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 1

Виконав: Бабій І.О.

Студент групи ТР-22мп

Київ 2022

# Завдання

Тема роботи:

Операції з координатами текстури.

Вимоги:

* Нанесіть текстуру на поверхню з практичного завдання #2.
* Реалізуйте масштабування текстури (координати текстури) масштабування/обертання навколо вказаної користувачем точки - непарні варіанти реалізують масштабування, парні варіанти реалізують обертання
* Повинна бути реалізована можливість переміщення точки вздовж простору поверхні (u,v) за допомогою клавіатури. Наприклад, клавіші A і D переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.

# Теоретичні відомості

Текстурування - важлива частина процесу 3D-моделювання. Всі тонкі візуальні характеристики в 3D-моделюванні, такі як зморшки та окремі нитки килима, є продуктом текстури, застосованої 3D-художником. За замовчуванням 3D-моделі, які ви створюєте, мають стандартний сірий колір програми. Щоб додати кольори, візерунки і текстури, потрібно розмістити на 3D-моделях 2D-фотографії. Додавання кольорів або властивостей поверхонь і матеріалів до 3D-моделі вимагає ще одного кроку вперед у процесі 3D-моделювання - 3D-текстурування. Цей підхід часто призводить до отримання повного кольору і властивостей поверхні 3D-моделі.

Стандартна процедура текстурування виглядає наступним чином:

## UV Mapping and Unwrapping

Щоб почати процес 3D-текстурування, потрібно спочатку розгорнути модель, що по суті те саме, що розгортання 3D-сітки. Коли художники текстурування отримають готові моделі з відділу 3D-моделювання, вони створять UV-карту для кожного 3D-об'єкта. UV-карта - це пласке зображення поверхні 3D-моделі, яке використовується для швидкого накладання текстур. Безпосередньо пов'язуючи 2D-зображення (текстуру) з вершинами багатокутника, UV-мапа може допомогти обернути 2D-зображення (текстуру) навколо 3D-об'єкта, а згенерована карта може бути використана безпосередньо в процесі текстурування і затінення.

Більшість програмних 3D-систем 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, вам майже завжди потрібно розгорнути вашу 3D-модель у компонент текстурування. Це текстури, створені за допомогою математичних методів (процесів), а не безпосередньо записаних даних у 2D або 3D.

# Виконання завдання

В ході другої практичної роботи було створено поверхню під назвою «Циклічна поверхня Віріча». Отриману поверхню з освітленням можна побачити на рисунку 3.1.

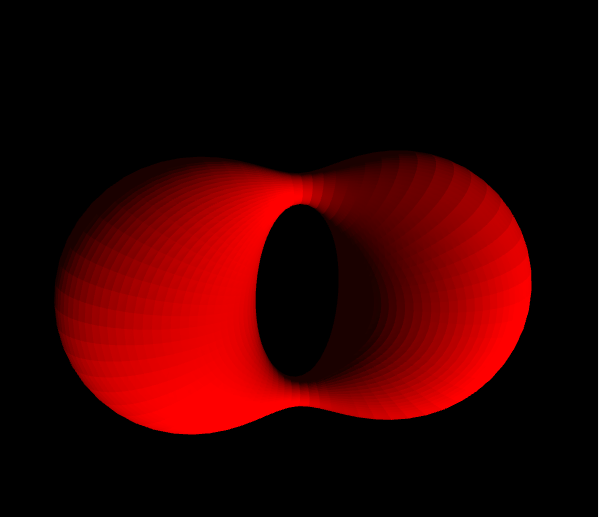


Рис. 3.1 «Циклічна поверхня Віріча» з освітленням

Для текстури було обрано картинку з інтернету формату «png». Після чого її було завантажено на репозиторій у Github, щоб в подальшому використовувати посилання на неї і не стикатися з проблемою Cross-Origin Resource Sharing policy.

В графічному редакторі налаштовано 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 створено посилання на них в коді програми. Також розроблено функції для генерації буфера 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.

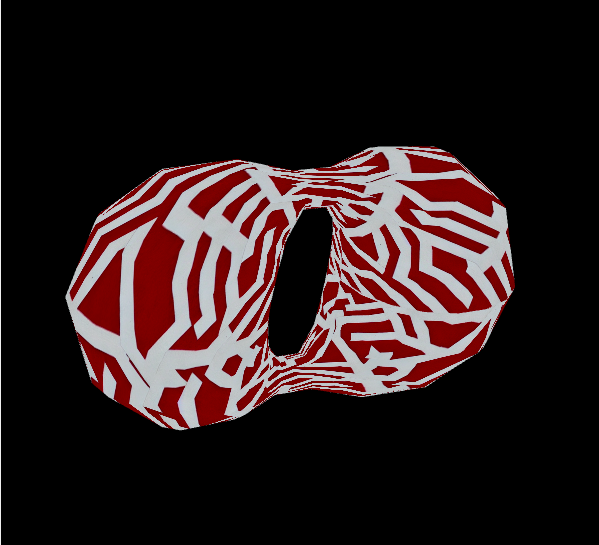


Рис. 3.3 «Циклічна поверхня Віріча» з накладеною текстурою

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 3.4.

