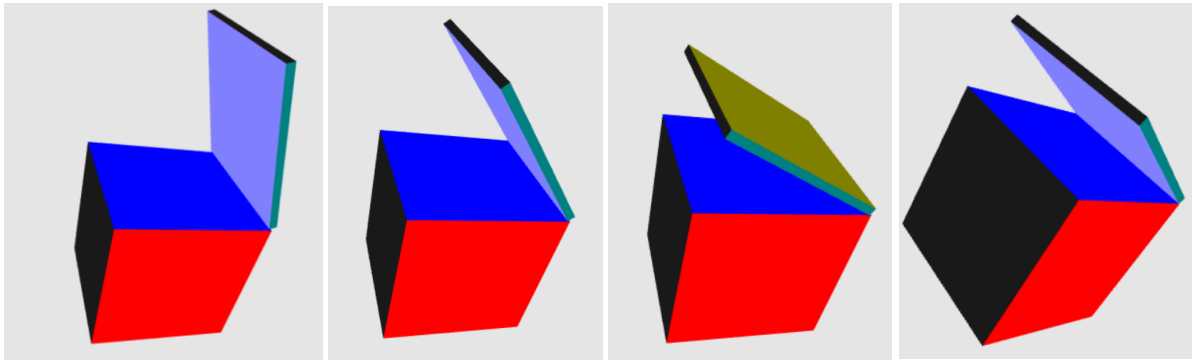
$$\mathbf{p}_{uvw} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xyz}.$$

Figure 4.1: *Camera Transformation matrix.*

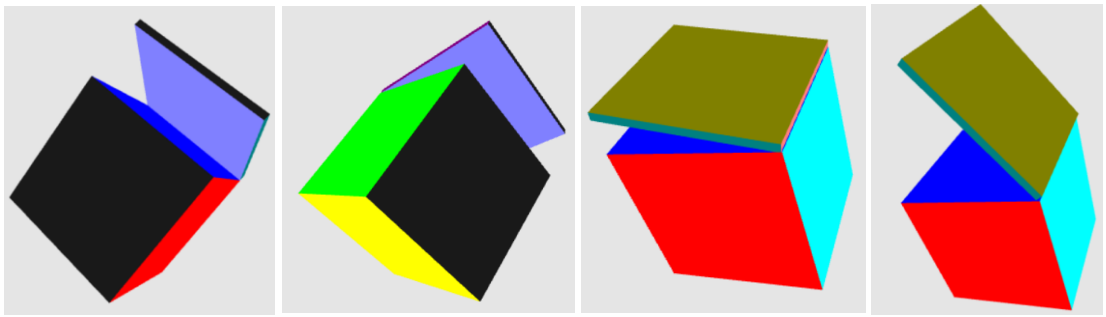
4.3. Task

Create a crate (box) with a lid. Both the crate and lid have the same coordinates; only lid is the scaled (skewed) version of the crate. You have freedom to choose colors. Attach two separate shader programs for the crate and the lid. Apply perspective projection on them. For each left and right arrow key pressing, the crate along with the lid (crate + lid) will rotate positive and negative degree along Y axis respectively. For each up and down arrow key pressing, only the lid will open and close respectively. See the following figure showing

different states of the crate and the lid. (a) – (c): Pressing down arrow key several times. (d) – (f): Pressing right arrow key several times. (g): Pressing left arrow key several times. (h): Pressing up arrow key several times.



(a) (b) (c) (d)



(e) (f) (g) (h)

Requirement:

1. Initially the (crate + lid) will create 30-degree w.r.t X and Y axis respectively.
2. Your program must be modularized into several functions. There must be two initiGL functions – which are initGL_1() and initGL_2(); and two draw functions which are draw_crate() and draw_lid() respectively.
3. Your program must have different functions for different transformation matrices, for perspective and for processing attribute buffers. The code skeleton contains the format of the function [see next section].
    a. *Transformation function:* For example, the rotate_Y(thetaY, loc) function will send a rotation matrix along Y axis with thetaY degree to the loc location that contains the matrix in shader. Here loc is defined inside initGL function.
    b. *Perspective function:* The function perspective (aspect, fov, near, far, loc) will take the necessary parameters (aspect, fov, near and far) and the location loc to be assigned.
    c. *Attribute buffer:* For example, passAttribData(data, att_buffer, loc) will take CPU data, the buffer att_buffer which is created inside initGL and the location loc where the data needs to be passed.

