МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ
КАФЕДРА ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Варіант - 3

Розрахунково-графічна робота

з дисципліни «Візуалізація графічної та геометричної інформації»

Виконав:
Студент 5-го курсу
ІАТЕ
групи ТР-32мп
Білик Максим Олександрович

Перевірив:
Демчишин Анатолій Анатолійович

Київ - 2023

ЗАВДАННЯ

1. Нанести текстуру на поверхню з практичного завдання №2;

2. Реалізувати обертання текстури (координат текстури) навколо вказаної користувачем точки;

3. Реалізувати можливість переміщення точки вздовж простору поверхні (u,v) за допомогою клавіатури. Клавіші A і D переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.

ТЕОРЕТИЧНІ ОСНОВИ

WebGL (Web Graphics Library) - це технологія, яка дозволяє створювати тривимірну графіку прямо у веб-браузері, не вимагаючи встановлення додаткових плагінів або розширень. Вона ґрунтується на OpenGL ES (Embedded Systems) і має ряд розширень для взаємодії з елементами веб-сторінок. WebGL дозволяє розробникам створювати вражаючі та взаємодійські веб-додатки, ігри та візуалізації, використовуючи високоякісну тривимірну графіку.

У контексті WebGL, текстура - це зображення, яке можна призначити тривимірній моделі для надання додаткової деталізації, кольору, рельєфу або інших візуальних характеристик. Текстури використовуються для імітації реальних матеріалів, створення візуальних ефектів та підвищення реалізму сцен. Головна мета використання текстур полягає в тому, щоб забезпечити тривимірним об'єктам деталізований вигляд, не збільшуючи кількість полігонів у моделі. Це допомагає оптимізувати продуктивність, забезпечуючи при цьому високу якість візуалізації. Для створення текстур в WebGL використовується функція createTexture(). Після цього ініціалізується об'єкт Image, в якому вказується URL-адреса зображення, яке буде використовуватися як текстура. Кожен вершинний піксель на тривимірній моделі має відповідні координати текстури, які визначають, який фрагмент текстури буде використовуватися для цього пікселя. Для оптимізації використання текстур можна застосовувати різні методи фільтрації, такі як білінійна або трилінійна фільтрація.

Існують різні види текстур, такі як 2D-текстури для загальних завдань, текстури висот (heightmaps) для створення рельєфу та куб-текстури для створення оточення чи інших візуальних ефектів, які вимагають 360-градусних зображень.

Загалом, текстури є важливою складовою графічного програмування та візуалізації, дозволяючи створювати реалістичні сцени та оптимізувати продуктивність без втрати якості візуалізації. Обертання в графіці включає процес зміни орієнтації об'єкта чи сцени навколо конкретної осі чи точки. Це є фундаментальною операцією у 3D-графіці, яка дозволяє створювати візуальні ефекти, змінювати перспективу та взаємодію об'єктів на сцені. Для реалізації обертання використовуються математичні формули, зокрема матриці обертання, які описують трансформації координат об'єктів у просторі.

ПРОГРАМНА РЕАЛІЗАЦІЯ

Текстурні координати зберігаються в створеному для цього буфері:

```
gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextCoordBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textCoords), gl.STREAM_DRAW);
```

Збережені координати текстур використовуються під час відображення сцени:

```
gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextCoordBuffer);
gl.vertexAttribPointer(shProgram.iAttribTextCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(shProgram.iAttribTextCoord);
```

Для встановлення відповідності між текстурними координатами та вершинами поверхні застосовуються параметри u та v. Для досягнення цієї відповідності необхідно нормалізувати значення цих параметрів до проміжку від 0 до 1.

```
function CalculateTextCoord(u, v) {
   u = (u - 0.25)/(maxR - 0.25);
   v = v / 2*Math.PI;
   return {u, v};
}
```

Функція викликається для кожної вершини об’єкту, яку ми знаходимо в функції CreateSurfaceData(), і отримані відповідні текстурні координати заносяться в

масив textCoordList. Цей масив заповнює буфер координат текстур. Текстура завантажується в функції LoadTexture(). Для цього використовується об’єкт класу Image, в якому вказується URL-посилання на віддалений ресурс, де зберігається потрібне зображення. Після вдалого завантаження, зображення прив’язується до створеного в цій же функції об’єкту текстури. Окрім цього, вказуються параметри фільтрації текстури. В кінці викликається функція рендерингу сцени:

```
function LoadTexture() {
   var texture = gl.createTexture();
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
   var image = new Image();
   image.crossOrigin = "anonymous";
   image.src = "https://i.ibb.co/b5xQL8G/texture5.jpg";
   image.addEventListener('load', () => {
      gl.bindTexture(gl.TEXTURE_2D, texture);
      gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
      draw();
   }
   );
}
```

