Ministry of Education and Science of Ukraine
National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”
Educational and Scientific Institute of Nuclear and Thermal Energy
Department of digital technologies in energy

Calculation and graphic work in the discipline “Visualization of graphic and geometric information”

Prepared by:
5th-year student
Group TR-41mp
Kondrashin O. M.

Kyiv - 2024

1 TASK DESCRIPTION

- Reuse texture mapping from Control task.
- Implement texture scaling (texture coordinates) scaling / rotation around user-specified point - odd variants implement scaling, even variants implement rotation
- It has to be possible to move the point along the surface (u,v) space using a keyboard. E.g. keys A and D move the point along u parameter and keys W and S move the point along v parameter.

2 THEORY DESCRIPTION

2.1 WebGL

WebGL (Web Graphics Library) is a JavaScript API designed for rendering interactive 2D and 3D graphics directly within a web browser, eliminating the need for additional plug-ins or software installations. It leverages the capabilities of OpenGL ES 2.0, making it highly versatile and compatible with a wide range of devices and platforms, including desktops, smartphones, and tablets. WebGL integrates seamlessly with other web technologies such as HTML5 and the DOM (Document Object Model) by operating directly within the browser environment. This integration allows developers to create dynamic and visually rich web applications, from simple animations to complex 3D visualizations and games. The cross-platform compatibility and native browser support ensure accessibility and performance consistency, making WebGL a powerful tool for modern web development.

2.2 Rendering Shapes in WebGL

Rendering shapes in WebGL is a fundamental process that involves defining the vertices and edges of geometric shapes within a 3D coordinate system. These shapes, also referred to as primitives, are the building blocks of 3D graphics and include points, lines, and triangles. Each shape is defined by a set of vertices, which are stored in Vertex Buffer Objects (VBOs). The VBOs allow efficient storage and transfer of vertex data to the GPU, enabling fast rendering. The appearance of these shapes is determined using vertex and fragment shaders, which are small programs written in GLSL (OpenGL Shading Language). Vertex shaders handle the position and transformations of vertices, while fragment shaders define the color, texture, and other visual properties of the shapes. This pipeline ensures flexibility and precision in creating visually complex and interactive scenes in WebGL applications.

2.3 Texturing

Texturing is a technique in WebGL that involves applying a 2D image onto a 3D object to enhance its visual appearance with realistic details. Texture mapping starts with texture coordinates, which define how the texture wraps around the shape. These coordinates are normalized values ranging from 0 to 1, corresponding to the proportion of the texture image applied to the object. The texture data, often derived from an image file, is accessed in the shaders through specialized variables called samplers. Samplers allow the GPU to read pixel information from the texture and apply it accurately to the rendered object. Texture Mapping is the process of associating vertex positions in 3D space with their corresponding texture coordinates, ensuring that the texture is aligned and displayed correctly. This method is widely used in 3D graphics to create lifelike surfaces, patterns, and effects.

2.4 Texture Center Movement and Rotation

Manipulating the position and orientation of a texture involves modifying the texture coordinates in the vertex shader. Moving a texture about its center requires adjusting the texture coordinates by adding or subtracting an offset value. This offset effectively shifts the texture in a specific direction, allowing for dynamic positioning on the surface of the 3D object. Rotating the texture about its center is a more complex operation that involves applying a 2D rotation matrix to the texture coordinates. The rotation is centered on the midpoint of the texture, ensuring smooth and visually coherent results. The rotation matrix incorporates trigonometric functions to calculate the new coordinates after rotation. These transformations enable advanced texture effects, such as creating rotating patterns or dynamically repositioning texture details in response to user interactions or animations.

3 IMPLEMENTATION DETAILS

3.1 Web page description, index.html

This HTML file implements an interactive 3D WebGL application that allows users to manipulate a shape with trackball-style mouse rotation. Two range sliders allow adjustment of the granularity or resolution of certain visual or animation effects. Sliders for adjusting the X, Y, and Z coordinates of the light source, affecting the shape's illumination. The page displays the center coordinates (U and V) of the shape's texture. A slider for setting the cube's rotation angle. A <canvas> element is used to render the 3D shape using WebGL. The file imports several external JavaScript and shader files for functionality, including trackball rotation, matrix operations, shaders, and the main application logic. JavaScript functions dynamically update UI elements and canvas when slider values are changed.

3.2 Main application logic, main.js

This JavaScript file implements the WebGL logic and interactivity for rendering a 3D surface with features like real-time lighting, texture handling, and keyboard control. Sets up the WebGL context and shader programs using initGL function and ShaderProgram constructor. Loads shaders and initializes transformation matrices for rendering. Updates the position of a light source dynamically based on user-controlled sliders for X, Y, and Z coordinates through updateLightPosition function. The animate function initializes WebGL and calls the draw function to render the scene. The draw function applies transformations, updates shader variables, and renders the surface using buffer data. Interactability is handled by handleKeyDown function, which adjusts texture center coordinates (U and V) using W, A, S, D keys, while TrackballRotator allows mouse-based rotation of the 3D view. The surface object manages 3D model data, including vertices, indices, normals, and textures.

