НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра цифрових технологій в енергетиці

**Розрахунково-графічна робота**

З дисципліни: «Візуалізація графічної та геометричної інформації»

Варіант 23

**Виконав:**

студент 5 курсу

групи ТР-23мп, ТЕФ

Таранець Антон

**Перевірив:**

Демчишин А. А.

Київ – 2023

**Завдання**

1. Накласти текстуру на поверхню отриману в результаті виконання лабораторної роботи №2.
2. Імплементувати масштабування або обертання текстури(текстурних координат) згідно з варіантом: непарні - масштабування, парні - обертання.
3. 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**

**WebGL** (скорочення від Web Graphics Library) — це API JavaScript для відтворення інтерактивної 2D і 3D графіки в будь-якому сумісному веб-браузері без використання плагінів. WebGL повністю інтегровано з іншими веб-стандартами, що дозволяє використовувати фізику, обробку зображень і ефекти з прискореним графічним процесором як частину полотна веб-сторінки. Елементи WebGL можна змішувати з іншими елементами HTML і поєднувати з іншими частинами чи фоном сторінки.

Програми WebGL складаються з керуючого коду, написаного на JavaScript, і коду шейдера, написаного мовою OpenGL ES Shading Language (GLSL ES), мовою, схожою на C або C++, і виконується на графічному процесорі (GPU) комп’ютера. WebGL розроблено та підтримується неприбутковою компанією Khronos Group.

WebGL 1.0 базується на OpenGL ES 2.0 і надає API для 3D-графіки. Він використовує елемент canvas HTML5 і доступ до нього здійснюється за допомогою інтерфейсів Document Object Model (DOM).

WebGL 2.0 базується на OpenGL ES 3.0 і забезпечує гарантовану доступність багатьох додаткових розширень WebGL 1.0 і надає нові API. Автоматичне керування пам'яттю неявно забезпечується JavaScript.

Як і OpenGL ES 2.0, WebGL не має фіксованих функцій API, представлених у OpenGL 1.0 і застарілих у OpenGL 3.0. Цю функцію, якщо вона потрібна, має реалізувати кінцевий розробник, надавши код шейдера та налаштувавши прив’язки даних у JavaScript.

Шейдери в WebGL виражаються безпосередньо в GLSL і передаються в API WebGL як текстові рядки. Реалізація WebGL компілює ці інструкції шейдерів у код GPU. Цей код виконується для кожної вершини, надісланої через API, і для кожного пікселя, растеризованого на екрані.

**Текстура** (англ. Texture mapping) — це спосіб надання поверхні 3D деталей — полігону: кольору, фактури, блиску, матовості та інших фізичних властивостей (для імітації найчастіше якогось природного матеріалу, наприклад: паперу, дерева, каменю, металу тощо). Першим цю техніку запровадив Едвін Кетмул у 1974 році.

Відображення текстур спочатку називалося дифузним відображенням. Це був методо, який просто зображав пікселі текстури на 3D-поверхні («обгортаючи» об’єкт зображенням). В останні десятиліття поява багатопрохідного рендерингу, мультитекстурування, mipmaps і більш складних відображень, таких як height mapping, bump mapping, normal mapping, displacement mapping, reflection mapping, specular mapping, occlusion mapping та багато інших варіацій зробили можливим імітацію майже фото-реалістичних об’єктів в режимі реального часу.

Досягти цього вдалось шляхом значного зменшення кількості багатокутників, а також розрахунку освітлення, необхідного для створення реалістичної та функціональної 3D-сцени.

**Карта текстури** — це зображення, нанесене на поверхню фігури або багатокутника. Це може бути растрове зображення або процедурна текстура.

Вони можуть бути одно, двох, або тривимірними. Для використання з сучасним апаратним забезпеченням дані карти текстури можуть зберігатися в розрізненому або мозаїчному порядку для покращення когерентності кешу. API візуалізації зазвичай керують ресурсами карти текстури (які можуть бути розташовані в пам’яті пристрою) як буферами або поверхнями.

Зазвичай вони містять дані кольору **RGB** (збережені як прямі кольори, стислі формати або індексовані кольори), а іноді й додатковий канал для альфа-змішування (**RGBA**). Можна використовувати альфа-канал (який може бути зручним для зберігання у форматах, аналізованих апаратним забезпеченням) для інших цілей, наприклад для відображення.