Lab 5 – Texture Mapping and Animation

5.1. Objective:

Objective of the topic is to (1) apply the theoretical knowledge of texture in WebGL and (2) implement basic animation in WebGL.

5.2. Texture Mapping:

Three-dimensional objects can be made to look more interesting and more realistic by adding a texture to their surfaces. A texture, in general, is some sort of variation from pixel to pixel within a single primitive. We will consider only one kind of texture: image textures. An image texture can be applied to a surface to make the color of the surface vary from point to point, something like painting a copy of the image onto the surface. Here is a picture that shows six the basic concept of texture mapping.

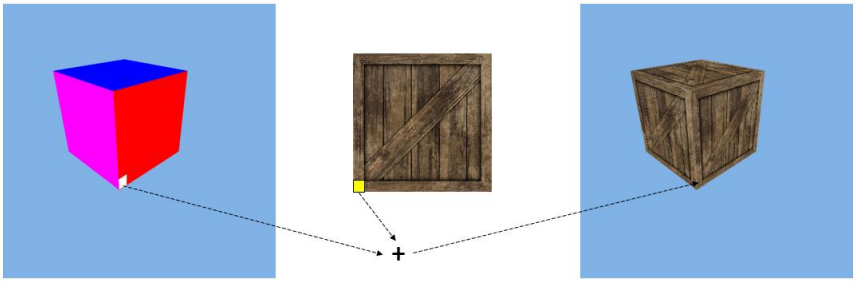
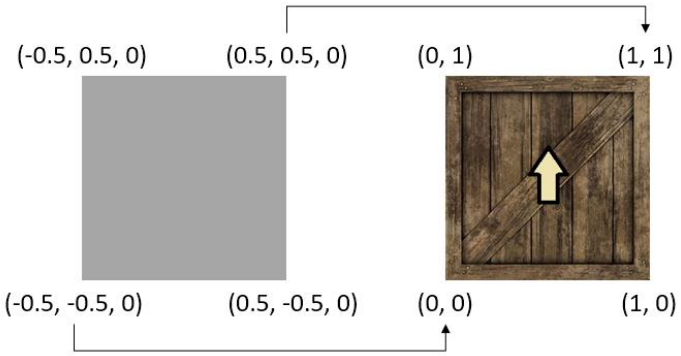


Figure 5.1: *Basic concept of texture mapping*

5.3. Texture Coordinates and Texture Lookup:

A texture image comes with its own 2D coordinates system. Traditionally, s used for the horizontal coordinate on the image and t is used for the vertical coordinate. The s coordinate is a real number that ranges from 0 on the left of the image to 1 on the right, while t ranges from 0 at the bottom to 1 at the top. Values of s or t outside of the range 0 to 1 are not inside the image, but such values are still valid as texture coordinates. Note that texture coordinates are not based on pixels. No matter what size the image is, values of s and t between 0 and 1 cover the entire image.



XY	UV
(-0.5, -0.5, 0)	(0, 0)
(0.5, -0.5, 0)	(1, 0)
(0.5, 0.5, 0)	(1, 1)
(-0.5, 0.5, 0)	(0, 1)

