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 15

Виконала: Лавро О.М.

Студентка групи ТР-22мп

Київ 2022

**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.

**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.

**Реалізація**

В ході другої лабораторної роботи було створено поверню під назвою

«Дінг-Донг». Отриману поверхню можна побачити на рисунку 1.

Изображение выглядит как текст, пропеллер, темный

Автоматически созданное описание

Рис. 1 «Дінг-Донг поверхня»

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

Було завантажено 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. Після чого були створення

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

генерації буфера даних текстури.

Обрану картинку можна побачити на рисунку 2.



Рис. 2 Обрана текстура

Поверхню з накладеною текстурою можна побачити на рисунку 3

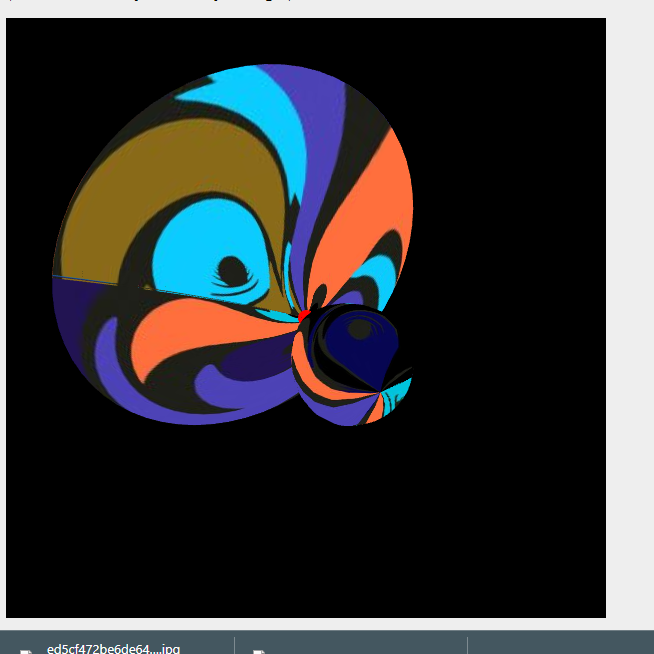


Рис. 3 «Дінг-Донг поверхня» з накладеною текстурою

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 відображення сфери необхідно було

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

умовною точкою зображено на рисунку 3.

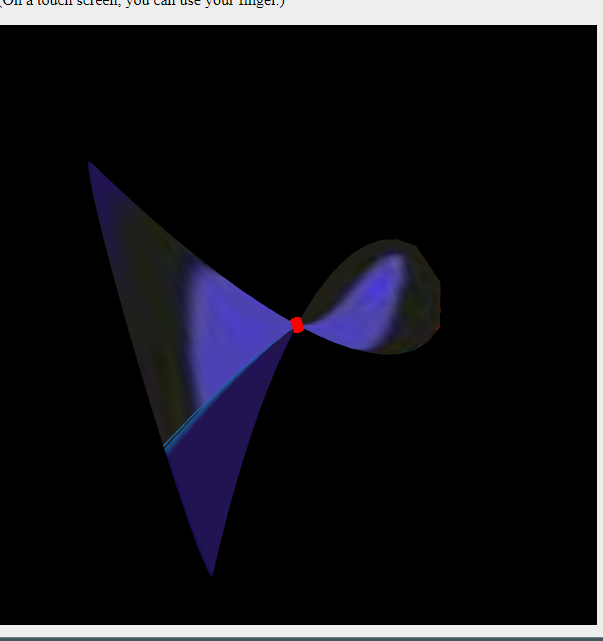


Рис. 3 Поверхня з умовною точкою

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**

Переміщення умовної точки реалізовано за допомогою введення з клавіатури(рисунок 4): 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.

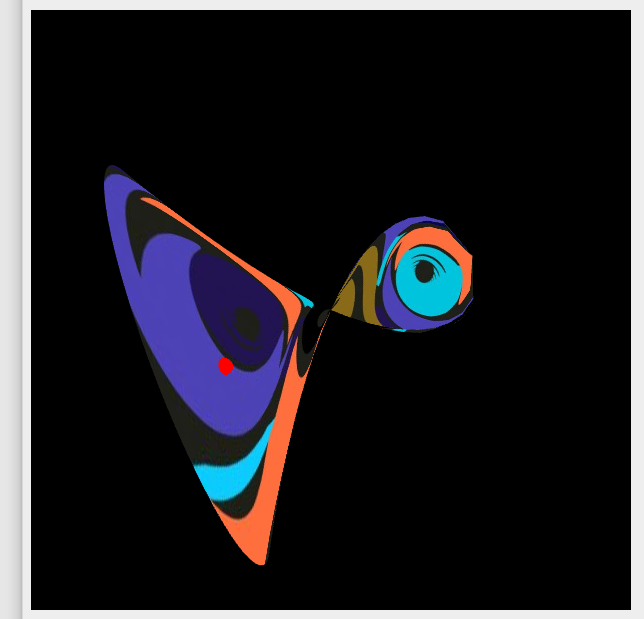
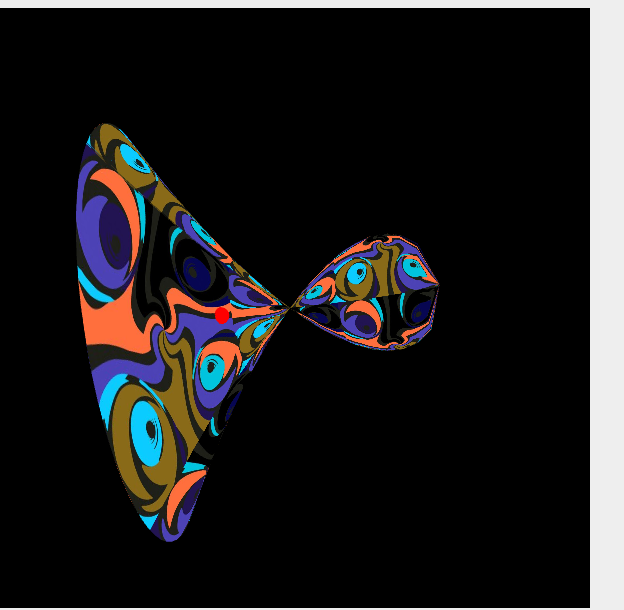
 

