Міністерство освіти і науки України

Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Кафедра інженерії програмного забезпечення в енергетиці

Графічна розрахункова робота

Варіант №11

Операції з текстурними координатами

Візуалізація графічної та геометричної інформації

**Виконав:**

студент 5-го курсу

групи ТР-32мп

Кривда Дмитро Олександрович

**Перевірив:**

Демчишин Анатолій Анатолійович

Київ – 2023

1. **Завдання**

* Нанести текстуру на поверхню з практичного завдання №2.
* Реалізувати масштабування/обертання текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання.
* Розробити функціонал для 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.

1. **Теорія**

WebGL

WebGL — це API JavaScript для відтворення інтерактивної 2D- і 3D-графіки у веб-браузері. Щоб реалізувати відображення текстур, масштабування, обертання та інтерактивне переміщення точок, WebGL API надає функції для обробки шейдерів, текстур і перетворень матриці.

* Шейдери використовуються для визначення того, як обробляються вершини та фрагменти, дозволяючи включати текстурні координати та перетворення.
* Текстурні об’єкти створюються та прив’язуються до певних текстурних одиниць для вибірки в шейдерах.
* Матричні перетворення застосовуються для керування положенням, масштабуванням і обертанням 3D-моделі.

Шейдери WebGL

У WebGL шейдери відіграють вирішальну роль у конвеєрі візуалізації, забезпечуючи гнучкий і програмований спосіб визначення вигляду 3D-графіки. У WebGL є два основних типи шейдерів: вершинні та фрагментні.

Вершинні шейдери відповідають за обробку кожної вершини 3D-моделі перед її відображенням на екрані. Вершинні шейдери беруть атрибути з даних вершин, такі як положення, колір і координати текстури, і обробляють їх для створення виходу. Одним із поширених способів використання вершинних шейдерів є застосування трансформацій до позицій вершин. Це включає переміщення, обертання та масштабування, які є вирішальними для позиціонування об’єктів у 3D-просторі. Вихідні дані вершинного шейдера містять перетворену позицію вершини та будь-які інтерпольовані значення, які будуть передані фрагментному шейдеру.

Фрагментні шейдери, які іноді називають піксельними шейдерами, працюють з кожним пікселем, який буде намальовано на екрані. Фрагментні шейдери отримують інтерпольовані значення від вершинного шейдера, наприклад координати кольору та текстури, а також інші дані, як-от інформацію про освітлення. Основним завданням фрагментного шейдера є визначення остаточного кольору кожного пікселя. Це може включати вибірку текстури, обчислення освітлення та інші ефекти. Кінцевий вихідний колір потім використовується для малювання пікселя на екрані.

І вершинний, і фрагментний шейдери написані мовою GLSL (OpenGL Shading Language) і їх потрібно скомпілювати, перш ніж їх можна буде використовувати. Після компіляції вони об’єднуються в програму шейдера. Програма шейдера повинна бути зв’язана перед відтворенням, дозволяючи WebGL використовувати вказані вершинні та фрагментні шейдери для конвеєра відтворення.

Uniforms – це значення, які залишаються постійними для всіх вершин або фрагментів під час візуалізації примітиву. Вони дозволяють передавати шейдерам зовнішні дані, такі як матриці трансформації або інформацію про глобальне освітлення.

Attributes — це дані для кожної вершини, які відрізняються між вершинами. Вони використовуються для передачі такої інформації, як положення вершин, нормалі та координати текстури.

Координати текстури зазвичай передаються як атрибути до вершинного шейдера, а потім інтерполюються через примітив для використання у фрагментному шейдері. У фрагментному шейдері координати текстури використовуються для вибірки кольорів із текстур. Вибрані кольори потім використовуються для визначення остаточного кольору кожного пікселя.

Накладання текстур. Текстурні координати

Відображення текстури — це техніка в комп’ютерній графіці, яка дозволяє відтворювати реалістичні поверхні шляхом нанесення зображень або текстур на 3D-моделі. У контексті WebGL відображення текстур передбачає асоціювання кожної вершини 3D-об’єкта з координатами текстури (u, v), які потім використовуються для вибірки кольорів із зображення текстури. Це покращує візуальний вигляд об’єкта, надаючи детальну інформацію про поверхню.