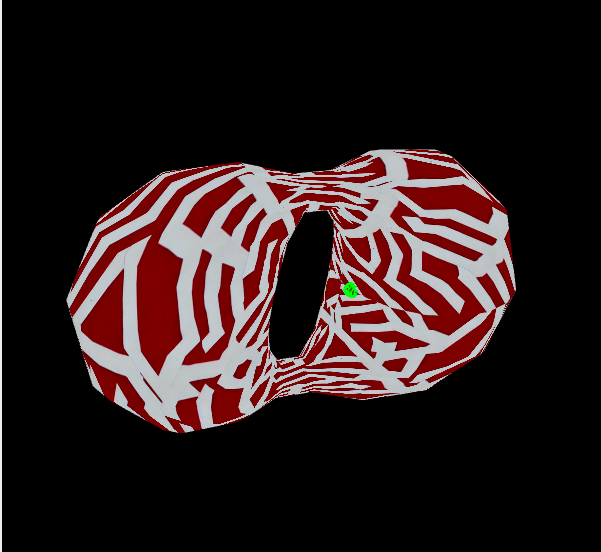


Рис. 3.4 Поверхня з умовною точкою

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 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 просторі. 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 напрямках відповідно, клавіші A та D здійснюють переміщення точки за параметром u у від’ємному та додатньому напрямках відповідно.

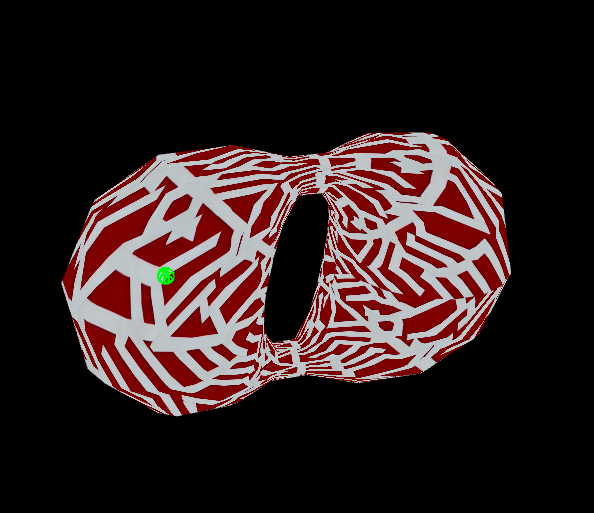
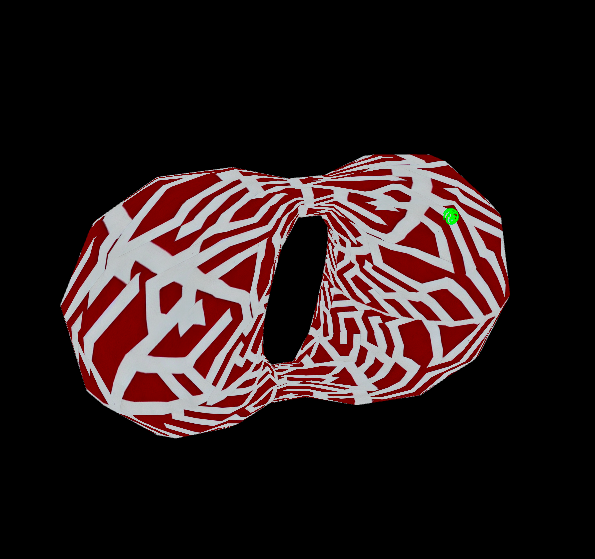
 

Рис. 4.1. Переміщення умовної точки

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)