Модифікував вершинний шейдер для обертання текстури, додавши в нього дві функції, що відповідають за створення матриці обертання та зсуву:

```
mat4 rotate(float angleInRadians) {
   float c = cos(angleInRadians);
   float s = sin(angleInRadians);
   return mat4(
       c,   s, 0.0, 0.0,
      -s,   c, 0.0, 0.0,
      0.0, 0.0, 1.0, 0.0,
      0.0, 0.0, 0.0, 1.0
   );
}
mat4 translate(float tx, float ty) {
   return 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,
        tx,  ty,  0.0, 1.0
    );
}
```

Обертання текстурної координати реалізовано наступним чином:

1. Знаходимо матриці обертання та зсуву

```
mat4 rotateMat = rotate(angleInRadians);
mat4 translateMat = translate(-userPoint.x, -userPoint.y);
mat4 translateMatBack = translate(userPoint.x, userPoint.y);
```

2. Переносимо текстурну координату до початку координат

```
vec4 textCoordTr = translateMat*vec4(textCoord,0,1.0);
```

3. Обертаємо координату

```
vec4 textCoordRotated = rotateMat*textCoordTr;
```

4. Повертаємо координату назад та передаємо отримане значення в фрагментний шейдер

```
vec4 textCoordTrBack = translateMatBack*textCoordRotated; textInterp = textCoordTrBack.xy;
```

Є функція handleKeyPress(), яка реалізує можливість переміщення точки на поверхні, навколо якої обертається текстура, за допомогою клавіш WASD. W та S переміщують точку по параметру u, A та D – по параметру v:

```
function handleKeyPress(event) {
    let stepSize = 0.05;
    switch (event.key) {
        case 'w':
        case 'W':
            userPoint[0] += stepSize;
            if (userPoint[0] > maxR)
            {
                userPoint[0] = 0.25;
            }
            break;
        case 's':
        case 'S':
            userPoint[0] -= stepSize;
            if (userPoint[0] < 0.25)
            {
```

```
                userPoint[0] = maxR;
            }
            break;
        case 'a':
        case 'A':
            userPoint[1] -= stepSize;
            if (userPoint[1] < 0)
            {
                userPoint[1] = 2 * Math.PI;;
            }
            break;
        case 'd':
        case 'D':
            userPoint[1] += stepSize;
            if (userPoint[1] > 2 * Math.PI)
            {
                userPoint[1] = 0;
            }
            break;
        default:
            return;
    }
    updateSurface();
}
```

ВЗАЄМОДІЯ З ПРОГРАМОЮ

Поверхня з нанесеною на неї текстурою:

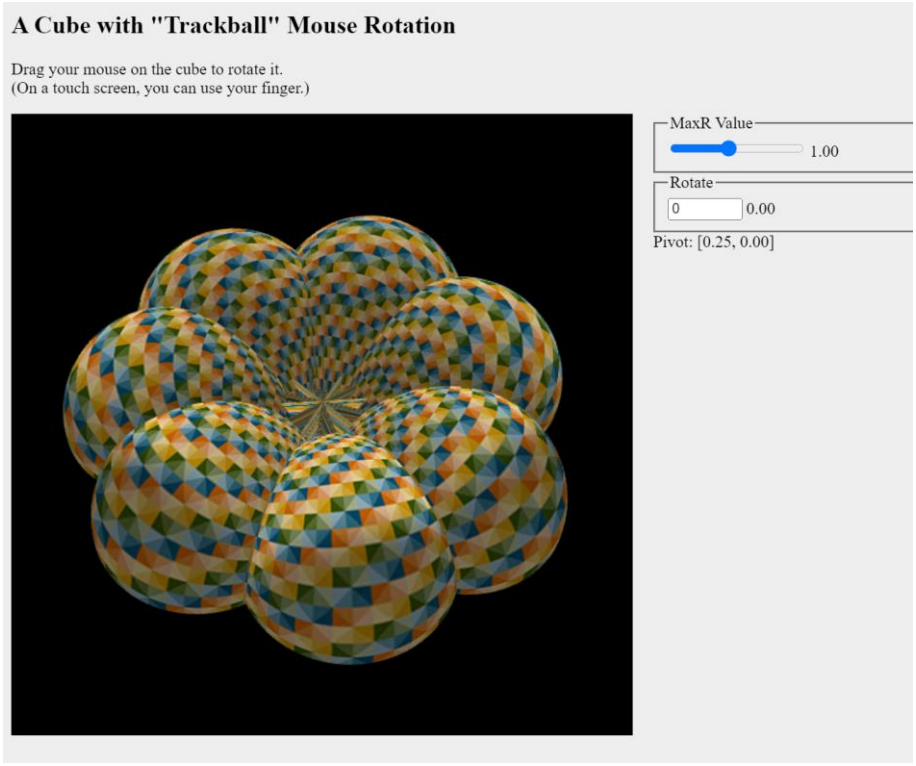

Рисунок 1. – Поверхня з параметрами за замовчуванням

Вказавши в полі Rotate значення, ми можемо обертати текстуру на відповідний кут. На рисунку 2 текстура обернена на 45 градусів:

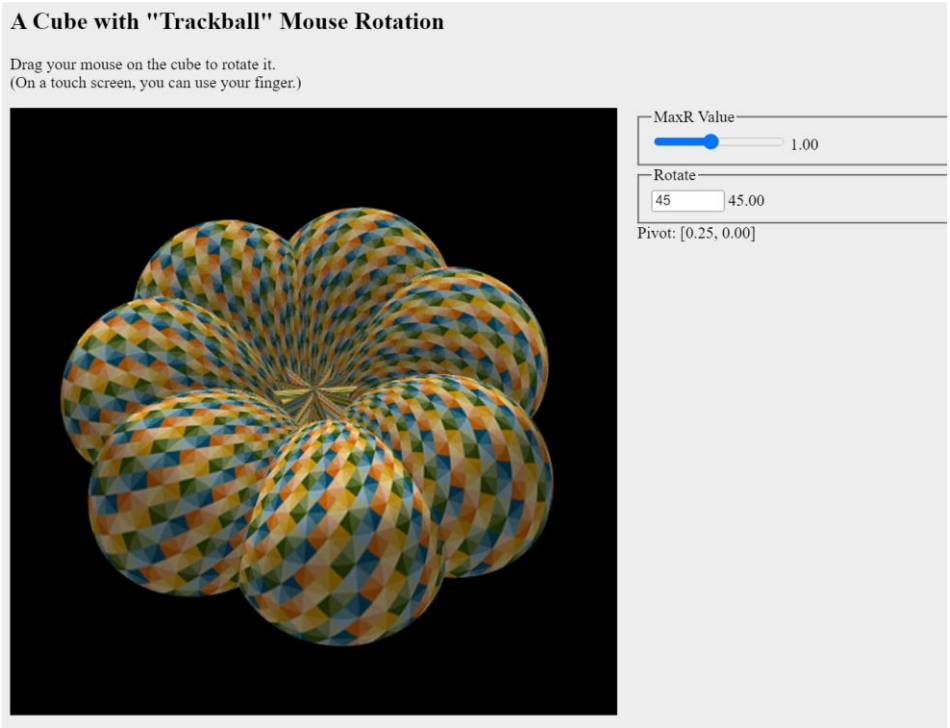

Рисунок 2. – Обертання текстури на 45 градусів

Натискаючи клавіші W, A, S, D, ми можемо переміщувати точку на нашій поверхні, відносно якої буде обертатися текстура. Координати точки обертання представлені в полі Pivot. На рисунку 3 представлена обернена на 45 градусів текстура відносно точки (1.0, 1.5).