Кілька текстурних карт (або каналів) можна комбінувати для контролю дзеркальності, нормалей, зміщення або підповерхневого розсіювання, наприклад для візуалізації шкіри.

**Реалізація**

В ході другої лабораторної роботи було створено поверхню під назвою

«Surface of Conjugation of Coaxial Cylinder and Cone».

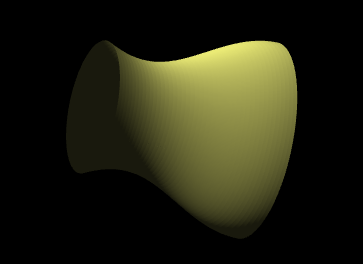


Рисунок 1 - Surface of Conjugation of Coaxial Cylinder and Cone

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 налаштовано розмір картинки так, щоб

ширина і висота були рівні.

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.

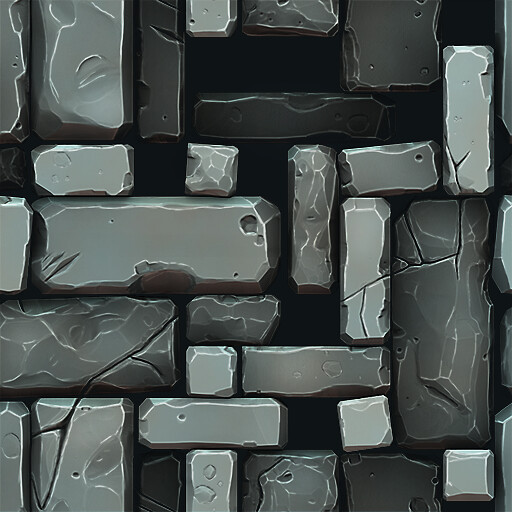


Рисунок 2 – Текстура

Поверхню з накладеною текстурою можна побачити на рисунку 3.

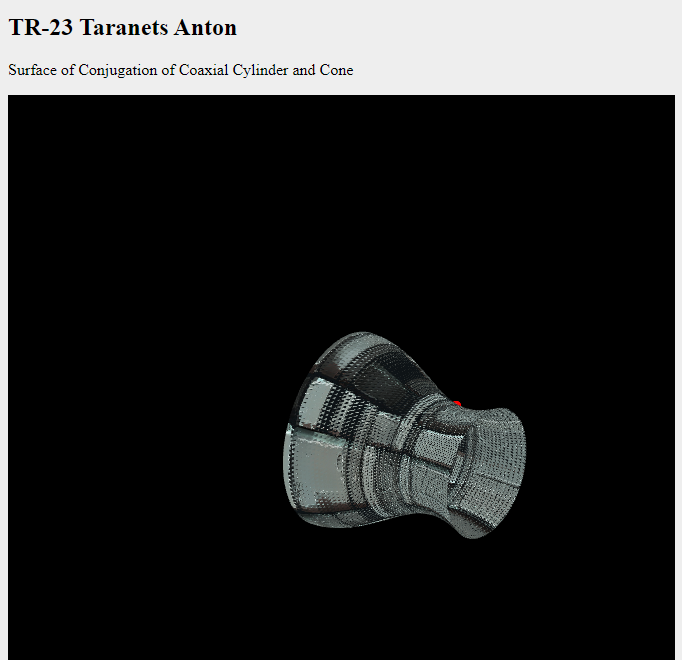


Рисунок 3 - Поверхня з накладеною текстурою

Для відображення умовної точки відносно якої буде виконуватися трансформація текстури, в класі моделі було створено відповідну функцію. Замість відображення точки було прийнято рішення відображати 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.

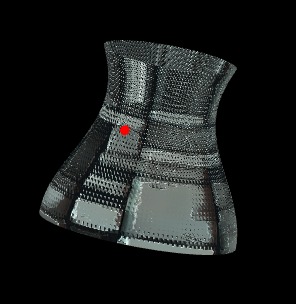


Рисунок 4 - Поверхня з умовною точкою

Для роботи з текстурою було створено ще кілька змінних в коді шейдера:

масштабування 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 масштабування текстури було додано відповідні функції на відповідні вхідні дані від користувача.

