Міністерство освіти й науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота

З дисципліни «Візуалізація графічної та геометричної інформації»

Варіант 6

Виконала:

студентка 5-го курсу

групи ТР-23мп

Дмитрик Ольга

Київ-2022

***Тема роботи***: Операціїї над тектурними координатами

***Вимоги:***

* Накласти текстуру на поверхню отриману в результаті виконання лабораторної роботи №2.
* Імплементувати масштабування або обертання текстури(текстурних координат) згідно з варіантом: непарні - масштабування, парні - обертання.
* Запровадити можливість переміщення точки відносно якої відбувається трансформація текстури по поверхні за рахунок зміни параметрів в просторі текстури. Наприклад, клавіші A та D для переміщення по осі абсцис, змінюючи параметр u текстури, а клавіші W та S по осі ординат, змінюючи параметр v.

***Теоретичні відомості***

Текстурування є дуже важливою частиною процесу 3D-моделювання. Усі дрібніші візуальні характеристики у 3D-моделюванні, такі як зморшки та окремі нитки килима, є продуктом текстури, нанесеної 3D-художником. Зазвичай створювані 3D-моделі мають стандартний сірий колір програми. Щоб додати кольори, малюнки та текстури, 2D-фотографії потрібно розмістити на 3D-моделях. Додавання кольорів або властивостей поверхні та матеріалу до 3D-моделі вимагає ще одного кроку вперед у процесі 3D-моделювання, тобто 3D-текстурування. Цей підхід часто призводить до повного кольору та властивостей поверхні 3D-моделі.

Стандартна процедура текстурування така:

UV Mapping and Unwrapping

Щоб почати процес 3D-текстурування, необхідно спочатку розгорнути модель, що, по суті, те саме, що розгортання 3D-сітки. Коли художники-фактуристи отримають готові моделі від відділу 3D-моделювання, вони створять UV-карту для кожного 3D-об’єкта. UV-карта — це плоске зображення поверхні 3D-моделі, яке використовується для швидкого накладання текстур. Прямо пов’язуючи 2D-зображення (текстуру) з вершинами багатокутника, UV-відображення може допомогти обернути 2D-зображення (текстуру) навколо 3D-об’єкта, а згенеровану карту можна використовувати безпосередньо в процесі текстурування та затінення.

Більшість програмних систем 3D мають кілька інструментів або підходів для розгортання 3D-моделей. Коли справа доходить до створення UV-карт, це питання особистих уподобань. Якщо ви не збираєтеся використовувати процедурні текстури, майже завжди потрібно розгортати 3D-модель у компоненті текстурування. Це текстури, створені за допомогою математичних методів (процесів), а не безпосередньо записаних даних у 2D або 3D.

***Виконання завдання***

В ході другої лабораторної роботи було створено поверхню під назвою «Klein Bottle». Отриману поверхню з освітленням можна побачити на рисунку 1.