3.3 Surface model class Model.js

This JavaScript file implements a 3D model data calculation and storage. A deg2rad function to convert degrees to radians for geometric calculations. Vertex and Triangle classes represent the basic building blocks of a 3D model, including position, normals, and connectivity. The Model class manages buffers for vertices, indices, normals, texture coordinates, and tangents. The class includes the following methods:

- BufferData uploads vertex and index data to GPU buffers.
- Draw renders the model using WebGL's gl.drawElements.
- loadTexture loads and assigns diffuse, normal, and specular textures.
- bindTextures binds textures to the shader program.

Functions like calculateNormals compute surface normals for lighting calculations. The CreateSurfaceData function generates vertices and triangles for the surface. The function adjusts the geometry based on user input.

The class utilizes WebGL to handle buffers and textures for real-time 3D rendering. The file combines procedural geometry creation, texture mapping, and lighting support to form a 3D rendering pipeline.

4 USER'S INSTRUCTION

4.1 Setup

Open the project in a modern web browser that supports WebGL using the Live server Visual Studio Code extension (Fig. 4.1).

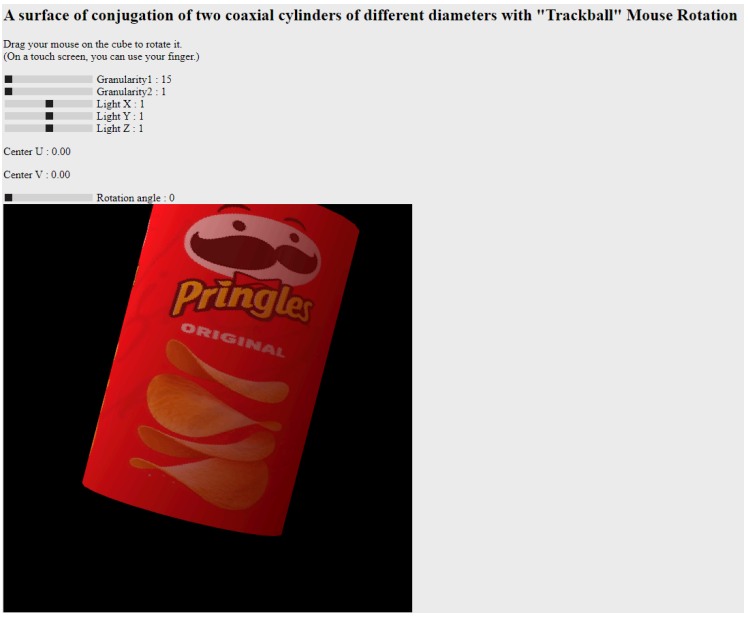

Figure 4.1. Homepage.

4.2 Adjusting model granularity

Use sliders Granularity1 and Granularity2 to adjust surface granularity over U and V directions (Fig. 4.2).

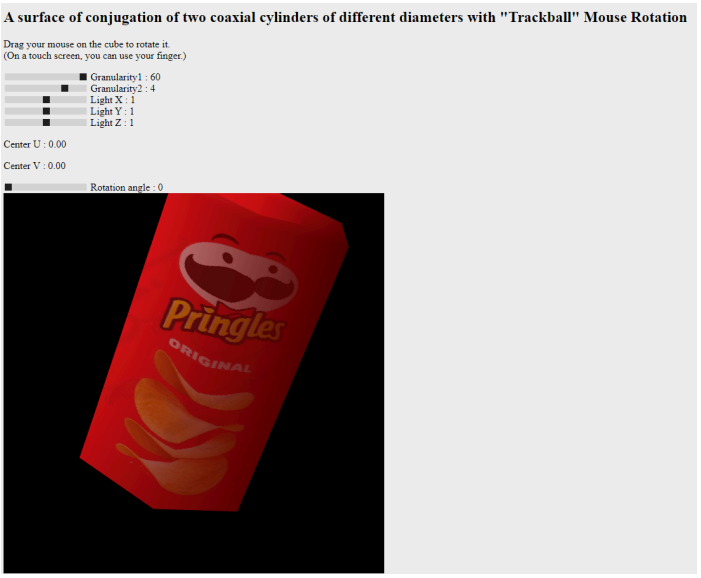

Figure 4.2. Granularity adjustment demonstration.

4.3 Adjusting model and light position.

Use sliders Light X, Y, and Z to adjust the light position and drag your mouse on the model to rotate it (Fig. 4.3).