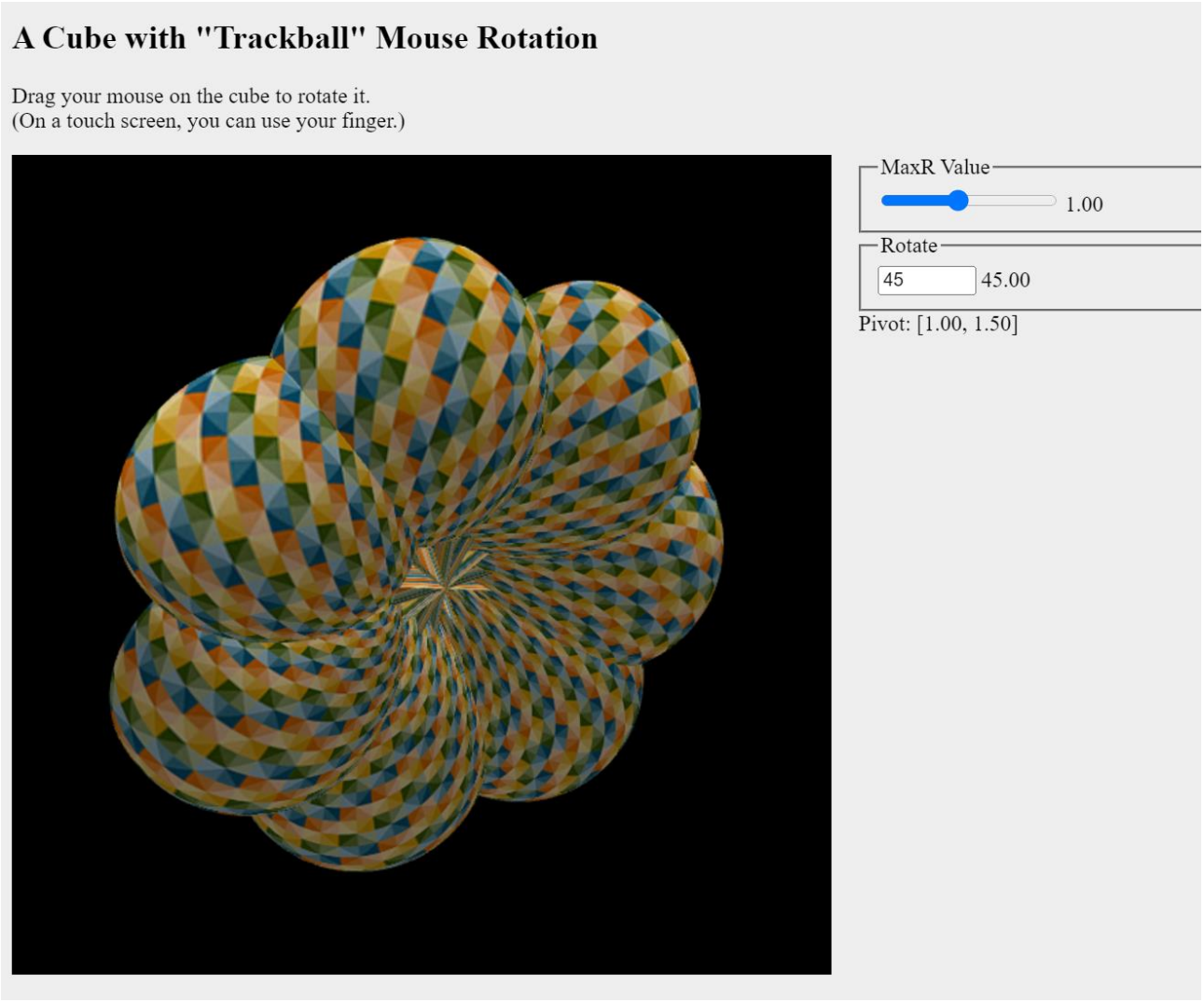

Рисунок 3. – Обертання текстури на 45 градусів навколо точки (1.0, 1.5)

ВИХІДНИЙ КОД

```
function Model(name) {
    this.name = name;
    this.iVertexBuffer = gl.createBuffer();
    this.iNormalBuffer = gl.createBuffer();
            this.iTextCoordBuffer = gl.createBuffer();
    this.count = 0;

    this.BufferData = function(vertices, normal, textCoords) {

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STREAM_DRAW);
```

```
        gl.bindBuffer(gl.ARRAY_BUFFER, this.iNormalBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normal), gl.STREAM_DRAW);

                gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextCoordBuffer);
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textCoords), gl.STREAM_DRAW);

        this.count = vertices.length/3;
    }

    this.Draw = function() {

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

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iNormalBuffer);
        gl.vertexAttribPointer(shProgram.iAttribNormal, 3, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribNormal);

                gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextCoordBuffer);
        gl.vertexAttribPointer(shProgram.iAttribTextCoord, 2, gl.FLOAT, false, 0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribTextCoord);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, this.count);
    }
}

function ShaderProgram(name, program) {

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

    // Location of the attribute variable in the shader program.
    this.iAttribVertex = -1;

    this.iAttribNormal = -1;

        this.iAttribTextCoord = -1;

    this.iLightPosition = -1;

        this.iAngleInRadians = -1;

    this.iUserPoint = -1;

    // Location of the uniform matrix representing the combined transformation.
    this.iModelViewProjectionMatrix = -1;

    this.iModelMatrixNormal = -1;

        this.iTMU = -1;

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