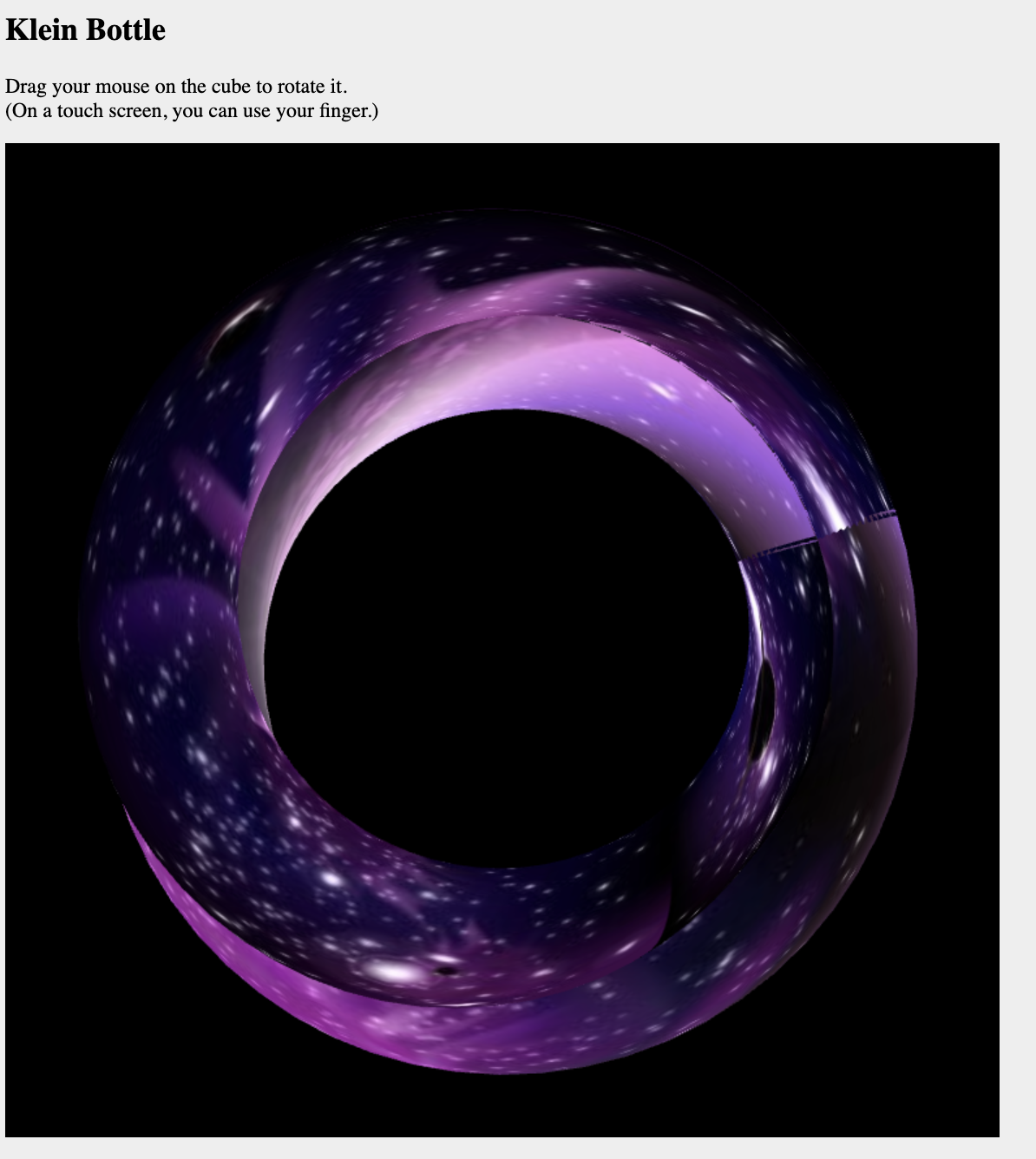


Рис.1. «Klein Bottle» з освітленням

Для текстури було обрано картинку з інтернету формату «jpg». Після чого завантажила її на github, щоб в подальшому використовувати посилання на неї і не стикатися з проблемою Cross-Origin Resource Sharing policy. В графічному редакторі було налаштувано розмір картинки так, щоб ширина і висота були рівні, а також, аби сторона мала розмір 2n в пікселях.

З метою накладання текстури на поверхню, в першу чергу було створено декілька змінних в коді шейдера. Після чого були створені посилання на них в коді програми. Були також створені функції для генерації бUVера даних текстури. Обрану картинку можна побачити на рисунку 2.



Рис. 2 Обрана текстура

Поверхню з накладеною текстурою можна побачити на рисунку 3.



Рис. 3 «Klein Bottle» з накладеною текстурою

Для відображення умовної точки відносно якої буде виконуватися трансформація текстури, в класі моделі було створено відповідну функцію. Замість відображення точки було прийнято рішення відобжати сферу, адже працюємо в 3д-просторі. Для відображення сфери необхідно було створити функцію, яка б створювала геометрію для неї. Модель з умовною точкою зображено на рисунку 4.



Рис. 4 Поверхня з умовною точкою

Для роботи з текстурою було створено ще кілька змінних в коді шейдера: обертання текстури, розташування умовної точки в (u,v) координатах, змінну для розташування сфери на відповідне місце поверхні в 3д-просторі. Для реалізації переміщення точки по поверхні та обертання текстури було додано відповідні функції на відповідні вхідні дані від користувача.

