Міністерство освіти і науки України

Національний технічний університет України «КПІ ім. Ігоря Сікорського»

Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота

з курсу “Візуалізація графічної та геометричної інформації”

на тему “Операції з текстурними координатами”

**Виконав:**

студент 5-го курсу

групи ТР-32мп

Черкас Олександр Євгенійович

**Перевірив:**

Демчишин А. А.

Київ – 2023

**Завдання:**

* Нанесіть текстуру на поверхню з практичного завдання №2.
* Реалізація масштабування текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання
* Повинна бути можливість переміщати точку вздовж простору поверхні *(u,v)* за допомогою клавіатури. Наприклад, клавіші **A** і **D** переміщують точку вздовж параметра *u , клавіші* **W** і **S** переміщують точку вздовж параметра *v.*

**Теоритична частина:**

*WebGL* (Web Graphics Library) - це JavaScript API для взаємодії з графікою та 3D-графікою у веб-браузерах. Це стандарт, розроблений для створення візуально насичених та інтерактивних веб-сцен та додавання тривимірної графіки до веб-сайтів без використання плагінів. WebGL базується на OpenGL ES, стандарті для вбудованих систем, що використовуються в основному для мобільних пристроїв.

Основні риси WebGL включають в себе можливість використання апаратного прискорення графіки для відтворення тривимірної графіки у реальному часі, створення складних візуальних ефектів, таких як тіні та відбиття, та взаємодія з користувачем через мишу, клавіатуру та інші пристрої введення. WebGL широко використовується в інтернет-галузі для створення вражаючих веб-інтерфейсів та віртуальної реальності.

Шейдери у WebGL — це програми, написані мовою GLSL (OpenGL Shading Language), які використовуються для визначення обчислень, пов'язаних з обробкою графічних об'єктів у тривимірному просторі. У контексті WebGL шейдери відповідають за визначення положення вершин об'єктів, їх кольору, а також обчислення світлових ефектів та інших графічних атрибутів.

Шейдери розділяються на два типи: вершинні та фрагментні. Вершинні шейдери відповідають за обробку вершин об'єктів, визначаючи їхнє положення в просторі. Фрагментні шейдери визначають колір кожного пікселя в зображенні, а також виконують обчислення світлових та інших ефектів.

Вони становлять ключовий елемент програмної частини WebGL, дозволяючи здійснювати гнучкий та реалістичний відображення графічних об'єктів у веб-браузері.

Вершинні шейдери виконують обчислення для кожної вершини об'єкта у тривимірному просторі. Основні завдання вершинного шейдера включають в себе трансформацію вершин з їхнього локального простору в глобальний, визначення їхнього кольору та інших атрибутів. Ці обчислення включають у себе матричні операції для обрання точного положення вершин у просторі, а також врахування камери та освітлення.

Вони можуть також використовуватися для передачі даних між вершинами, таких як нормалі чи текстурні координати. Ці шейдери генерують вихідні дані, які використовуються фрагментними шейдерами для обчислення кольору.

Фрагментні шейдери відповідають за обчислення кольору для кожного пікселя у зображенні. Вони працюють на рівні фрагментів, які генеруються після растеризації трикутників вершинним шейдером. Основна мета фрагментного шейдера - визначити кінцевий колір для кожного пікселя.

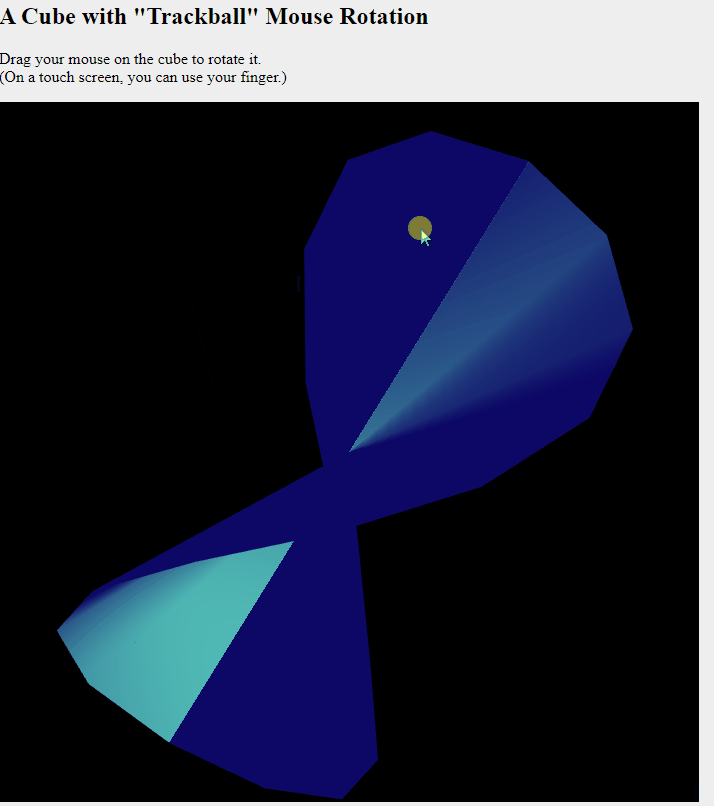
Фрагментні шейдери отримують вхідні дані від вершинного шейдера, такі як координати вершин, нормалі, текстурні координати та інші атрибути. Вони також можуть використовувати освітлювання, тіньові та текстурні обчислення для створення фінального кольору пікселя.

