Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота

З дисципліни «Візуалізація графічної та геометричної інформації»

Варіант 4

Виконав: Власов Ростислав

Студент групи ТР-22мп

Київ 2022

1. Завдання

Тема роботи: Операціїї над тектурними координатами

Вимоги:

* Накласти текступу 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.

1. 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.

1. Виконання завдання

В ході другої лабораторної роботи було створено поверню під назвою

«Richmond`s Minimal Surface». Отриману поверхню з освітленням можна

побачити на рисунку 3.1.



Рис. 3.1 «Richmond`s Minimal Surface» з освітленням

Для текстури було обрано картинку з інтернету формату «jpg». Після чого

завантажив 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 створення

посилання на них в коді програми. Були також створені функції для

генерації UV даних текстури.

Обрану картинку можна побачити на рисунку 3.2.



Рис. 3.2 Обрана текстура

Поверхню з накладеною текстурою можна побачити на рисунку 3.3.

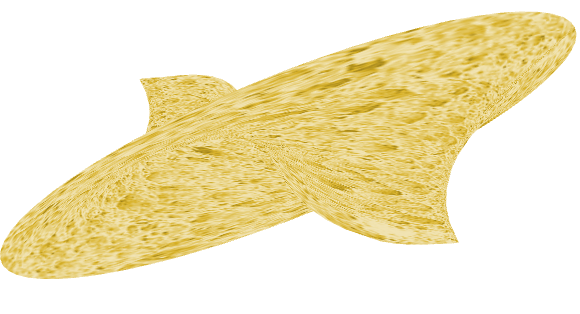


Рис. 3.3 «Richmond`s Minimal Surface ace» з накладеною текстурою

Для відображення умовної точки відносно якої буде виконватися

трансформація текстури, в класі моделі було сворено відповідну функцію.

Замість відображення точки було прийнято рішення відобжати сферу,

адже працюємо в 3д-просторі. Для відорбраження сфери необхідно було

створити функцію, яка б створювала геометрію для неї. Модель з

умовною точкою зображено на рисунку 3.4.

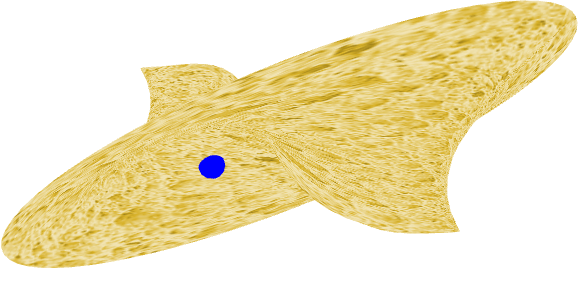


Рис. 3.4 Поверхня з умовною точкою

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.

1. 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 4.1): 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.

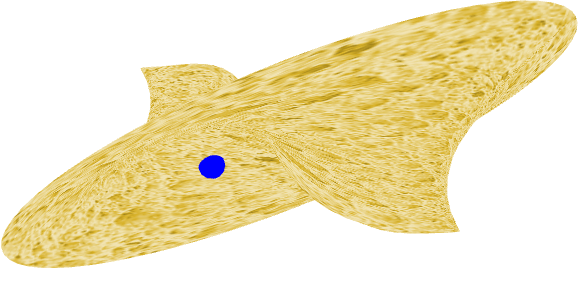
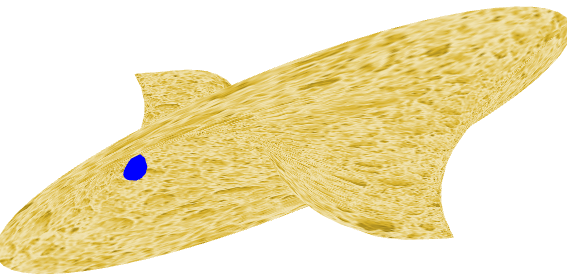
 

Рис. 4.1. Переміщення умовної точки

Орієнтація поверхні в просторі, а також трансформація текстури(рисунок 4.2) здійснюється за допомогою введення з миші: необхідно затиснути лівою клавішею миші у області відображення поверхні та потягнути в будь-яку сторону. Варто зауважити, що обертання текстури залежить від переміщення миші по горизонталі, тобто, при зміщенні затиснутої мишу тільки вгору або тільки вниз буде здійснюватись лише зміна орієнтації поверхні в просторі(рисунок 4.3)