***Вказівки користувачу***

Користувач може керувати переміщенням умовної точки по поверхні, обертанням текстури відносно умовної точки, а також орієнтацією поверхні в просторію При чому останні два пункти здійснюються в один і той же спосіб.

Переміщення умовної точки реалізовано за допомогою введення з клавіатури (рисунок 5): клавіші W та S здійснюють переміщення точки за параметром v в додатньому та від’ємному напрамках відповідно, клавіші A та D здійснюють переміщення точки за параметром u у від’ємному та додатньому напрямках відповідно.

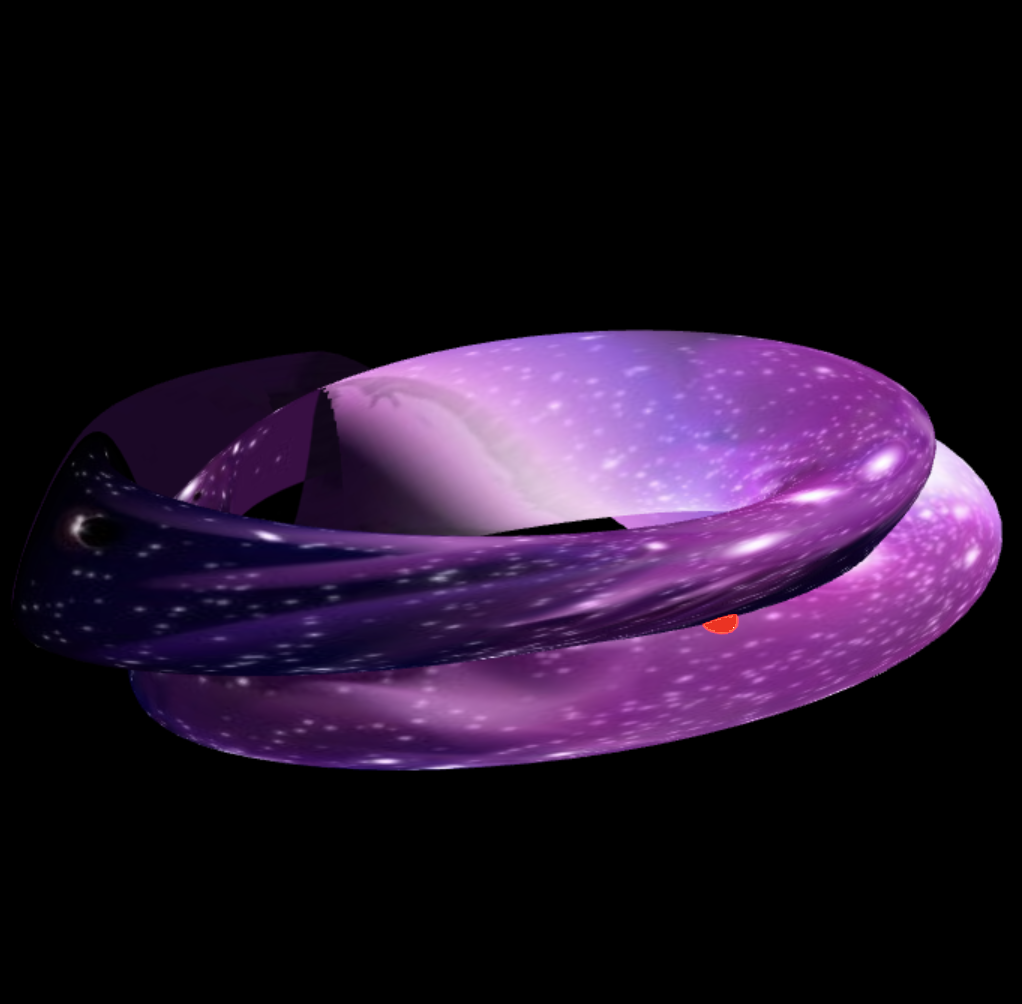
 

Рис. 5. Переміщення умовної точки

***Код***

main.js

'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 sphere;

let userPointCoord;

let userRotAngle;

function deg2rad(angle) {

return angle \* Math.PI / 180;