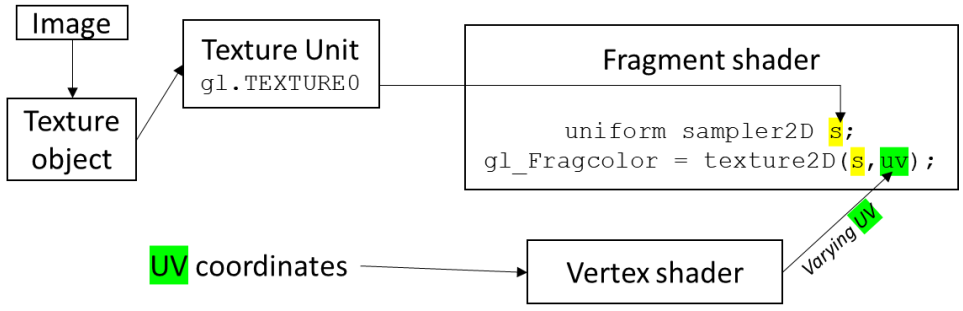
Figure 5.2: *An example of texture lookup*

There are some important terms that we should know to implement texture mapping:

- **Texture object** is a data structure that contains the color data for an image texture.
- **Sampling** is the process of computing a color from an image texture and texture coordinates.
- **A texture unit (TU)** is a hardware component in a GPU that does sampling.

5.4. Basic Implementation of Texture Mapping:

A basic implementation of texture mapping follows the stages as shown below followed by an example code to implement it.



a) Defining the texture coordinates

```
texCoords = new Float32Array( [0.0, 0.0,
                               1.0, 0.0,
                               1.0, 1.0,
                               0.0, 1.0] );
```

b) Passing texture coordinates to the fragment shader through vertex shader using varying:

```
var vertexShaderSource =

    `attribute vec3 a_coords;
     attribute vec3 a_colors;
     attribute vec2 a_texCoords;
     varying vec3 v_color;
     varying vec2 v_texCoords;

     void main() {

         gl_Position = vec4(a_coords, 1.0);
         v_color = a_colors;
         v_texCoords = a_texCoords;
     }`;
```

c) Applying texture mapping via texture2D() function that takes the texture object as a sampler and the interpolated texture coordinates.

```
var fragmentShaderSource =

    `precision mediump float;
     varying vec3 v_color;
     uniform sampler2D u_sampler_texture;
     varying vec2 v_texCoords;

     void main() {
         vec4 color = texture2D( u_sampler_texture, v_texCoords );
         gl_FragColor = color;
     }`;
```

d) Create a texture object and allocates some memory for the texture object.

```
textureObject = gl.createTexture();
```

e) Loading the image and converting it to store into texture data structure.

```
<img id="doorimage" src="crate2.jpg" width="0" height="0"></img>
```

```
loadTexture(textureObject, "doorimage");
```

```
function loadTexture(textureObject, imageID) {

    gl.bindTexture(gl.TEXTURE_2D, textureObject);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);

    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1);

    gl.texImage2D(gl.TEXTURE_2D,
                  0,
                  gl.RGBA,
                  gl.RGBA,
                  gl.UNSIGNED_BYTE,
                  document.getElementById(imageID));
}
```

f) Assigning a texture unit to use the texture object. We need to make the texture unit active, which is done by calling the function gl.activeTexture. The parameter is one of the constants gl.TEXTURE0, gl.TEXTURE1, gl.TEXTURE2, which represent the available texture units.

```
gl.activeTexture(gl.TEXTURE0);
```

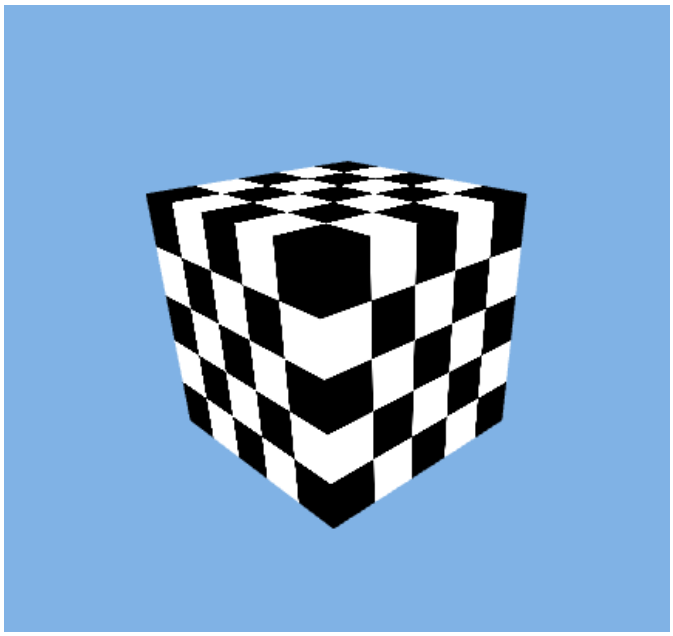
```
gl.bindTexture(gl.TEXTURE_2D, textureObject);
```