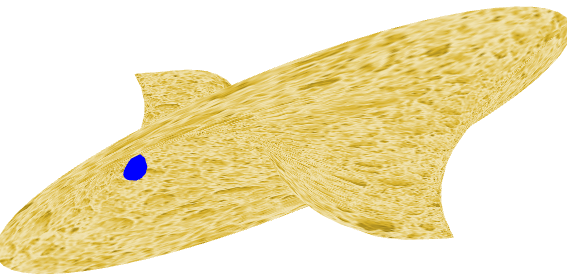
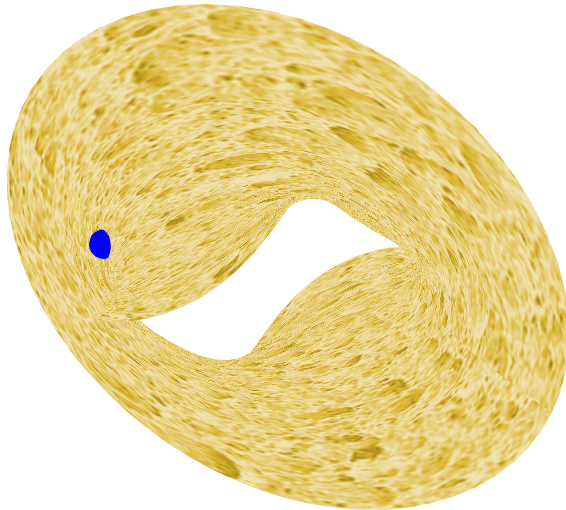
 

Рис. 4.2. Трансформація текстури

На рисунку 4.3 можна помітити що точка та текстура залишились на одному і тому самому місці відносно поверхні. Змінилась лише орієнтація поверхні в просторі.

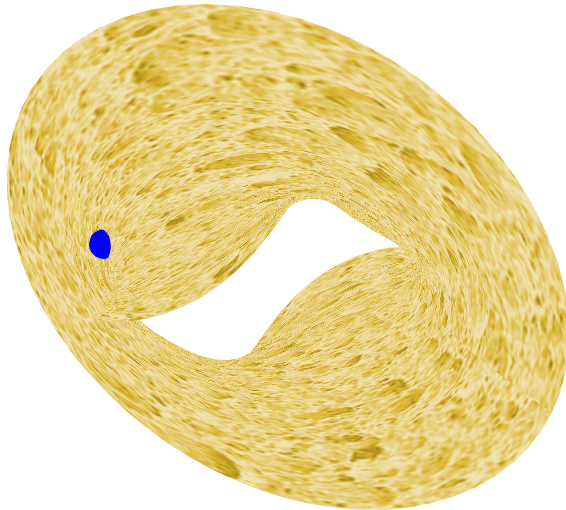
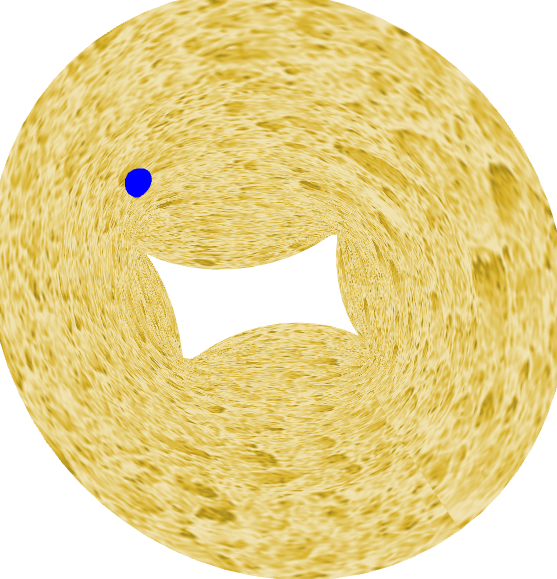
 

Рис. 4.3. Лише орієнтація поверхні в просторі

Код

'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 userPoint;

let angle;

function deg2rad(angle) {

return angle \* Math.PI / 180;

}

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = gl.createBuffer();

this.iVertexTextureBuffer = gl.createBuffer();

this.count = 0;

this.textureCount = 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.TextureBufferData = function (vertices) {

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

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

this.textureCount = vertices.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.iVertexTextureBuffer);

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

gl.enableVertexAttribArray(shProgram.iAttribVertexTexture);

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.TRIANGLE\_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.iAttribVertexTexture = -1;

// Location of the uniform matrix representing the combined transformation.

this.iModelViewProjectionMatrix = -1;

this.iTMU = -1;

this.iUserPoint = -1;

this.iAngle = 0;

this.iTranslateSphere = -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(1., 1., 1., 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);

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

let translate = cylindrical(map(userPoint.x, 0, 1, 0.25, 1), map(userPoint.y, 0, 1, 0, Math.PI\*2))

gl.uniform3fv(shProgram.iTranslateSphere, [translate.x, translate.y, translate.z])

gl.uniform1f(shProgram.iB, 1);

sphere.DrawSphere();

gl.uniform1i(shProgram.iTMU, 0);

gl.enable(gl.TEXTURE\_2D);

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

gl.uniform1f(shProgram.iAngle, angle);

gl.uniform1f(shProgram.iB, -1);

gl.uniform3fv(shProgram.iTranslateSphere, [-0., -0., -0.])

surface.Draw();