}

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = gl.createBuffer();

this.iNormalBuffer = gl.createBuffer();

this.iTextureBuffer = gl.createBuffer();

this.count = 0;

this.countT = 0;

this.BufferData = function (vertices) {

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STREAM\_DRAW);

this.count = vertices.length / 3;

}

this.NormalBufferData = function (normals) {

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iNormalBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(normals), gl.STREAM\_DRAW);

this.count = normals.length / 3;

}

this.TextureBufferData = function (points) {

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTextureBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(points), gl.STREAM\_DRAW);

this.countT = points.length / 2;

}

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.iTextureBuffer);

gl.vertexAttribPointer(shProgram.iAttribTexture, 2, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(shProgram.iAttribTexture);

gl.drawArrays(gl.TRIANGLE\_STRIP, 0, this.count);

}

this.DrawSphere = function () {

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(shProgram.iAttribVertex);

gl.drawArrays(gl.LINE\_STRIP, 0, this.count);

}

}

// Constructor

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.iAttribTexture = -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;

this.lightPosLoc = -1;

this.iUserPoint = -1;

this.irotAngle = 0;

this.iUP = -1;

this.iTMU = -1;

this.Use = function () {

gl.useProgram(this.prog);

}

}

/\* Draws a colored cube, along with a set of coordinate axes.

\* (Note that the use of the above drawPrimitive function is not an efficient

\* way to draw with WebGL. Here, the geometry is so simple that it doesn't matter.)

\*/

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);

let para = 3

let projection = m4.orthographic(-para, para, -para, para, 0, para \* 4);

/\* 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);

gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection);

let modelviewInv = new Float32Array(16);

let normalmatrix = new Float32Array(16);

mat4Invert(modelViewProjection, modelviewInv);

mat4Transpose(modelviewInv, normalmatrix);

gl.uniformMatrix4fv(shProgram.iNormalMatrix, false, normalmatrix);

/\* Draw the six faces of a cube, with different colors. \*/

gl.uniform4fv(shProgram.iColor, [0.2, 0.8, 0, 1]);

gl.uniform3fv(shProgram.lightPosLoc, [5 \* Math.cos(Date.now() \* 0.005), (5 \* Math.sin(Date.now() \* 0.005)) \*\* 2, 0]);

gl.uniform2fv(shProgram.iUserPoint, [userPointCoord.x, userPointCoord.y]); //giving coordinates of user point

gl.uniform1f(shProgram.irotAngle, userRotAngle);

gl.uniform2fv(shProgram.iUserPoint, [userPointCoord.x, userPointCoord.y]);

surface.Draw();

let t = kleinBottle(map(userPointCoord.x, 0, 1, 0, Math.PI\*2), map(userPointCoord.y, 0, 1, 0, Math.PI\*2))

gl.uniform3fv(shProgram.iUP, [t.x, t.y, t.z]);

gl.uniform2fv(shProgram.iUserPoint, [0.0, 0.0]);

sphere.DrawSphere();

}

function animation() {

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);

let projection = m4.orthographic(-4, 4, -4, 4, 0, 4 \* 4);

/\* 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);

gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, modelViewProjection);

let modelviewInv = new Float32Array(16);

let normalmatrix = new Float32Array(16);

mat4Invert(modelViewProjection, modelviewInv);

mat4Transpose(modelviewInv, normalmatrix);

gl.uniformMatrix4fv(shProgram.iNormalMatrix, false, normalmatrix);

/\* Draw the six faces of a cube, with different colors. \*/

gl.uniform4fv(shProgram.iColor, [0.2, 0.8, 0, 1]);

gl.uniform3fv(shProgram.lightPosLoc, [5 \* Math.cos(Date.now() \* 0.005), (5 \* Math.sin(Date.now() \* 0.005)) \*\* 2, 0]);

surface.Draw();

window.requestAnimationFrame(animation)