g) Assigning a texture unit along with the texture object to fragment shader so that *texture2d()* function can work properly.

```
u_sampler_texture_location = gl.getUniformLocation(prog, "u_sampler_texture");
```

```
gl.uniform1i(u_sampler_texture_location, 0);
```

The final result is –



5.5. Task:

Draw a box. The textures on its surface will swap between two images based on mouse clicking.

Lab 6 – Surface Shading

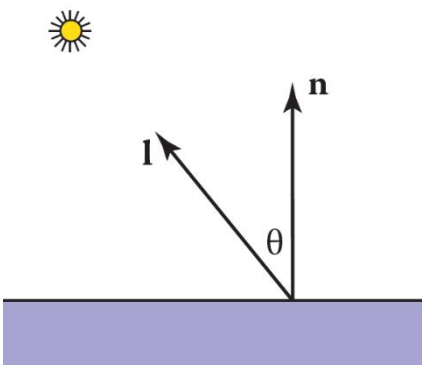
6.1. Objective:

Objective of the topic is to (1) apply the theoretical knowledge of lighting and shading in WebGL and (2) implement the Phong shading using GLSL.

6.1. Objective:

6.1. Surface Shading: To make objects appear to have more volume, it can help to use shading, i.e., the surface is "painted" with light.

6.2. Diffuse Shading: Many objects in the world have a surface appearance loosely described as "matte," indicating that the object is not at all shiny. Examples include paper, unfinished wood, and dry, unpolished stones. To a large degree, such objects do not have a color change with a change in viewpoint. For example, if you stare at a particular point on a piece of paper move while keeping your gaze fixed on that point, the color at that point will stay relatively constant. Such matte objects can be considered as behaving as Lambertian objects. A Lambertian object obeys Lambert's cosine law –



color c of a surface is proportional to the cosine of the angle between the surface normal (n) and the direction to the light source (l).

$$c \propto \cos\theta,$$

$$c \propto \mathbf{n} \cdot \mathbf{l},$$

Based on the law, the shading model is computed as –

$$c = c_r c_l \max(0, \mathbf{n} \cdot \mathbf{l})$$

6.3. Ambient Shading: A more common approach is to add an ambient term**.**

$$c = c_r \left(c_a + c_l \max\left(0, \mathbf{n} \cdot \mathbf{l}\right)\right)$$

6.4. Phong Shading: Some surfaces are essentially like matte surfaces, but they have highlights. Examples surfaces include polished tile floors, gloss paint, and whiteboards. In such cases, the highlights move across a surface as the viewpoint moves.

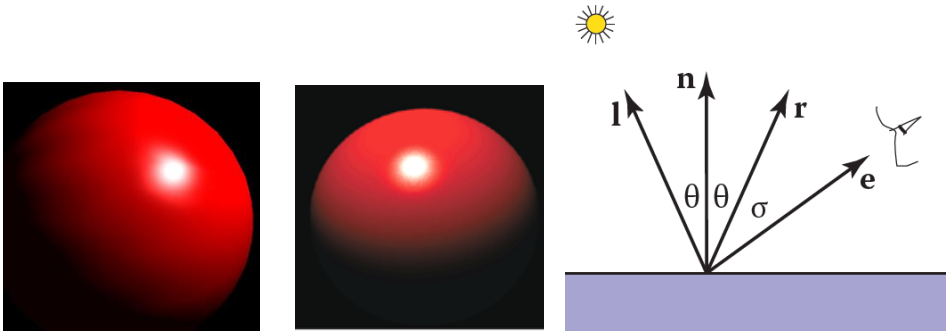


Figure 6.1: *Left and middle: Highlights on the surface. It moves across a surface as the viewpoint moves. Right: eye points matter in highlight.*