```
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 / 6, 1, 8, 15);

   /* 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,-11);

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

   let modelViewProjection = m4.multiply(projection, matAccum1 );

   let inversion = m4.inverse(modelViewProjection);
   let transposedModel = m4.transpose(inversion);

   gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection );
   gl.uniformMatrix4fv(shProgram.iModelMatrixNormal, false, transposedModel );
   gl.uniform3fv(shProgram.iLightPosition, [0.0, -1.0, 0.0]);

         gl.uniform3fv(shProgram.iSpotDirection, [0.0, 1.0, 0.0]);
         gl.uniform1f(shProgram.iSpotCutoff, Math.cos(deg2rad(40.0)));
         gl.uniform1f(shProgram.iSpotExponent, 10.0);

         gl.uniform1f(shProgram.iAngleInRadians, deg2rad(angle));
   gl.uniform2fv(shProgram.iUserPoint, userPoint);

         gl.uniform1i(shProgram.iTMU, 0);

   surface.Draw();
}

function CreateSurfaceData() {
   let vertexList = [];
   let normalList = [];
         let textCoordList = [];
   let step = 0.03;
   let delta = 0.001;

   for (let u = -3.5 * Math.PI; u <= 3.5 * Math.PI; u += step) {
      for (let v = 0.005 * Math.PI; v < Math.PI / 2; v += step) {

         let v1 = equations(u, v);
         let v2 = equations(u, v + step);
         let v3 = equations(u + step, v);
         let v4 = equations(u + step, v + step);

         vertexList.push(v1.x, v1.y, v1.z);
         vertexList.push(v2.x, v2.y, v2.z);
         vertexList.push(v3.x, v3.y, v3.z);

         vertexList.push(v2.x, v2.y, v2.z);
         vertexList.push(v4.x, v4.y, v4.z);
         vertexList.push(v3.x, v3.y, v3.z);
```

```
            let n1 = CalculateNormal(u, v, delta);
            let n2 = CalculateNormal(u, v + step, delta);
            let n3 = CalculateNormal(u + step, v, delta);
            let n4 = CalculateNormal(u + step, v + step, delta)

            normalList.push(n1.x, n1.y, n1.z);
            normalList.push(n2.x, n2.y, n2.z);
            normalList.push(n3.x, n3.y, n3.z);

            normalList.push(n2.x, n2.y, n2.z);
            normalList.push(n4.x, n4.y, n4.z);
            normalList.push(n3.x, n3.y, n3.z);

                                let t1 = CalculateTextCoord(u, v);
            let t2 = CalculateTextCoord(u, v + step);
            let t3 = CalculateTextCoord(u + step, v);
            let t4 = CalculateTextCoord(u + step, v + step);

            textCoordList.push(t1.u, t1.v);
            textCoordList.push(t2.u, t2.v);
            textCoordList.push(t3.u, t3.v);

            textCoordList.push(t2.u, t2.v);
            textCoordList.push(t4.u, t4.v);
            textCoordList.push(t3.u, t3.v);
        }
    }

    return { vertices: vertexList, normal: normalList, textCoords: textCoordList };
}


function CalculateNormal(u, v, delta) {
    let currentPoint = equations(u, v);
    let pointR = equations(u + delta, v);
    let pointTheta = equations(u, v + delta);

    let dg_du = {
        x: (pointR.x - currentPoint.x) / delta,
        y: (pointR.y - currentPoint.y) / delta,
        z: (pointR.z - currentPoint.z) / delta
    };

    let dg_dv = {
        x: (pointTheta.x - currentPoint.x) / delta,
        y: (pointTheta.y - currentPoint.y) / delta,
        z: (pointTheta.z - currentPoint.z) / delta
    };

    let normal = cross(dg_du, dg_dv);

    normalize(normal);

    return normal;
}

function CalculateTextCoord(u, v) {
```

```
    u = (u - 0.25)/(maxR - 0.25);
    v = v / 2*Math.PI;

    return {u, v};
}

function equations(u, v) {
    let C = 2;
    let fiU = -u / (Math.sqrt(C + 1)) + Math.atan(Math.sqrt(C + 1) * Math.tan(u));
    let aUV = 2 / (C + 1 - C * Math.pow(Math.sin(v), 2) * Math.pow(Math.cos(u), 2));
    let rUV = (aUV / Math.sqrt(C)) * Math.sqrt((C + 1) * (1 + C * Math.pow(Math.sin(u), 2))) * Math.sin(v);