**Вказівки користувачу**

Переміщення умовної точки реалізовано за допомогою клавіш W та S здійснюють переміщення точки за параметром v в додатному та від’ємному напрямках відповідно, клавіші A та D здійснюють переміщення точки за параметром u у від’ємному та додатному напрямках відповідно.

Масштабування здійснюється за допомогою миші. Це можна спостерігати на рисунках 5, 6.

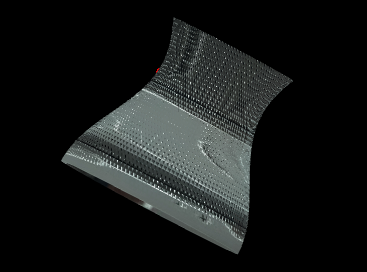


Рисунок 5 – Масштабування текстури

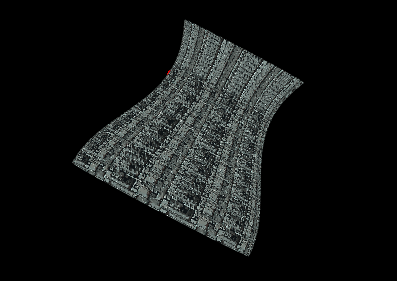


Рисунок 6 - Масштабування текстури

**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;

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.countT = 0;

    this.count = 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 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);

    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);

    gl.uniform1i(shProgram.iTMU, 0);

    gl.enable(gl.TEXTURE\_2D);

    gl.uniform1f(shProgram.iScale, userScaleFactor)

    surface.Draw();

    gl.uniform1f(shProgram.iScale, -1.0)

    let a = -0.2633257397764612;

    let c = 3.2;

    let b = 1.6099263856487789;

    let trS = conjugation(map(userPointCoord.x, 0, 1, 0, b), map(userPointCoord.y, 0, 1, 0, Math.PI \* 2), a, c)

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

    sphere.DrawPoint();

}

function CreateSurfaceData() {

    let vertexList = [];

    let i = 0;

    let j = 0;

    let a = -0.2633257397764612;

    let c = 3.2;

    let b = 1.6099263856487789;

    let step = 0.05;

    while (i < b + 0.5) {

        while (j < Math.PI \* 2) {

            let v1 = conjugation(i, j, a, c)

            let v2 = conjugation(i + step, j, a, c)

            let v3 = conjugation(i, j + step, a, c)

            let v4 = conjugation(i + step, j + step, a, c)

            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);

            j += step

        }

        j = 0;

        i += step

    }

    return vertexList;

}

function CreateTextureData() {

    let texCoordList = [];

    let i = 0;

    let j = 0;

    let a = -0.2633257397764612;

    let c = 3.2;

    let b = 1.6099263856487789;

    let step = 0.05;

    while (i < b + 0.5) {

        while (j < Math.PI \* 2) {

            let u = map(i, 0, b, 0, 1);

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

            texCoordList.push(u, v);

            u = map(i + step, 0, b, 0, 1);

            texCoordList.push(u, v);

            u = map(i, 0, b, 0, 1);

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

            texCoordList.push(u, v);

            u = map(i + step, 0, b, 0, 1);

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

            texCoordList.push(u, v);

            u = map(i + step, 0, Math.PI, 0, 1);

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

            texCoordList.push(u, v);

            u = map(i, 0, b, 0, 1);

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

            texCoordList.push(u, v);

            j += step;

        }

        j = 0

        i += step;

    }

    return texCoordList;

}

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);

            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 conjugation(z, b, a, c) {

    let r = a \* (1 - Math.cos(Math.PI \* 2 \* z / c)) + 0.8 \* 1.5;

    let x = 0.75 \* r \* Math.cos(b);

    let y = 0.75 \* r \* Math.sin(b)

    let z1 = 0.75 \* z

    return { x: x, y: y, z: z1 }

}

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);

}

/\* 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(0.04))

    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 {

        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);

    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/anton-taranec/GW/main/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.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))

    draw();

}

onmousemove = (e) => {

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

    draw()

};