Some surfaces are essentially like matte surfaces, but they have highlights. This means that we must add a unit vector e toward the eye into our equations. This means that we must add a unit vector e toward the eye into our equations. We want to add a fuzzy "spot" the same color as the light source in the right place. The center of the spot should be drawn where e "lines" up with the natural direction of reflection r. We would like to have the highlight so that the eye sees some highlight wherever σ is small. c is bright when e = r and falls off gradually when e moves away from r. That gives us the following formula –

$$c = c_l \max(0, \mathbf{e} \cdot \mathbf{r})^p$$

Here,

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}$$

Therefore, our final formula for the Phong shading model is –

shade = ambient + diffuse + highlights

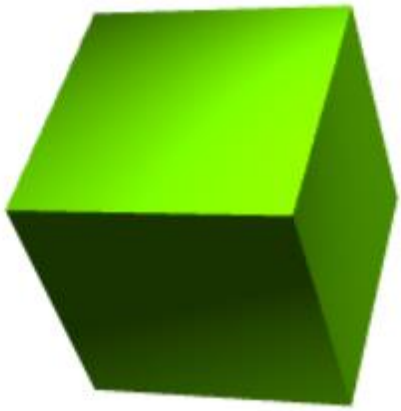
$$c = c_r \left(c_a + c_l \max\left(0, \mathbf{n} \cdot \mathbf{l}\right)\right) + c_l c_p \max(0, \mathbf{e} \cdot \mathbf{r})^p$$

Code: 6.1. Vertex and Fragment Shader source code for a shading model

```
var vertexShaderSource =
       `

        attribute vec3 a_coords;
        attribute vec3 a_colors;
        attribute vec3 a_normal;
        uniform mat4 u_T;
        uniform mat4 u_Rx;
        uniform mat4 u_Ry;
        uniform mat4 u_Basis;
        uniform mat4 u_Eye;
        uniform mat4 u_Pers;
```

```
        varying vec3 v_normal;
        varying vec3 v_coords;
        varying vec3 v_color;

        void main() {
            mat4 M = u_T*u_Ry*u_Rx;
            mat4 V = u_Basis*u_Eye;
            mat4 P = u_Pers;
            mat4 MVP = P*V*M;
            gl_Position = MVP*vec4(a_coords, 1.0);
            vec3 eyeNorm = mat3(V*M) * a_normal;
            v_normal = normalize(eyeNorm);
            v_color = a_colors;
            v_coords = a_coords;
        }`;

var fragmentShaderSource =

       `precision mediump float;
        uniform vec3 u_LightColor;
        uniform vec3 u_LightPosition;
        uniform vec3 u_AmbientLight;
        varying vec3 v_normal;
        varying vec3 v_coords;
        varying vec3 v_color;

        void main() {
            vec3 normal = normalize(v_normal);
            vec3 lightDirection = normalize(u_LightPosition - v_coords);
            float nDotL = max(dot(lightDirection, normal), 0.0);
            vec3 diffuse = u_LightColor * v_color * nDotL;
            vec3 ambient = u_AmbientLight * v_color;
            gl_FragColor = vec4(diffuse + ambient, 1.0);
        }`;
```

Output



6.5. Task:

Implement Toon shading as described in the text book

Midterm Examination

There will be an examination containing algorithm problems, simulations and multiple-choice questions.

Lab Final Examination

There will be an examination containing algorithm problems and simulations.

Project

Student will form groups and will develop project based on the contents taught in the lab sessions.

[END]