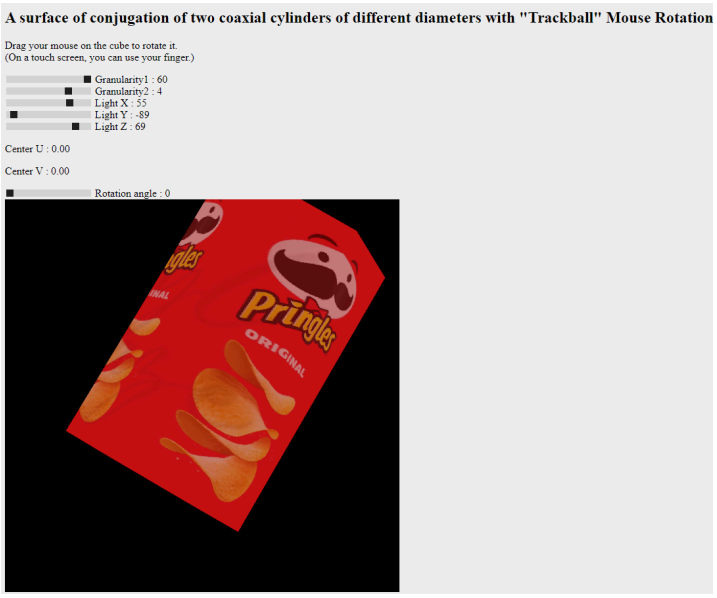

Figure 4.3. Position adjustment demonstration.

4.4 Adjusting texture position and rotation.

Use W, A, S, D keys to adjust texture position and the Rotation angle slider to adjust texture rotation (Fig 4.4).

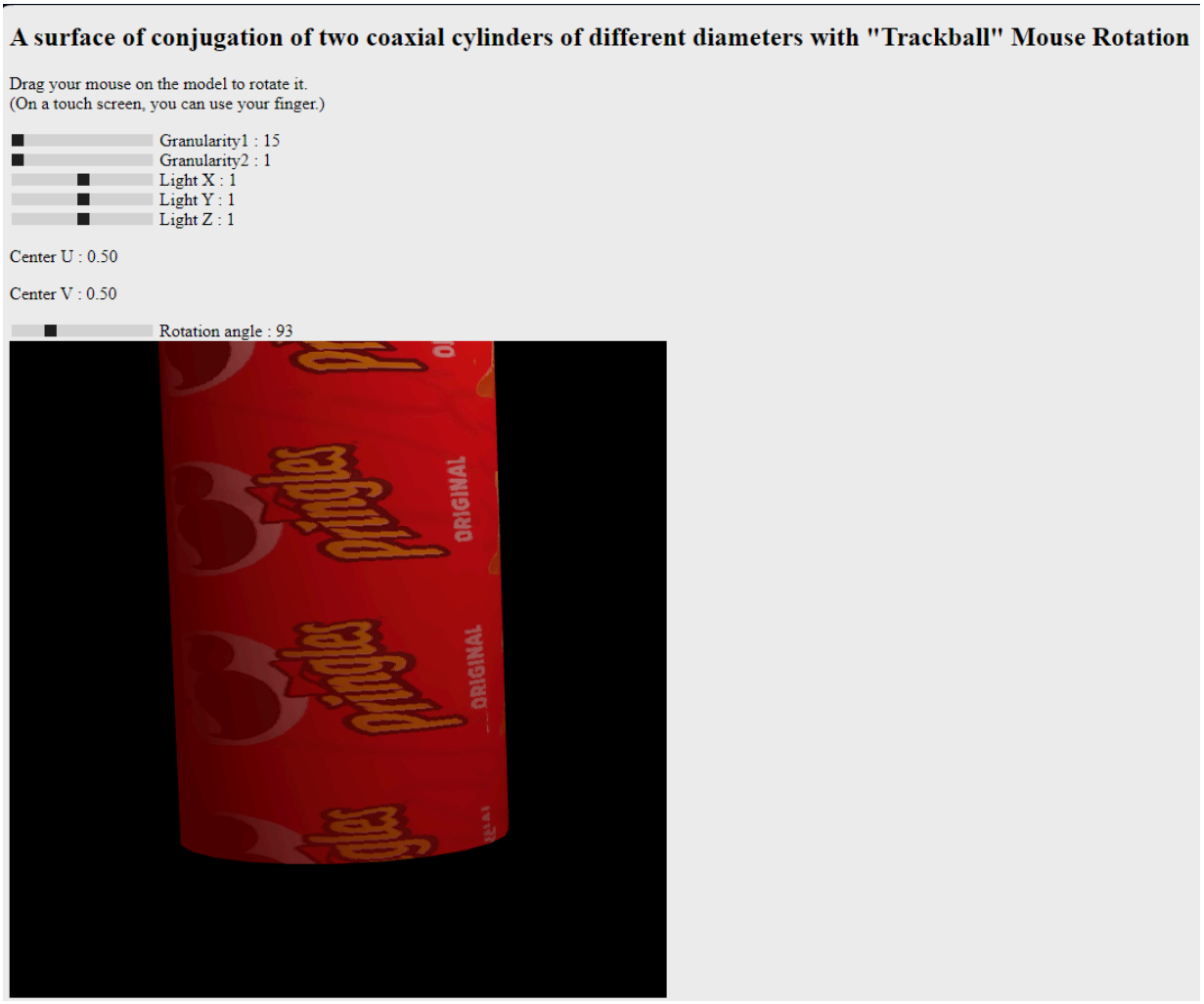

Figure 4.4. Texture adjustment demonstration.