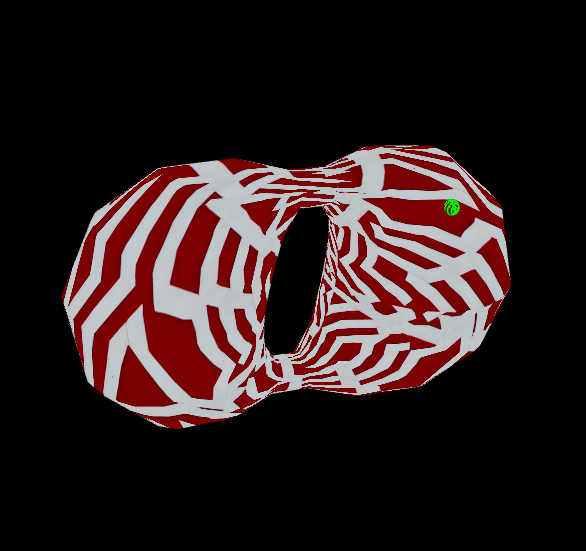
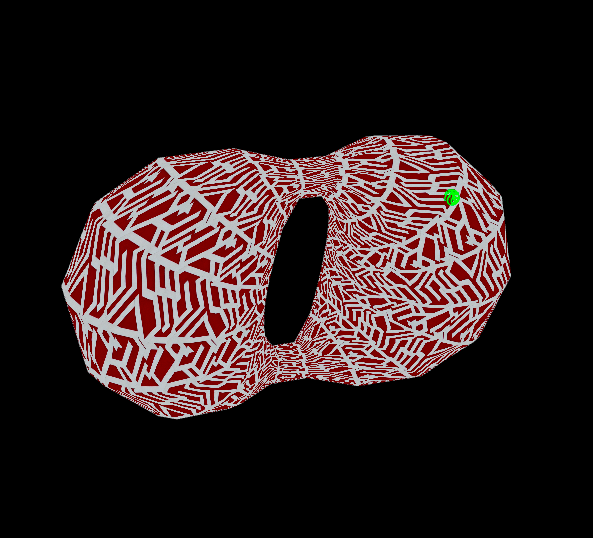
 

Рис. 4.2. Трансформація текстури

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.

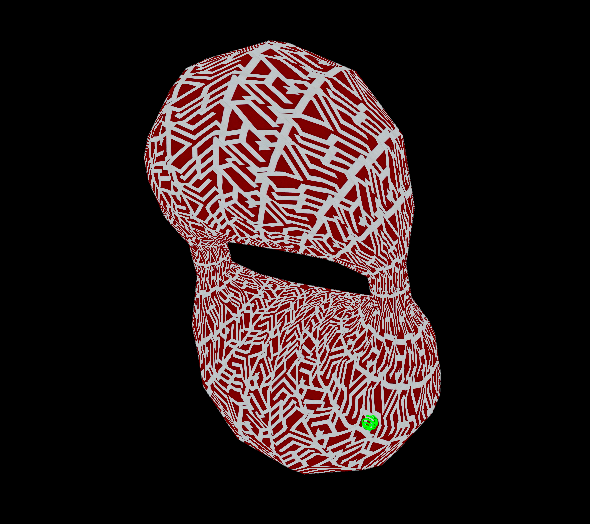
 

Рис. 4.3. Орієнтація поверхні в просторі

# 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 point;

let userPointCoord;

let userScl;

function deg2rad(angle) {

return angle \* Math.PI / 180;

}

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = gl.createBuffer();

this.count = 0;

this.iTextureBuffer = gl.createBuffer();

this.countText = 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 (points) {

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

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

this.countText = 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.iTextureBuffer);

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

gl.enableVertexAttribArray(shProgram.iAttribTexture);

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

}

this.DrawPoint = 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);

}

}

function CreateSphereSurface(r = 0.05) {

let vertexList = [];

let lon = -Math.PI;

let lat = -Math.PI \* 0.5;

while (lon < Math.PI) {

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

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

let v2 = sphereSurfaceData(r, lon+0.1, lat);

let v3 = sphereSurfaceData(r, lon, 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);

lat += 0.1;

}

lat = -Math.PI \* 0.5

lon += 0.5;

}

return vertexList;

}

function sphereSurfaceData(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 };

}

// Constructor

function ShaderProgram(name, program) {

this.name = name;

this.prog = program;

this.iAttribVertex = -1;

this.iAttribTexture = -1;

this.iModelViewProjectionMatrix = -1;

this.iUserPoint = -1;

this.iScale = 1.0;

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);

/\* 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);

gl.uniform1i(shProgram.iTMU, 0);

gl.enable(gl.TEXTURE\_2D);

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

gl.uniform1f(shProgram.iScale, userScl);

surface.Draw();

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

gl.uniform3fv(shProgram.iUP, [loc[0], loc[1], loc[2]])

gl.uniform1f(shProgram.iScale, -1.0);

point.DrawPoint();

}

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 getFunc(t, v) {

let k = 5;

let a = 1.5;

let b = 3;

let c = 2;

let d = 4;

let f = a \* b / Math.sqrt(a \*\* 2 \* Math.sin(v) \*\* 2 + b \*\* 2 \* Math.cos(v) \*\* 2)

let x = 0.5 \* (f \* (1 + Math.cos(t)) + (d \*\* 2 - c \*\* 2) \* (1 - Math.cos(t)) / f) \* Math.cos(v)

let y = 0.5 \* (f \* (1 + Math.cos(t)) + (d \*\* 2 - c \*\* 2) \* (1 - Math.cos(t)) / f) \* Math.sin(v)

let z = 0.5 \* (f - (d \*\* 2 - c \*\* 2) / f) \* Math.sin(t)

return [x / k, y / k, z / k];