}

function dot(a, b) {

let c = [(a[1] \* b[2] - a[2] \* b[1]), (a[0] \* b[2] - b[0] \* a[2]), (a[0] \* b[1] - a[1] \* b[0])]

return c

}

function normalize(a) {

let d = Math.sqrt(a[0] \*\* 2 + a[1] \*\* 2 + a[2] \*\* 2)

let n = [a[0] / d, a[1] / d, a[2] / d]

return n;

}

function CreateTextureData() {

let texCoordList = [];

let i = 0;

let j = 0;

while (i <= Math.PI \* 2) {

while (j <= Math.PI \* 2) {

let u = map(i, 0, Math.PI \* 2, 0, 1);

let v = map(j, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

u = map(i + 0.1, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

u = map(i, 0, Math.PI \* 2, 0, 1);

v = map(j + 0.1, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

u = map(i + 0.1, 0, Math.PI \* 2, 0, 1);

v = map(j, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

u = map(i + 0.1, 0, Math.PI \* 2, 0, 1);

v = map(j + 0.1, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

u = map(i, 0, Math.PI \* 2, 0, 1);

v = map(j + 0.1, 0, Math.PI \* 2, 0, 1);

texCoordList.push(u, v);

j += 0.1;

}

j = 0

i += 0.1;

}

return texCoordList;

}

function map(val, f1, t1, f2, t2) {

let m;

m = (val - f1) \* (t2 - f2) / (t1 - f1) + f2

return Math.min(Math.max(m, f2), t2);

}

function CreateSurfaceData(norms = false) {

let vertexList = [];

let normalsList = [];

let i = 0;

let j = 0;

let d = 0.1

while (i <= Math.PI \* 2) {

while (j <= Math.PI \* 2) {

let v1 = kleinBottle(i, j)

let v2 = kleinBottle(i + 0.1, j)

let v3 = kleinBottle(i, j + 0.1)

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

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

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

let v4 = kleinBottle(i + 0.1, j + 0.1);

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 v21 = { x: v2.x - v1.x, y: v2.y - v1.y, z: v2.z - v1.z }

let v31 = { x: v3.x - v1.x, y: v3.y - v1.y, z: v3.z - v1.z }

let n1 = vec3Cross(v21, v31);

vec3Normalize(n1);

normalsList.push(n1.x, n1.y, n1.z);

normalsList.push(n1.x, n1.y, n1.z);

normalsList.push(n1.x, n1.y, n1.z);

let v42 = { x: v4.x - v2.x, y: v4.y - v2.y, z: v4.z - v2.z };

let v32 = { x: v3.x - v2.x, y: v3.y - v2.y, z: v3.z - v2.z };

let n2 = vec3Cross(v42, v32);

vec3Normalize(n2);

normalsList.push(n2.x, n2.y, n2.z);

normalsList.push(n2.x, n2.y, n2.z);

normalsList.push(n2.x, n2.y, n2.z);

j += d

}

// if (d > 0) {

// j = 1;

// d \*= -1;

// }

// else {

// j = -1;

// d \*= -1;

// }

j = 0;

i += 0.1;

}

if (norms) {

return normalsList;

}

return vertexList;

}

function kleinBottle(u, v) {

let a = 4

let x = (a + Math.cos(u / 2) \* Math.sin(v) - Math.sin(u / 2) \* Math.sin(2 \* v)) \* Math.cos(u)

let y = (a + Math.cos(u / 2) \* Math.sin(v) - Math.sin(u / 2) \* Math.sin(2 \* v)) \* Math.sin(u)

let z = Math.sin(u / 2) \* Math.sin(v) + Math.cos(u / 2) \* Math.sin(2 \* v);

return { x: 0.5 \* x, y: 0.5 \* y, z: 0.5 \* z }

}

function CreateSphereSurface(r = 0.1) {

let vertexList = [];

let lon = -Math.PI;

let lat = -Math.PI \* 0.5;

while (lon < Math.PI) {

while (lat < Math.PI \* 0.5) {

let v1 = sphereSurfaceDate(r, lon, lat);

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

lat += 0.05;

}

lat = -Math.PI \* 0.5

lon += 0.05;