5 SOURCE CODE SAMPLE

5.1 Web page sample

```
<!DOCTYPE html>
<html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8">


<title>CGW</title>
<style>
    body {
        background-color: #EEEEEE;
    }
</style>

<script src="./Utils/trackball-rotator.js"></script>
<script src="./Utils/m4.js"></script>
<script src="./shader.gpu"></script>
<script src="./Model.js"></script>
<script src="./main.js"></script>

</head>
<body onload="init()">

<h2>A surface of conjugation of two coaxial cylinders of different diameters with
"Trackball" Mouse Rotation</h2>

<p id="message">Drag your mouse on the model to rotate it.<br>
(On a touch screen, you can use your finger.)</p>
<div>
     <input type="range" id="stepRange" name="stepRange" min="15" max="60" value="15"
step="15" oninput="updateRangeValue(this.value)"/>
    <label for="stepRange">Granularity1 : <span id="rangeValue">15</span></label>


<script>
    function updateRangeValue(value) {
        document.getElementById("rangeValue").textContent = value;
        animate();
    }
</script>
  </div>
  <div>
     <input type="range" id="step2Range" name="step2Range" min="1" max="5" value="1"
oninput="updateRange2Value(this.value)"/>
    <label for="step2Range">Granularity2 : <span id="range2Value">1</span></label>


<script>
    function updateRange2Value(value) {
        document.getElementById("range2Value").textContent = value;
        animate();

    }
</script>
  </div>
```

```
<!-- // light x -->
  <div>
      <input type="range" id="lxRange" name="lxRange" min="-100" max="100" value="1"
oninput="updatelxRange(this.value)"/>
    <label for="lxRange">Light X : <span id="lxrangeValue">1</span></label>


<script>
    function updatelxRange(value) {
        document.getElementById("lxrangeValue").textContent = value;
        animate();
    }
</script>
  </div>
  <!-- // light y -->
  <div>
      <input type="range" id="lyRange" name="lyRange" min="-100" max="100" value="1"
oninput="updatelyRange(this.value)"/>
    <label for="lyRange">Light Y : <span id="lyrangeValue">1</span></label>


<script>
    function updatelyRange(value) {
        document.getElementById("lyrangeValue").textContent = value;
        animate();
    }
</script>
  </div>
  <!-- // light z -->
  <div>
      <input type="range" id="lzRange" name="lzRange" min="-100" max="100" value="1"
oninput="updatelzRange(this.value)"/>
    <label for="lzRange">Light Z : <span id="lzrangeValue">1</span></label>


<script>
    function updatelzRange(value) {
        document.getElementById("lzrangeValue").textContent = value;
        animate();
    }
</script>
  </div>

<!-- // texture center  -->
<div>

  <p>Center U : <span id="CUpValue">0.00</span></p>
  <p>Center V : <span id="CVpValue">0.00</span></p>
  </div>

<!-- // rotation angle -->
<div>
       <input type="range" id="RARange" name="RARange" min="0" max="360" value="0"
oninput="updateRARange(this.value)"/>
    <label for="RARange">Rotation angle : <span id="RArangeValue">0</span></label>


  <script>
    function updateRARange(value) {
        document.getElementById("RArangeValue").textContent = value;
        animate();
    }
  </script>
</div>
```

```
<div id="canvas-holder">
                    <canvas       width="600"       height="600"       id="webglcanvas"
style="background-color:red"></canvas>
</div>


</body></html>
```