Координати текстури (u, v) є важливими параметрами, які визначають спосіб нанесення текстури на поверхню. Вони варіюються від 0 до 1 і використовуються для звернення до конкретних точок на зображенні текстури. У контексті WebGL координати текстури пов’язані з кожною вершиною 3D-моделі та інтерполюються по всій поверхні під час візуалізації. Це гарантує, що текстура точно охоплює модель.

1. **Деталі розробки**

За варіантом мені було надано Continuous Topographic Surface of Cassini. При виконанні практичного завдання №2 було розроблено програму, що виводить поверхню у вигляді суцільних трикутників.

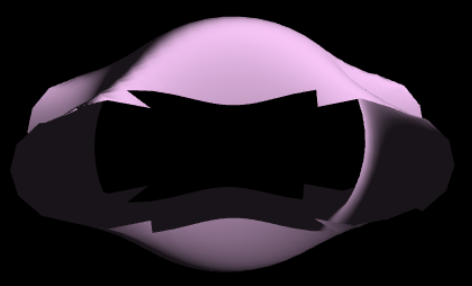


Рисунок 1 - Вигляд поверхні

Я обрав наступне зображення для подальшого виконання розрахунково-графічної роботи. Для підтримки на більшій кількості операційних систем та браузерів було обрано зображення розміром 512х512 пікселів. Формат зображення png.



Рисунок 2 - Зображення текстури

Було накладено текстуру на поверхню. Для накладання текстури на поверхню було підготовлено буфер текстурних координат, кожна з яких відповідає елементу масива з буферу вершин. Згідно варіанту текстура має масштабуватися, тому було створено відповідний uniform, який визначатиме фактор маштабування. В програмі шейдера для визначення кольору пікселя фігури на яку накладено текстуру необхідно використовувати функцію texture2D(). Ця функція першим аргументом приймає обʼєкт класу sampler2D, який зберігає дані про зображення, а другим текстурну координату.



Рисунок 3 - Текстура накладена на поверхню

Було створено новий об’єкт класу Model для відображення точки відносно якої буде здійснюватися трансформація текстури. Об’єкт графічно відображається як сфера. Сфера перебуває на поверхні.



Рисунок 4 – Зображення поверхні з точкою

Згідно варіанту було імплементовано масштабування текстури відносно точки на поверхні.

1. **Інструкції користувача**

Фігуру можна обертати відносно центру затиснувши ліву клавішу миші та потягнувши в сторону бажаного обертання.

Рисунок 5 - Вигляд фігури до та після обертання

Переміщувати точку відносно якої здійснюється обертання можна за допомогою клавіш WASD. Кожне натискання переміщує точку по поверхні на визначений крок. Переміщення здійснюється до визначеної межі.

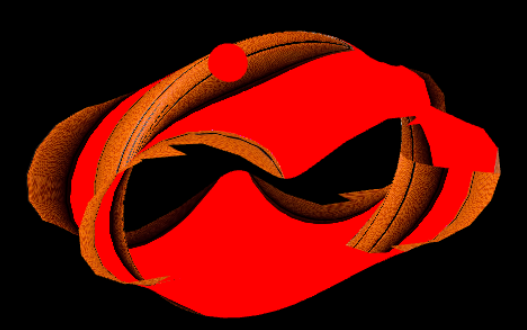
 

Рисунок 6 - Демонстрація переміщення точки відносно нерухомої фігури

За допомогою слайдера з підписом «scale factor ” можна змінювати фактор масштабування відносно умовної точки на поверхні. Якщо значення фактору масштабування не дорівнює нулю, то можна помітити, що при переміщенні точки зміщується і текстура, адже масштабування відбувається відносно іншої точки на поверхні , яка в свою чергу відповідає іншій текстурній координаті.

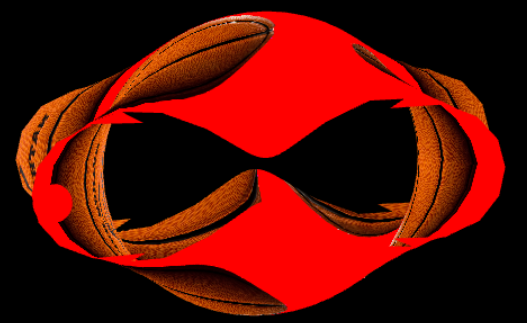
 

Рисунок 7 - Демонстрація зміни фактору масштабування текстури

При перезавантаженні сторінки обертання фігури, а також значення фактору масштабування буде встановлено по замовчуванню. Положення точки відносно поверхні також буде скинуто до значення по замовчуванню.