}

function CreateSurfaceData() {

let vertexList = [];

let tStep = Math.PI \* 5 / 45;

let vStep = Math.PI \* 5 / 45;

let size = Math.PI \* 2

for (let t = 0; t <= size; t += tStep) {

for (let v = 0; v <= size; v += vStep) {

let vA = getFunc(t, v);

let vB = getFunc(t + tStep, v);

let vC = getFunc(t, v + vStep);

let vD = getFunc(t + tStep, v + vStep);

vertexList = vertexList.concat(vA);

vertexList = vertexList.concat(vB);

vertexList = vertexList.concat(vC);

vertexList = vertexList.concat(vB);

vertexList = vertexList.concat(vD);

vertexList = vertexList.concat(vC);

}

}

return vertexList;

}

function CreateTextureData() {

let vertexList = [];

let tStep = Math.PI \* 5 / 45;

let vStep = Math.PI \* 5 / 45;

let size = Math.PI \* 2

for (let t = 0; t <= size; t += tStep) {

for (let v = 0; v <= size; v += vStep) {

let u1 = map(t, 0, size, 0, 1)

let v1 = map(v, 0, size, 0, 1)

vertexList.push(u1,v1)

u1 = map(t+tStep, 0, size, 0, 1)

vertexList.push(u1,v1)

u1 = map(t, 0, size, 0, 1)

v1 = map(v+vStep, 0, size, 0, 1)

vertexList.push(u1,v1)

u1 = map(t+tStep, 0, size, 0, 1)

v1 = map(v, 0, size, 0, 1)

vertexList.push(u1,v1)

v1 = map(v+vStep, 0, size, 0, 1)

vertexList.push(u1,v1)

u1 = map(t, 0, size, 0, 1)

v1 = map(v+vStep, 0, size, 0, 1)

vertexList.push(u1,v1)

}

}

return vertexList;

}

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

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

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

shProgram.iScale = gl.getUniformLocation(prog, 'scl');

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

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

surface = new Model('Surface');

surface.BufferData(CreateSurfaceData());

LoadTexture()

surface.TextureBufferData(CreateTextureData());

point = new Model('Point');

console.log(CreateSphereSurface())

point.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: 0.0, y: 0.0}

userScl = 1.0;

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();

}

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/ivbabiy-student/CGW/main/pattern.png";

image.onload = () => {

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

gl.texImage2D(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

image

);

console.log("imageLoaded")

draw()

}

}

window.onkeydown = (e) => {

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.abs(userPointCoord.x % 1)

userPointCoord.y = Math.abs(userPointCoord.y % 1)

console.log(userPointCoord);

draw();

}

onmousemove = (e) => {

userScl = map(e.clientX, 0, window.outerWidth, 0.1, 10.0)

draw()

};

## shader.gpu

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 texture;

uniform mat4 ModelViewProjectionMatrix;

varying vec2 v\_texcoord;

uniform vec3 translateUP;

uniform vec2 userPoint;

uniform float scl;

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 scaling(float sx, float sy, float sz){

mat4 dst;

dst[0][0] = sx;

dst[0][ 1] = 0.0;

dst[0][ 2] = 0.0;

dst[0][ 3] = 0.0;

dst[1][ 0] = 0.0;

dst[1][ 1] = sy;

dst[1][ 2] = 0.0;

dst[1][ 3] = 0.0;

dst[2][ 0] = 0.0;

dst[2][ 1] = 0.0;

dst[2][ 2] = sz;

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 sclMatrix = scaling(scl,scl,scl);

mat4 transMatrix1 = translation(userPoint.x,userPoint.y,0.0);

mat4 transMatrix2 = translation(-userPoint.x,-userPoint.y,0.0);

vec4 texture1 = vec4(texture,0.0,0.0)\*transMatrix1;

vec4 textureScaled = texture1\*sclMatrix;

vec4 texture2 = textureScaled\*transMatrix2;

v\_texcoord = vec2(texture2.x,texture2.y);

vec4 vertPos4 = ModelViewProjectionMatrix \* vec4(vertex, 1.0);

gl\_Position = vertPos4;

if(scl<0.){

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

gl\_Position=ModelViewProjectionMatrix\*sphereLoc;

}

}`;

// 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 scl;

void main() {

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

gl\_FragColor = texColor;

if(scl<0.0){

gl\_FragColor = vec4(0.0,1.0,0.,1.);

}

}`;

## index.html

<!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="./main.js"></script>

</head>

<body onload="init()">

<h2>A Virich Cyclic Surface with "Trackball" Mouse Rotation</h2>

<p id="message">Drag your mouse on the to rotate it.<br>

(On a touch screen, you can use your finger.)</p>

<div id="canvas-holder">

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

</div>

</body></html>