    let x = rUV * Math.cos(fiU);
    let y = rUV * Math.sin(fiU);
    let z = (Math.log(Math.tan(v / 2)) + aUV * (C + 1) * Math.cos(v)) / Math.sqrt(C);

    return { x: x, y: y, z: z };
}

function cross(a, b) {
    let x = a.y * b.z - b.y * a.z;
    let y = a.z * b.x - b.z * a.x;
    let z = a.x * b.y - b.x * a.y;
    return { x: x, y: y, z: z }
}

function normalize(a) {
    var b = Math.sqrt(a.x * a.x + a.y * a.y + a.z * a.z);
    a.x /= b;
    a.y /= b;
    a.z /= b;
}

function updateSurface() {
    maxR = parseFloat(document.getElementById("paramR").value);
    angle = parseFloat(document.getElementById("angle").value);

    let data = CreateSurfaceData(maxR);
    surface.BufferData(data.vertices, data.normal, data.textCoords);

    document.getElementById("currentMaxR").textContent = maxR.toFixed(2);
    document.getElementById("currentAngle").textContent = angle.toFixed(2);

    const userPointElement = document.getElementById("userPointValues");
    userPointElement.textContent = `[${userPoint[0].toFixed(2)}, ${userPoint[1].toFixed(2)}]`;

    draw();
}


function LoadTexture() {

    var texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);

    //gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, new
Uint8Array([0,0,255,255]));
```

```
    var image = new Image();
    image.crossOrigin = "anonymous";
    image.src = "https://i.ibb.co/b5xQL8G/texture5.jpg";
    image.addEventListener('load', () => {
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

        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.iAttribNormal              = gl.getAttribLocation(prog, "normal");
    shProgram.iModelViewProjectionMatrix = gl.getUniformLocation(prog, "ModelViewProjectionMatrix");
    shProgram.iModelMatrixNormal         = gl.getUniformLocation(prog, "ModelNormalMatrix");

        shProgram.iLightPosition = gl.getUniformLocation(prog, "lightPosition");
        shProgram.iSpotDirection = gl.getUniformLocation(prog, "spotDirection");
        shProgram.iSpotCutoff = gl.getUniformLocation(prog, "spotCutoff");
        shProgram.iSpotExponent = gl.getUniformLocation(prog, "spotExponent");

        shProgram.iAttribTextCoord           = gl.getAttribLocation(prog, "textCoord");
    shProgram.iTMU                       = gl.getUniformLocation(prog, "tmu");
    shProgram.iAngleInRadians            = gl.getUniformLocation(prog, "angleInRadians");
    shProgram.iUserPoint                 = gl.getUniformLocation(prog, "userPoint");

    surface = new Model('Surface');
    let data = CreateSurfaceData(1);
    surface.BufferData(data.vertices, data.normal, data.textCoords);

        LoadTexture()

    gl.enable(gl.DEPTH_TEST);
}


/* Creates a program for use in the WebGL context gl, and returns the
 * identifier for that program.  If an error occurs while compiling or
 * linking the program, an exception of type Error is thrown.  The error
 * string contains the compilation or linking error.  If no error occurs,
 * the program identifier is the return value of the function.
 * The second and third parameters are strings that contain the
 * source code for the vertex shader and for the fragment shader.
 */
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);
          document.addEventListener('keydown', handleKeyPress);

    updateSurface();
}

function handleKeyPress(event) {
    let stepSize = 0.05;

    switch (event.key) {
        case 'w':
        case 'W':
            userPoint[0] += stepSize;
            if (userPoint[0] > maxR)
```

```
            {
                userPoint[0] = 0.25;
            }
            break;
        case 's':
        case 'S':
            userPoint[0] -= stepSize;
            if (userPoint[0] < 0.25)
            {
                userPoint[0] = maxR;
            }
            break;
        case 'a':
        case 'A':
            userPoint[1] -= stepSize;
            if (userPoint[1] < 0)
            {
                userPoint[1] = 2 * Math.PI;;
            }
            break;
        case 'd':
        case 'D':
            userPoint[1] += stepSize;
            if (userPoint[1] > 2 * Math.PI)
            {
                userPoint[1] = 0;
            }
            break;
        default:
            return;
    }
    updateSurface();
}
```