}

return vertexList;

}

function sphereSurfaceDate(r, u, v) {

let x = r \* Math.sin(u) \* Math.cos(v);

let y = r \* Math.sin(u) \* Math.sin(v);

let z = r \* Math.cos(u);

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

}

function vec3Cross(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 vec3Normalize(a) {

var mag = Math.sqrt(a[0] \* a[0] + a[1] \* a[1] + a[2] \* a[2]);

a[0] /= mag; a[1] /= mag; a[2] /= mag;

}

/\* 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.iAttribTexture = gl.getAttribLocation(prog, "texCoord");

shProgram.iModelViewProjectionMatrix = gl.getUniformLocation(prog, "ModelViewProjectionMatrix");

shProgram.iNormalMatrix = gl.getUniformLocation(prog, "NormalMatrix");

shProgram.iColor = gl.getUniformLocation(prog, "color");

shProgram.lightPosLoc = gl.getUniformLocation(prog, "lightPos");

shProgram.iTMU = gl.getUniformLocation(prog, 'tmu');

shProgram.iUserPoint = gl.getUniformLocation(prog, 'userPoint');

shProgram.irotAngle = gl.getUniformLocation(prog, 'rotA');

shProgram.iUP = gl.getUniformLocation(prog, 'translateUP');

surface = new Model('Surface');

sphere = new Model('Sphere');

surface.BufferData(CreateSurfaceData());

surface.NormalBufferData(CreateSurfaceData(1));

LoadTexture();

surface.TextureBufferData(CreateTextureData());

sphere.BufferData(CreateSphereSurface())

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() {

userPointCoord = { x: 0.5, y: 0.5 }

userRotAngle = 0.0;

let canvas;

try {

let resolution = Math.min(window.innerHeight, window.innerWidth);

canvas = document.querySelector('canvas');

gl = canvas.getContext("webgl");

canvas.width = resolution;

canvas.height = resolution;

gl.viewport(0, 0, resolution, resolution);

if (!gl) {

throw "Browser does not support WebGL";

}

}

catch (e) {

document.querySelector('"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);

// window.requestAnimationFrame(animation);

draw();

}

function mat4Transpose(a, transposed) {

var t = 0;

for (var i = 0; i < 4; ++i) {

for (var j = 0; j < 4; ++j) {

transposed[t++] = a[j \* 4 + i];

}

}

}

function mat4Invert(m, inverse) {

var inv = new Float32Array(16);

inv[0] = m[5] \* m[10] \* m[15] - m[5] \* m[11] \* m[14] - m[9] \* m[6] \* m[15] +

m[9] \* m[7] \* m[14] + m[13] \* m[6] \* m[11] - m[13] \* m[7] \* m[10];

inv[4] = -m[4] \* m[10] \* m[15] + m[4] \* m[11] \* m[14] + m[8] \* m[6] \* m[15] -

m[8] \* m[7] \* m[14] - m[12] \* m[6] \* m[11] + m[12] \* m[7] \* m[10];

inv[8] = m[4] \* m[9] \* m[15] - m[4] \* m[11] \* m[13] - m[8] \* m[5] \* m[15] +

m[8] \* m[7] \* m[13] + m[12] \* m[5] \* m[11] - m[12] \* m[7] \* m[9];

inv[12] = -m[4] \* m[9] \* m[14] + m[4] \* m[10] \* m[13] + m[8] \* m[5] \* m[14] -

m[8] \* m[6] \* m[13] - m[12] \* m[5] \* m[10] + m[12] \* m[6] \* m[9];

inv[1] = -m[1] \* m[10] \* m[15] + m[1] \* m[11] \* m[14] + m[9] \* m[2] \* m[15] -

m[9] \* m[3] \* m[14] - m[13] \* m[2] \* m[11] + m[13] \* m[3] \* m[10];

inv[5] = m[0] \* m[10] \* m[15] - m[0] \* m[11] \* m[14] - m[8] \* m[2] \* m[15] +

m[8] \* m[3] \* m[14] + m[12] \* m[2] \* m[11] - m[12] \* m[3] \* m[10];