5.2 Main logic file sample

```
'use strict';

let gl;                         // The webgl context.
let surface;                    // A surface model
let shProgram;                  // A shader program
let spaceball;                  // A SimpleRotator object that lets the user rotate the
view by mouse.

let lightAngle = 0;
let lightRadius = 180.0;

function updateLightPosition() {
    lightAngle += 0.01;

    let lightX = +document.getElementById("lxRange").value
    let lightY = +document.getElementById("lyRange").value
    let lightZ = +document.getElementById("lzRange").value

    gl.uniform3fv(shProgram.iLightSource, [lightX, lightY, lightZ]);
}

function animate() {
    initGL();  // initialize the WebGL graphics context
    draw();
}

// Constructor
function ShaderProgram(name, program) {

    this.name = name;
    this.prog = program;

    // Location of the attribute variable in the shader program.
    this.iAttribVertex = -1;
    // Location of the uniform specifying a color for the primitive.
    this.iColor = -1;
    // Location of the uniform matrix representing the combined transformation.
    this.iModelViewProjectionMatrix = -1;
    this.iNormalMatrix = -1;

    // Location of the uniform matrix representing the modelview transformation
    this.iModelViewMatrix = -1;

    this.Use = function() {
        gl.useProgram(this.prog);
    }
}
```

```
function handleKeyDown(event) {
      switch (event.key) {
            case "a":
            case "A":
            if(+document.getElementById("CUpValue").textContent >= 0.1){
                                    document.getElementById("CUpValue").textContent  =
(+document.getElementById("CUpValue").textContent - 0.1).toFixed(2);
                animate()
            }
                  break;
            case "d":
            case "D":
            if(+document.getElementById("CUpValue").textContent <= 0.9){
                                    document.getElementById("CUpValue").textContent  =
(+document.getElementById("CUpValue").textContent + 0.1).toFixed(2);
                animate()
            }
                  break;
            case "w":
            case "W":
            if(+document.getElementById("CVpValue").textContent <= 0.9){
                                    document.getElementById("CVpValue").textContent  =
(+document.getElementById("CVpValue").textContent + 0.1).toFixed(2);
                animate()
            }
                  break;
            case "s":
            case "S":
            if(+document.getElementById("CVpValue").textContent >= 0.1){
                                    document.getElementById("CVpValue").textContent  =
(+document.getElementById("CVpValue").textContent - 0.1).toFixed(2);
                animate()
            }
                  break;
      }
}

function draw() {
    gl.clearColor(0,0,0,1);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    /* Set the values of the projection transformation */
    let projection = m4.perspective(Math.PI/8, 1, 8, 12);

    /* Get the view matrix from the SimpleRotator object.*/
    let modelView = spaceball.getViewMatrix();

    let rotateToPointZero = m4.axisRotation([0.707,0.707,0], 0.7);
    let translateToPointZero = m4.translation(0,0,-10);

    let matAccum0 = m4.multiply(rotateToPointZero, modelView );
    let matAccum1 = m4.multiply(translateToPointZero, matAccum0 );

    /* Multiply the projection matrix times the modelview matrix to give the
       combined transformation matrix, and send that to the shader program. */
    let modelViewProjection = m4.multiply(projection, matAccum1 );
    let normalMatrix = m4.transpose(m4.inverse(matAccum1));

                    gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix,     false,
modelViewProjection );
    gl.uniformMatrix4fv(shProgram.iNormalMatrix, false, normalMatrix );
    gl.uniformMatrix4fv(shProgram.iModelViewMatrix, false, matAccum1 );
    gl.uniform1i(shProgram.iTMU0, 0);
```

```
    gl.uniform1i(shProgram.iTMU1, 1);

    updateLightPosition();
    /* Draw the six faces of a cube, with different colors. */
    gl.uniform4fv(shProgram.iColor, [1,1,0,1] );

    surface.bindTextures();
    surface.Draw();
}



/* Initialize the WebGL context. Called from init() */
function initGL() {
    let prog = createProgram( gl, vertexShaderSource, fragmentShaderSource );



    shProgram = new ShaderProgram('Basic', prog);
    shProgram.Use();

    shProgram.iAttribVertex              = gl.getAttribLocation(prog, "vertex");
                shProgram.iModelViewProjectionMatrix    =    gl.getUniformLocation(prog,
"ModelViewProjectionMatrix");
    shProgram.iColor                     = gl.getUniformLocation(prog, "color");
    shProgram.iAttribNormal              = gl.getAttribLocation(prog, "normal");
    shProgram.iLightSource               = gl.getUniformLocation(prog, "lightPos");
    shProgram.iNormalMatrix = gl.getUniformLocation(prog, "normalMatrix");

         shProgram.iModelViewMatrix                      =  gl.getUniformLocation(prog,
"ModelViewMatrix");
        shProgram.textureNormal                          =  gl.getUniformLocation(prog,
"textureNormal");
         shProgram.textureDiffuse                        =  gl.getUniformLocation(prog,
"textureDiffuse");
         shProgram.textureSpecular                       =  gl.getUniformLocation(prog,
"textureSpecular");
    shProgram.iAttribTexCoords           = gl.getAttribLocation(prog, "texCoord");
    shProgram.iAttribTangent             = gl.getAttribLocation(prog, "tangent");


    let data = {};

    CreateSurfaceData(data)

    surface = new Model('Surface');
              surface.BufferData(data.verticesF32,   data.indicesU16,   data.normalsF32,
data.texcoordsF32, data.tangentsF32);
    surface.loadTexture();

    gl.enable(gl.DEPTH_TEST);
}


function createProgram(gl, vShader, fShader) {
    let vsh = gl.createShader( gl.VERTEX_SHADER );
    gl.shaderSource(vsh,vShader);
    gl.compileShader(vsh);
    if ( ! gl.getShaderParameter(vsh, gl.COMPILE_STATUS) ) {
        throw new Error("Error in vertex shader:  " + gl.getShaderInfoLog(vsh));
     }
    let fsh = gl.createShader( gl.FRAGMENT_SHADER );
    gl.shaderSource(fsh, fShader);
    gl.compileShader(fsh);
```

```
    if ( ! gl.getShaderParameter(fsh, gl.COMPILE_STATUS) ) {
       throw new Error("Error in fragment shader:  " + gl.getShaderInfoLog(fsh));
    }
    let prog = gl.createProgram();
    gl.attachShader(prog,vsh);
    gl.attachShader(prog, fsh);
    gl.linkProgram(prog);
    if ( ! gl.getProgramParameter( prog, gl.LINK_STATUS) ) {
       throw new Error("Link error in program:  " + gl.getProgramInfoLog(prog));
    }
    return prog;
}


/**
 * initialization function that will be called when the page has loaded
 */
function init() {
    let canvas;
    try {
        canvas = document.getElementById("webglcanvas");
        gl = canvas.getContext("webgl");
        if ( ! gl ) {
            throw "Browser does not support WebGL";
        }
    }
    catch (e) {
        document.getElementById("canvas-holder").innerHTML =
            "<p>Sorry, could not get a WebGL graphics context.</p>";
        return;
    }
    try {
        initGL();  // initialize the WebGL graphics context
    }
    catch (e) {
        document.getElementById("canvas-holder").innerHTML =
            "<p>Sorry, could not initialize the WebGL graphics context: " + e + "</p>";
        return;
    }

    spaceball = new TrackballRotator(canvas, draw, 0);

    //draw();
    animate();
    window.addEventListener("keydown", handleKeyDown);
}
```