1. **Код програми**

Основний лістинг програми на javascript

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);

    const projVal = 17;

    let projection = m4.orthographic(-projVal, projVal, -projVal, projVal, -projVal, projVal);

    /\* 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, -5);

    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);

    const normalMat = m4.identity();

    m4.inverse(modelView, normalMat);

    m4.transpose(normalMat, normalMat);

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

    gl.uniformMatrix4fv(shProgram.iNormalMat, false, normalMat);

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

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

    gl.uniform1f(shProgram.iScale, document.getElementById('scale').value);

    surface.Draw();

    gl.uniform1f(shProgram.iScale, 0.0);

    console.log(document.getElementById('scale').value)

    gl.uniform2f(shProgram.iPointPos, map(pointPos[0], 0, PI \* 2, 0, 1), map(pointPos[1], -3, 3, 0, 1))

    gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection, m4.translation(...Object.values(cassiniVertex(...pointPos)))));

    point.Draw();

}

function animation() {

    draw()

    window.requestAnimationFrame(animation)

}

const { cos, sin, sqrt, pow, PI } = Math

function CreateSurfaceData() {

    let vertexList = [];

    const NUM\_STEPS\_U = 50,

        NUM\_STEPS\_Z = 50,

        MAX\_U = PI \* 2,

        MAX\_Z = 3,

        STEP\_U = MAX\_U / NUM\_STEPS\_U,

        STEP\_Z = 2 \* MAX\_Z / NUM\_STEPS\_Z

    for (let u = 0; u < MAX\_U; u += STEP\_U) {

        for (let z = -3; z < MAX\_Z; z += STEP\_Z) {

            let vertex = cassiniVertex(u, z)

            vertexList.push(...vertex)

            vertex = cassiniVertex(u + STEP\_U, z)

            vertexList.push(...vertex)

            vertex = cassiniVertex(u, z + STEP\_Z)

            vertexList.push(...vertex)

            vertexList.push(...vertex)

            vertex = cassiniVertex(u + STEP\_U, z)

            vertexList.push(...vertex)

            vertex = cassiniVertex(u + STEP\_U, z + STEP\_Z)

            vertexList.push(...vertex)

        }

    }

    return vertexList;

}

function r(u, z) {

    return sqrt(c(z) \* c(z) \* cos(2 \* u) + sqrt(pow(a, 4) - pow(c(z), 4) \* pow(sin(2 \* u), 2)))

}

function c(z) {

    return 3 \* z;

}

const a = 8

const scaler = 1;

function cassiniVertex(u, z) {

    // console.log(r(u, z))

    let x = r(u, z) \* cos(u),

        y = r(u, z) \* sin(u),

        cZ = z;

    return [scaler \* x, scaler \* y, scaler \* cZ];

}

function CreateNormals() {

    let normalList = [];

    const NUM\_STEPS\_U = 50,

        NUM\_STEPS\_Z = 50,

        MAX\_U = PI \* 2,

        MAX\_Z = 3,

        STEP\_U = MAX\_U / NUM\_STEPS\_U,

        STEP\_Z = 2 \* MAX\_Z / NUM\_STEPS\_Z

    for (let u = 0; u < MAX\_U; u += STEP\_U) {

        for (let z = -3; z < MAX\_Z; z += STEP\_Z) {

            let vertex = normalAnalytic(u, z)

            normalList.push(...vertex)

            vertex = normalAnalytic(u + STEP\_U, z)

            normalList.push(...vertex)

            vertex = normalAnalytic(u, z + STEP\_Z)

            normalList.push(...vertex)

            normalList.push(...vertex)

            vertex = normalAnalytic(u + STEP\_U, z)

            normalList.push(...vertex)

            vertex = normalAnalytic(u + STEP\_U, z + STEP\_Z)

            normalList.push(...vertex)

        }

    }

    return normalList;

}

const e = 0.0001

function normalAnalytic(u, z) {

    let u1 = cassiniVertex(u, z),

        u2 = cassiniVertex(u + e, z),

        z1 = cassiniVertex(u, z),

        z2 = cassiniVertex(u, z + e);

    const dU = [], dZ = []

    for (let i = 0; i < 3; i++) {

        dU.push((u1[i] - u2[i]) / e)

        dZ.push((z1[i] - z2[i]) / e)