}

function CreateSurfaceData() {

let vertexList = [];

const STEP = 0.1

for (let uc = 0.25; uc < 1; uc += STEP) {

for (let vc = 0; vc < Math.PI \* 2; vc += STEP) {

const v1 = cylindrical(uc, vc);

const v2 = cylindrical(uc + STEP, vc);

const v3 = cylindrical(uc, vc + STEP);

const v4 = cylindrical(uc + STEP, vc + 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)

}

}

return vertexList;

}

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 CreateSurfaceTextureData() {

let vertexTextureList = [];

const STEP = 0.1

for (let uc = 0.25; uc < 1; uc += STEP) {

for (let vc = 0; vc < Math.PI \* 2; vc += STEP) {

let u = map(uc, 0.25, 1, 0, 1)

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

vertexTextureList.push(u, v)

u = map(uc + STEP, 0.25, 1, 0, 1)

vertexTextureList.push(u, v)

u = map(uc, 0.25, 1, 0, 1)

v = map(vc + STEP, 0, Math.PI\*2, 0, 1)

vertexTextureList.push(u, v)

u = map(uc + STEP, 0.25, 1, 0, 1)

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

vertexTextureList.push(u, v)

v = map(vc + STEP, 0, Math.PI\*2, 0, 1)

vertexTextureList.push(u, v)

u = map(uc, 0.25, 1, 0, 1)

v = map(vc + STEP, 0, Math.PI\*2, 0, 1)

vertexTextureList.push(u, v)

}

}

return vertexTextureList;

}

function cylindrical(r, theta) {

let x = (-Math.cos(theta) / (2 \* r)) - (r \*\* 3 \* Math.cos(3 \* theta)) / 6;

let y = (-Math.sin(theta) / (2 \* r)) + (r \*\* 3 \* Math.sin(3 \* theta)) / 6;

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

const COEF = 1;

return {

x: x \* COEF,

y: y \* COEF,

z: z \* COEF

}

}

/\* 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.iAttribVertexTexture = gl.getAttribLocation(prog, "vertexTexture");

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

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

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

shProgram.iAngle = gl.getUniformLocation(prog, 'rotate');

shProgram.iTranslateSphere = gl.getUniformLocation(prog, 'translateSphere');

shProgram.iB = gl.getUniformLocation(prog, 'b');

LoadTexture()

surface = new Model('Surface');

surface.BufferData(CreateSurfaceData());

surface.TextureBufferData(CreateSurfaceTextureData());

sphere = new Model('Sphere');

sphere.BufferData(CreateSphereSurface())

gl.enable(gl.DEPTH\_TEST);

}

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);

let v2 = sphereSurfaceDate(r, lon + 0.5, lat);

let v3 = sphereSurfaceDate(r, lon, lat + 0.5);

let v4 = sphereSurfaceDate(r, lon + 0.5, lat + 0.5);

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);

lat += 0.5;

}

lat = -Math.PI \* 0.5

lon += 0.5;

}

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 };

}

/\* 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;

userPoint = { x: 0.5, y: 0.5 }

angle = 0.0;

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();

}

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);

const image = new Image();

image.crossOrigin = 'anonymus';

image.src = "https://raw.githubusercontent.com/Dunadan48/VGGI\_LAB/main/Polantis-Sponge-3d.png";

image.onload = () => {

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

gl.texImage2D(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

image

);

draw()

}

}

onmousemove = (e) => {

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

draw()

};

window.onkeydown = (e) => {

switch (e.keyCode) {

case 87:

userPoint.y -= 0.01;

break;

case 83:

userPoint.y += 0.01;

break;

case 65:

userPoint.x += 0.01;

break;

case 68:

userPoint.x -= 0.01;

break;

}

userPoint.x = Math.max(0.01, Math.min(userPoint.x, 0.999))

userPoint.y = Math.max(0.01, Math.min(userPoint.y, 0.999))

draw();

}

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 vertexTexture;

uniform mat4 ModelViewProjectionMatrix;

varying vec2 v\_texcoord;

uniform float b;

uniform vec3 translateSphere;

uniform vec2 userPoint;

uniform float rotate;

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 rMatrix = rotation(rotate);

mat4 tMatrix1 = translation(userPoint.x,userPoint.y,0.0);

mat4 tMatrix2 = translation(-userPoint.x,-userPoint.y,0.0);

vec4 textureT1 = vec4(vertexTexture,0.,0.)\*tMatrix1;

vec4 textureR = textureT1\*rMatrix;

vec4 textureT2 = textureR\*tMatrix2;

v\_texcoord = vec2(textureT2.x,textureT2.y);

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

if(b>0.0){

vec4 sphere = translation(translateSphere.x,translateSphere.y,translateSphere.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

varying vec2 v\_texcoord;

uniform sampler2D tmu;

uniform float b;

void main() {

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

gl\_FragColor = texColor;

if(b>0.){

gl\_FragColor = vec4(0.,0.,1.,1.);

}

}`;