5.3 Model class sample

```
function deg2rad(angle) {
    return angle * Math.PI / 180;
}


function Vertex(p, t, a)
{
    this.p = p;
    this.normal = [];
    this.triangles = [];
    this.t = t
```

```
    this.a = a
}

function Triangle(v0, v1, v2)
{
    this.v0 = v0;
    this.v1 = v1;
    this.v2 = v2;
    this.normal = [];
    this.tangent = [];
}

// Constructor
function Model(name) {
    this.name = name;
    this.iVertexBuffer = gl.createBuffer();
    this.iIndexBuffer = gl.createBuffer();
    this.iNormalBuffer = gl.createBuffer();
    this.iTexCoordsBuffer = gl.createBuffer();
    this.iTangentBuffer = gl.createBuffer();

    this.textureDiffuse = gl.createTexture();
    this.textureNormal = gl.createTexture();
    this.textureSpecular = gl.createTexture();

    this.count = 0;

    this.BufferData = function(vertices, indices, normals, texCoords, tangents) {

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STREAM_DRAW);
        gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribVertex);

        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.iIndexBuffer);
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STREAM_DRAW);

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iNormalBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, normals, gl.STREAM_DRAW);
        gl.vertexAttribPointer(shProgram.iAttribNormal, 3, gl.FLOAT, true, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribNormal);

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iTexCoordsBuffer);
        gl.bufferData(gl.ARRAY_BUFFER,texCoords, gl.STREAM_DRAW);
        gl.vertexAttribPointer(shProgram.iAttribTexCoords, 2, gl.FLOAT, false, 0, 0);
            gl.enableVertexAttribArray(shProgram.iAttribTexCoords);

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iTangentBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, tangents, gl.STREAM_DRAW);
        gl.vertexAttribPointer(shProgram.iAttribTangent, 3, gl.FLOAT, false, 0, 0);
            gl.enableVertexAttribArray(shProgram.iAttribTangent);


        this.count = indices.length;
    }

    this.Draw = function() {
        gl.drawElements(gl.TRIANGLES, this.count, gl.UNSIGNED_SHORT, 0);
    }

    this.loadTexture = function(){
        //Diffuse
        gl.bindTexture(gl.TEXTURE_2D, this.textureDiffuse);
        gl.texImage2D(
```

```
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    1,
    1,
    0,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    new Uint8Array([100, 0, 0, 200]),
);

const image1 = new Image();
image1.onload = () => {
    gl.bindTexture(gl.TEXTURE_2D, this.textureDiffuse);
    gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        image1,
    );
    gl.generateMipmap(gl.TEXTURE_2D);
};
image1.src = "Texture/Pringles_BaseColor.jpg";

// Normal
gl.bindTexture(gl.TEXTURE_2D, this.textureNormal);
gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    1,
    1,
    0,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    new Uint8Array([100, 0, 0, 200]),
);

const image2 = new Image();
image2.onload = () => {
    gl.bindTexture(gl.TEXTURE_2D, this.textureSpecular);
    gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
        image2,
    );
    gl.generateMipmap(gl.TEXTURE_2D);
};
image2.src = "Texture/Pringles_Roughness.jpg";

const image3 = new Image();
image3.onload = () => {
    gl.bindTexture(gl.TEXTURE_2D, this.textureNormal);
    gl.texImage2D(
        gl.TEXTURE_2D,
        0,
        gl.RGBA,
        gl.RGBA,
        gl.UNSIGNED_BYTE,
```

```
                image3,
            );
            gl.generateMipmap(gl.TEXTURE_2D);
        };
        image3.src = "Texture/Pringles_Normal.png";

        // Specular
        gl.bindTexture(gl.TEXTURE_2D, this.textureSpecular);
        gl.texImage2D(
            gl.TEXTURE_2D,
            0,
            gl.RGBA,
            1,
            1,
            0,
            gl.RGBA,
            gl.UNSIGNED_BYTE,
            new Uint8Array([100, 0, 0, 200]),
        );


    }