    }

    const n = m4.normalize(m4.cross(dU, dZ))

    return n

}

function CreateTextures() {

    let textureList = [];

    const NUM\_STEPS\_U = 50,

        NUM\_STEPS\_Z = 50;

    for (let u = 0; u < NUM\_STEPS\_U; u++) {

        for (let z = 0; z < NUM\_STEPS\_Z; z++) {

            textureList.push(u / NUM\_STEPS\_U, z / NUM\_STEPS\_Z)

            textureList.push((u + 1) / NUM\_STEPS\_U, z / NUM\_STEPS\_Z)

            textureList.push(u / NUM\_STEPS\_U, (z + 1) / NUM\_STEPS\_Z)

            textureList.push(u / NUM\_STEPS\_U, (z + 1) / NUM\_STEPS\_Z)

            textureList.push((u + 1) / NUM\_STEPS\_U, z / NUM\_STEPS\_Z)

            textureList.push((u + 1) / NUM\_STEPS\_U, (z + 1) / NUM\_STEPS\_Z)

        }

    }

    return textureList;

}

function map(value, a, b, c, d) {

    value = (value - a) / (b - a);

    return c + value \* (d - c);

}

function CreateSphereData() {

    let vertexList = [];

    let u = 0,

        t = 0;

    while (u < Math.PI \* 2) {

        while (t < Math.PI) {

            let v = getSphereVertex(u, t);

            let w = getSphereVertex(u + 0.1, t);

            let wv = getSphereVertex(u, t + 0.1);

            let ww = getSphereVertex(u + 0.1, t + 0.1);

            vertexList.push(...v);

            vertexList.push(...w);

            vertexList.push(...wv);

            vertexList.push(...wv);

            vertexList.push(...w);

            vertexList.push(...ww);

            t += 0.1;

        }

        t = 0;

        u += 0.1;

    }

    return vertexList;

}

function getSphereVertex(long, lat) {

    return [

        Math.cos(long) \* Math.sin(lat),

        Math.sin(long) \* Math.sin(lat),

        Math.cos(lat)

    ]

}

/\* 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, "texture");

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

    shProgram.iNormalMat = gl.getUniformLocation(prog, "normalMat");

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

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

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

    shProgram.iPointPos = gl.getUniformLocation(prog, 'pointPos');

    LoadTexture();

    surface = new Model('Surface');

    // console.log(CreateSurfaceData().length)

    // console.log(CreateNormals().length)

    // console.log(CreateTextures().length)

    surface.BufferData(CreateSurfaceData(), CreateNormals(), CreateTextures());

    point = new Model('Point');

    point.BufferData(CreateSphereData(), CreateSphereData(), CreateSphereData())

    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() {

    document.getElementById("speed").onchange = () => {

        v = parseFloat(document.getElementById("speed").value)

    }

    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);

    animation();

}

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/Dmitry200285/Vgg/CGW/Projection.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()

    }

}

let v = 0.15

window.onkeydown = (e) => {

    // console.log(e.keyCode)

    if (e.keyCode == 87) { //w

        pointPos[0] = Math.min(pointPos[0] + v, Math.PI \* 2);

    }

    else if (e.keyCode == 65) { //a

        pointPos[1] = Math.max(pointPos[1] - v, -2.5);

    }

    else if (e.keyCode == 83) { //s

        pointPos[0] = Math.max(pointPos[0] - v, 0);

    }

    else if (e.keyCode == 68) { //d

        pointPos[1] = Math.min(pointPos[1] + v, 2.5);

    }

    draw();

}

1. **Код програми**

Лістинг файлу 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>Continuous Topographic Surface of Cassini</h2>

    <h2> with "Trackball" Mouse Rotation</h2>

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

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

    <p>WASD для переміщення червоної точки відносно якої відбувається масштабування текстури</p>

    <label for="speed">Speed of point translation</label>

    <input type="range" name="speed" id="speed" min="0.01" max="0.3" step="0.01" value="0.15">

    <label for="scale">Scale Factor</label>

    <input type="range" name="scale" id="scale" min="0.1" max="3" step="0.1" onchange="draw()">

    <div id="canvas-holder">

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

    </div>

</body>

</html>