inv[9] = -m[0] \* m[9] \* m[15] + m[0] \* m[11] \* m[13] + m[8] \* m[1] \* m[15] -

m[8] \* m[3] \* m[13] - m[12] \* m[1] \* m[11] + m[12] \* m[3] \* m[9];

inv[13] = m[0] \* m[9] \* m[14] - m[0] \* m[10] \* m[13] - m[8] \* m[1] \* m[14] +

m[8] \* m[2] \* m[13] + m[12] \* m[1] \* m[10] - m[12] \* m[2] \* m[9];

inv[2] = m[1] \* m[6] \* m[15] - m[1] \* m[7] \* m[14] - m[5] \* m[2] \* m[15] +

m[5] \* m[3] \* m[14] + m[13] \* m[2] \* m[7] - m[13] \* m[3] \* m[6];

inv[6] = -m[0] \* m[6] \* m[15] + m[0] \* m[7] \* m[14] + m[4] \* m[2] \* m[15] -

m[4] \* m[3] \* m[14] - m[12] \* m[2] \* m[7] + m[12] \* m[3] \* m[6];

inv[10] = m[0] \* m[5] \* m[15] - m[0] \* m[7] \* m[13] - m[4] \* m[1] \* m[15] +

m[4] \* m[3] \* m[13] + m[12] \* m[1] \* m[7] - m[12] \* m[3] \* m[5];

inv[14] = -m[0] \* m[5] \* m[14] + m[0] \* m[6] \* m[13] + m[4] \* m[1] \* m[14] -

m[4] \* m[2] \* m[13] - m[12] \* m[1] \* m[6] + m[12] \* m[2] \* m[5];

inv[3] = -m[1] \* m[6] \* m[11] + m[1] \* m[7] \* m[10] + m[5] \* m[2] \* m[11] -

m[5] \* m[3] \* m[10] - m[9] \* m[2] \* m[7] + m[9] \* m[3] \* m[6];

inv[7] = m[0] \* m[6] \* m[11] - m[0] \* m[7] \* m[10] - m[4] \* m[2] \* m[11] +

m[4] \* m[3] \* m[10] + m[8] \* m[2] \* m[7] - m[8] \* m[3] \* m[6];

inv[11] = -m[0] \* m[5] \* m[11] + m[0] \* m[7] \* m[9] + m[4] \* m[1] \* m[11] -

m[4] \* m[3] \* m[9] - m[8] \* m[1] \* m[7] + m[8] \* m[3] \* m[5];

inv[15] = m[0] \* m[5] \* m[10] - m[0] \* m[6] \* m[9] - m[4] \* m[1] \* m[10] +

m[4] \* m[2] \* m[9] + m[8] \* m[1] \* m[6] - m[8] \* m[2] \* m[5];

var det = m[0] \* inv[0] + m[1] \* inv[4] + m[2] \* inv[8] + m[3] \* inv[12];

if (det == 0) return false;

det = 1.0 / det;

for (var i = 0; i < 16; i++) inverse[i] = inv[i] \* det;

return true;

}

function LoadTexture() {

let 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, 512, 512, 0, );

const image = new Image();

image.crossOrigin = 'anonymus';

image.src = "https://raw.githubusercontent.com/IGSystemI/VGGI/CGW/txtr.png";

image.onload = () => {

gl.bindTexture(gl.TEXTURE\_2D, texture);

gl.texImage2D(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

image

);

draw()

}

}

window.onkeydown = (e) => {

// console.log(e.keyCode)

switch (e.keyCode) {

case 87:

userPointCoord.x -= 0.01;

break;

case 83:

userPointCoord.x += 0.01;

break;

case 65:

userPointCoord.y += 0.01;

break;

case 68:

userPointCoord.y -= 0.01;

break;

}

userPointCoord.x = Math.max(0.001, Math.min(userPointCoord.x, 0.999))

userPointCoord.y = Math.max(0.001, Math.min(userPointCoord.y, 0.999))

// console.log(userPointCoord);

draw();

}

onmousemove = (e) => {

userRotAngle = map(e.clientX, 0, window.outerWidth, 0, Math.PI)