    this.bindTextures = function(){
        gl.activeTexture(gl.TEXTURE0);
        gl.bindTexture(gl.TEXTURE_2D, this.textureDiffuse);
        gl.uniform1i(shProgram.diffuseTextureUni, 0);

        gl.activeTexture(gl.TEXTURE1);
        gl.bindTexture(gl.TEXTURE_2D, this.textureSpecular);
        gl.uniform1i(shProgram.specularTextureUni, 1);

        gl.activeTexture(gl.TEXTURE2);
        gl.bindTexture(gl.TEXTURE_2D, this.textureNormal);
        gl.uniform1i(shProgram.normalTextureUni, 2);
     }
}

function calculateNormal(v0, v1, v2){
    let t1 = [v1[0] - v0[0], v1[1] - v0[1], v1[2] - v0[2]]
    let t2 = [v2[0] - v0[0], v2[1] - v0[1], v2[2] - v0[2]]
    let n0 = t1[1] * t2[2] - t1[2] * t2[1]
    let n1 = -1 * (t1[0] * t2[2] - t1[2] * t2[0])
    let n2 = t1[0] * t2[1] - t1[1] * t2[0]
    //let n = [n0, n1, n2]
    let nlen = Math.sqrt(n0 * n0 + n1 * n1 + n2 * n2)
    let n = [n0 / nlen, n1 / nlen, n2 / nlen]
    return [...n]
}

function calculateNormals(indices, vertices, triangles){

    // for(let i = 0; i < indices.length; i+=3){
    //     let norm = calculateNormal(vertices[indices[i]].p, vertices[indices[i+1]].p,
vertices[indices[i+2]].p)
    //     vertices[indices[i]].normal = norm
    //     vertices[indices[i+1]].normal = norm
    //     vertices[indices[i+2]].normal = norm

    // }
    for(let i = 0; i < indices.length; i++){
              vertices[indices[i]].normal  =  [Math.cos(deg2rad(vertices[indices[i]].a)),
Math.sin(deg2rad(vertices[indices[i]].a)), 0]
```

```
    }

}

function CreateSurfaceData(data)
{
    let vertices = [];
    let triangles = [];
    let r1 = 0.5
    let r2 = 1
    let hMax = 3
    let zoffset =-2

    let step = +document.getElementById("stepRange").value
    let iterCount = Math.ceil(360/step)+1

    let heightIterCount = +document.getElementById("step2Range").value
    let heightStep = hMax / heightIterCount
    let centerU = +document.getElementById("CUpValue").textContent
    let centerV = +document.getElementById("CVpValue").textContent
    let angle = deg2rad(+document.getElementById("RARange").value)
    let cosAngle = Math.cos(angle);
    let sinAngle = Math.sin(angle);

    for (let i=0, ang = 0; i<iterCount; i++, ang+=step) {
            vertices.push( new Vertex( [r1 * Math.sin(deg2rad(ang)), zoffset+0, r1 *
Math.cos(deg2rad(ang))], [ang/360, 0], angle ));
            vertices.push( new Vertex( [r2 * Math.sin(deg2rad(ang)), zoffset+0, r2 *
Math.cos(deg2rad(ang))], [-ang/360, 0], angle ));
        // FLOOR
        let v0ind = vertices.length-1;
        if (i > 0)
            {

                let v1ind = v0ind - 2;
                let v2ind = v0ind - 1
                let v3ind = v0ind - 3

                let trian = new Triangle(v0ind, v3ind, v2ind);
                let trianInd = triangles.length;

                triangles.push( trian );
                vertices[v0ind].triangles.push(trianInd);
                vertices[v1ind].triangles.push(trianInd);
                vertices[v3ind].triangles.push(trianInd);

                let trian2 = new Triangle(v3ind, v0ind, v1ind);
                let trianInd2 = triangles.length;

                triangles.push( trian2 );
                vertices[v0ind].triangles.push(trianInd2);
                vertices[v3ind].triangles.push(trianInd2);
                vertices[v1ind].triangles.push(trianInd2);

            }
    }
    for(let j = 0, h = heightStep; j < heightIterCount; j++, h+=heightStep){
    for (let i=0, ang = 0; i<iterCount; i++, ang+=step) {

        let v0ind = vertices.length -1;


            vertices.push( new Vertex( [r1 * Math.sin(deg2rad(ang)), zoffset+h, r1 *
Math.cos(deg2rad(ang))], [ang/360, h/hMax], angle ));
```

```
            vertices.push( new Vertex( [r2 * Math.sin(deg2rad(ang)), zoffset+h, r2 *
Math.cos(deg2rad(ang))], [-ang/360, h/hMax], angle ));

         // v0    v2
         //   o - o
         //   | \ |
         //   o - o
         // v3     v1
        // CEILING
        v0ind+=2
        if (i > 0 && h == hMax)