Рис. 4. Переміщення умовної точки

Масштабування здійснюється за допомогою миші. Це можна спостерігати на рисунку 5

Изображение выглядит как цветной, цвета, темный, яркий

Автоматически созданное описание

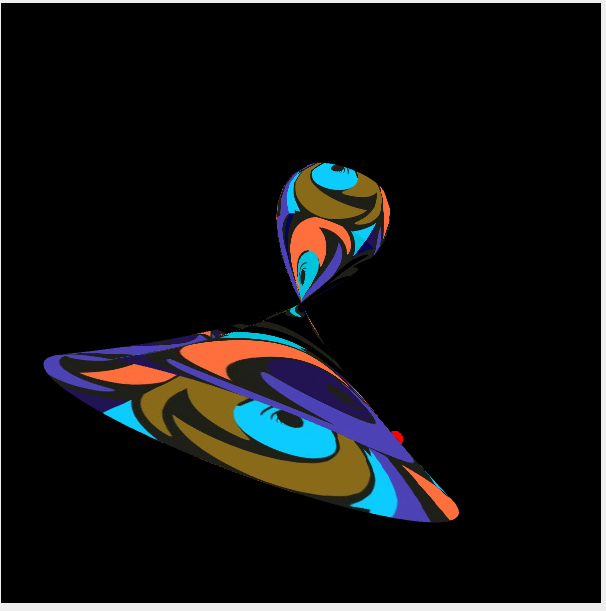


Рис. 5. Масштабування текстури

**Програмний 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; // A model representing point on a surface

let userPointCoord;

let userScaleFactor;

function deg2rad(angle) {

return angle \* Math.PI / 180;

}

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = 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.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.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);

}

}

// Constructor

function ShaderProgram(name, program) {

this.name = name;

this.prog = program;

// Location of the attribute variable in the shader program.

this.iAttribVertex = -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.iTMU = -1;

this.iUserPoint = -1;

this.iScale = 1.0;

this.iUP = -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);

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

gl.uniform1i(shProgram.iTMU, 0);

gl.enable(gl.TEXTURE\_2D);

gl.uniform1f(shProgram.iScale, userScaleFactor)

surface.Draw();

gl.uniform1f(shProgram.iScale, -1.0)

let trS = ding(map(userPointCoord.x, 0, 1, 0, Math.PI\*2), map(userPointCoord.y, 0, 1, -1, 1))

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

sphere.DrawPoint();

}

function animation() {

draw()

window.requestAnimationFrame(animation)

}

function CreateSurfaceData() {

let vertexList = [];

let i = 0;

let j = -1;

let d = 0.1;

while (j <= 1) {

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

let v1 = ding(i, j)

let v2 = ding(i + d, j)

let v3 = ding(i, j + d)

let v4 = ding(i + d, j + d)

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);

i += d

}

i = 0;

j += d

}

return vertexList;

}

function CreateTextureData() {

let texCoordList = [];

let i = 0;

let j = -1;

let d = 0.1;

while (j <= 1) {

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

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

let v = map(j, -1, 1, 0, 1);

texCoordList.push(u, v);

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

texCoordList.push(u, v);

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

v = map(j + d, -1, 1, 0, 1);

texCoordList.push(u, v);

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

v = map(j, -1, 1, 0, 1);

texCoordList.push(u, v);

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

v = map(j + d, -1, 1, 0, 1);

texCoordList.push(u, v);

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

v = map(j + d, -1, 1, 0, 1);

texCoordList.push(u, v);

i += d;

}

i = 0

j += d;

}

return texCoordList;

}

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 = sphereSurfaceData(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 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 };

}

function ding(u, v) {

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

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

let z = v;

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

}

function r(v) {

return (v \* Math.sqrt(1 - v))

}

/\* 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.iTMU = gl.getUniformLocation(prog, 'tmu');

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

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

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

surface = new Model('Surface');

surface.BufferData(CreateSurfaceData());

LoadTexture();

surface.TextureBufferData(CreateTextureData());

sphere = new Model('Sphere');

sphere.BufferData(CreateSphereSurface(0.05))

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 }

userScaleFactor = 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);

// window.requestAnimationFrame(animation)

draw()

}

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 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;

}

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/OlenaLavro/VGGI/CGW/texture.jpg";

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) => {

switch (e.keyCode) {

case 87:

userPointCoord.y += 0.01;

break;

case 83:

userPointCoord.y -= 0.01;

break;

case 65:

userPointCoord.x -= 0.01;

break;

case 68:

userPointCoord.x += 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))

draw();

}

onmousemove = (e) => {

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

draw()

};