// console.log(e.clientX)

draw()

};

shader.gpu

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec3 normal;

attribute vec2 texCoord;

varying vec3 normalInterp;

uniform mat4 ModelViewProjectionMatrix;

uniform mat4 NormalMatrix;

uniform vec3 lightPos;

varying vec4 color;

varying vec2 v\_texcoord;

uniform vec3 translateUP;

uniform vec2 userPoint;

uniform float rotA;

varying float col;

mat4 translation(float tx, float ty, float tz) {

mat4 dst;

dst[0][0] = 1.0;

dst[0][ 1] = 0.0;

dst[0][ 2] = 0.0;

dst[0][ 3] = 0.0;

dst[1][ 0] = 0.0;

dst[1][ 1] = 1.0;

dst[1][ 2] = 0.0;

dst[1][ 3] = 0.0;

dst[2][ 0] = 0.0;

dst[2][ 1] = 0.0;

dst[2][ 2] = 1.0;

dst[2][ 3] = 0.0;

dst[3][ 0] = tx;

dst[3][ 1] = ty;

dst[3][ 2] = tz;

dst[3][ 3] = 1.0;

return dst;

}

mat4 rotation(float angleInRadians) {

mat4 dst;

float c = cos(angleInRadians);

float s = sin(angleInRadians);

dst[0][0] = c;

dst[0][ 1] = s;

dst[0][ 2] = 0.0;

dst[0][ 3] = 0.0;

dst[1][ 0] = -s;

dst[1][ 1] = c;

dst[1][ 2] = 0.0;

dst[1][ 3] = 0.0;

dst[2][ 0] = 0.0;

dst[2][ 1] = 0.0;

dst[2][ 2] = 1.0;

dst[2][ 3] = 0.0;

dst[3][ 0] = 0.0;

dst[3][ 1] = 0.0;

dst[3][ 2] = 0.0;

dst[3][ 3] = 1.0;

return dst;

}

void main() {

mat4 rotate = rotation(rotA);

mat4 t = translation(userPoint.x,userPoint.y,0.0);

mat4 tt = translation(-userPoint.x,-userPoint.y,0.0);

vec4 textureTranslatedOnce = vec4(texCoord,0.0,0.0)\*t;

vec4 textureRotated = textureTranslatedOnce\*rotate;

vec4 textureTranslatedTwice = textureRotated\*tt;

v\_texcoord = vec2(textureTranslatedTwice.x,textureTranslatedTwice.y);

normalInterp = vec3(NormalMatrix \* vec4(normal, 0.0));

vec3 N = normalize(normalInterp);

vec3 L = normalize(lightPos - vertex);

gl\_Position = ModelViewProjectionMatrix \* vec4(vertex,1.0);

float lambertian = max(dot(N, L), 0.0);

float specular = 0.0;

if(lambertian > 0.0) {

vec3 R = reflect(-L, N); // Reflected light vector

vec3 V = normalize(-vertex); // Vector to viewer

// Compute the specular term

float specAngle = max(dot(R, V), 0.0);

specular=pow(specAngle,1.0);

}

vec3 ambientColor = vec3(0.1, 0.0, 0.0);

vec3 diffuseColor = vec3(1.0,0.0,0.0);

vec3 specularColor = vec3(0.0,0.0,0.0);

color = vec4(0.9 \* ambientColor +

0.9 \* lambertian \* diffuseColor +

0.9 \* specular \* specularColor, 1.0);

if(userPoint.x==0.0){

vec4 sphere = translation(translateUP.x,translateUP.y,translateUP.z)\*vec4(vertex,1.0);

gl\_Position=ModelViewProjectionMatrix\*sphere;

}

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

precision highp float;

#else

precision mediump float;

#endif

//uniform vec4 color;

varying vec4 color;

varying vec2 v\_texcoord;

uniform sampler2D tmu;

uniform vec2 userPoint;

void main() {

vec4 texColor = texture2D(tmu, v\_texcoord);

if(userPoint.x==0.0){

texColor= vec4(1.,0.,0.,0.);

}

gl\_FragColor = texColor;

}`;