            {

                let v1ind = v0ind - 2;
                let v2ind = v0ind - 1
                let v3ind = v0ind - 3

                let trian = new Triangle(v0ind, v3ind, v2ind);
                let trianInd = triangles.length;

                triangles.push( trian );
                vertices[v0ind].triangles.push(trianInd);
                vertices[v1ind].triangles.push(trianInd);
                vertices[v3ind].triangles.push(trianInd);

                let trian2 = new Triangle(v3ind, v0ind, v1ind);
                let trianInd2 = triangles.length;

                triangles.push( trian2 );
                vertices[v0ind].triangles.push(trianInd2);
                vertices[v3ind].triangles.push(trianInd2);
                vertices[v1ind].triangles.push(trianInd2);

            }
        //OUTER WALLS
        v0ind-=2
        if (i > 0)
        {
            let v1ind = v0ind - iterCount*2;
            let v2ind = v0ind + 2
            let v3ind = v0ind - iterCount*2 +2
            let trian = new Triangle(v0ind, v3ind, v2ind);
            let trianInd = triangles.length;

            triangles.push( trian );
            vertices[v0ind].triangles.push(trianInd);
            vertices[v1ind].triangles.push(trianInd);
            vertices[v3ind].triangles.push(trianInd);

            let trian2 = new Triangle(v3ind, v0ind, v1ind);
            let trianInd2 = triangles.length;

            triangles.push( trian2 );
            vertices[v0ind].triangles.push(trianInd2);
            vertices[v3ind].triangles.push(trianInd2);
            vertices[v1ind].triangles.push(trianInd2);

        }
        // INNER WALLS
        v0ind--
        if (i > 0)
            {

                let v1ind = v0ind - iterCount*2;
```

```
                let v2ind = v0ind + 2
                let v3ind = v0ind - iterCount*2 +2

                let trian = new Triangle(v0ind, v3ind, v2ind);
                let trianInd = triangles.length;

                triangles.push( trian );
                vertices[v0ind].triangles.push(trianInd);
                vertices[v1ind].triangles.push(trianInd);
                vertices[v3ind].triangles.push(trianInd);

                let trian2 = new Triangle(v3ind, v0ind, v1ind);
                let trianInd2 = triangles.length;

                triangles.push( trian2 );
                vertices[v0ind].triangles.push(trianInd2);
                vertices[v3ind].triangles.push(trianInd2);
                vertices[v1ind].triangles.push(trianInd2);

            }

    }
    }
    data.verticesF32 = new Float32Array(vertices.length*3);
    data.texcoordsF32 = new Float32Array(vertices.length*2);
    for (let i=0, len=vertices.length; i<len; i++)
    {
        data.verticesF32[i*3 + 0] = vertices[i].p[0];
        data.verticesF32[i*3 + 1] = vertices[i].p[1];
        data.verticesF32[i*3 + 2] = vertices[i].p[2];

        let u = vertices[i].t[0];
        let v = vertices[i].t[1];
        data.texcoordsF32[i*2 + 0] = u * cosAngle - v * sinAngle + centerU;
        data.texcoordsF32[i*2 + 1] = u * sinAngle + v * cosAngle + centerV;
    }

    data.indicesU16 = new Uint16Array(triangles.length*3);
    for (let i=0, len=triangles.length; i<len; i++)
    {
        data.indicesU16[i*3 + 0] = triangles[i].v0;
        data.indicesU16[i*3 + 1] = triangles[i].v1;
        data.indicesU16[i*3 + 2] = triangles[i].v2;
    }
    calculateNormals(data.indicesU16, vertices, triangles)
    data.normalsF32 = new Float32Array(vertices.length*3);
    for (let i=0, len=vertices.length; i<len; i++)
        {
            data.normalsF32[i*3 + 0] = vertices[i].normal[0];
            data.normalsF32[i*3 + 1] = vertices[i].normal[1];
            data.normalsF32[i*3 + 2] = vertices[i].normal[2];
    }
    data.tangentsF32 = new Float32Array(vertices.length*3);
    for (let i=0, len=vertices.length; i<len; i++)
        {
            data.tangentsF32[i*3 + 0] = 1;
            data.tangentsF32[i*3 + 1] = 0;
            data.tangentsF32[i*3 + 2] = 0;
    }


}
```