Одна з основних задач фрагментного шейдера - визначення впливу світла та тіньових ефектів на конкретний піксель. Крім того, вони дозволяють реалізовувати різноманітні візуальні ефекти, такі як зміна кольору, текстурні ефекти та використання різних матеріалів для моделювання поверхні об'єктів.

**Впровадження:**

Варіант – 29: Conical Surface with a Directrix Curve on a Sphere.

На момент виконання практичного завдання №2 робота виглядала наступним чином:



**Рисунок 1: Conical Surface with a Directrix Curve on a Sphere**

В якості текстури було обрано наступне зображення (пізніше буде пояснено цей вибір):



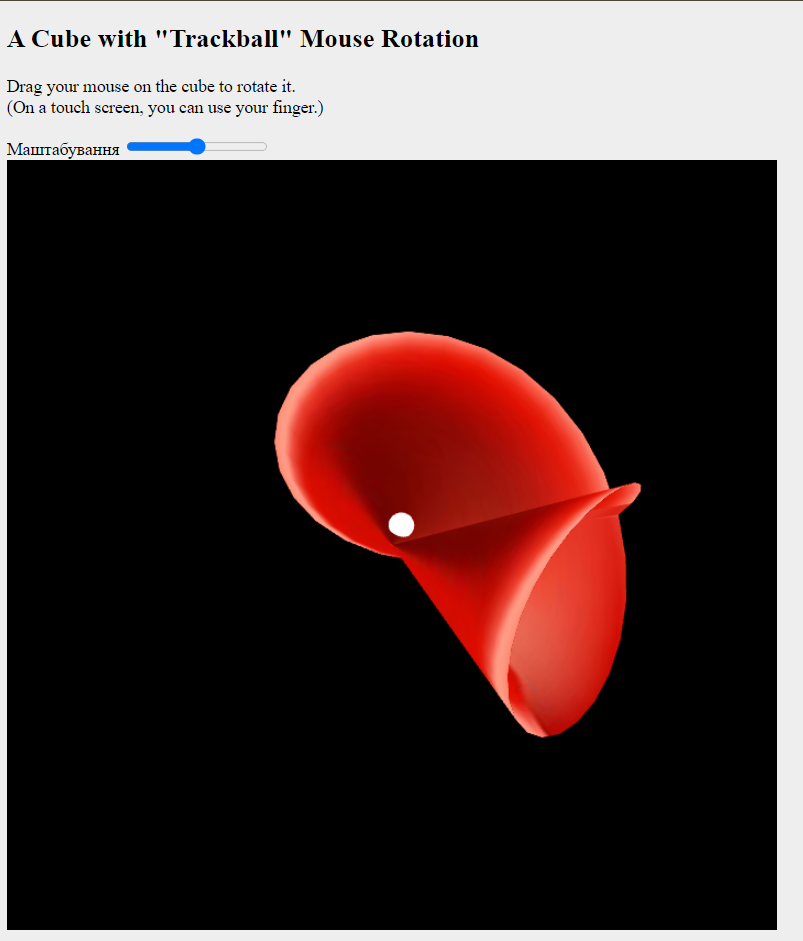
**Рисунок 2: Обрана текстура**

Було підготовлено буфер координат текстури та накладено її на поверхню, був створений параметр для визначення фактору масштабування, оскільки так потрібно за завданням для непарних варіантів.

У програмі шейдера для визначення кольору пікселя фігури, на яку застосовано текстуру, використано функцію texture2D(). Ця функція приймає об'єкт класу sampler2D як перший аргумент, який містить дані про зображення, та другий аргумент - текстурну координату.

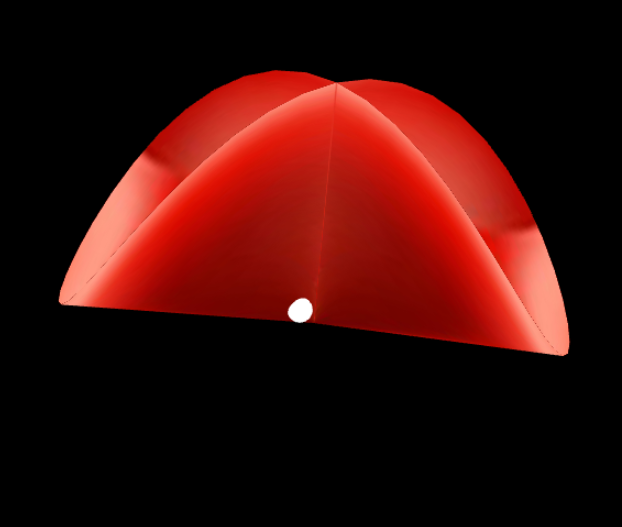
Графічне відображення фігури представлено у якості сфери на поверхні.

Результат наступний:



**Рисунок 3: Загальний вигляд роботи**

Також згідно завдання на поверхню з текстурою була накладена точка.



**Рисунок 4: Точка на поверхні**

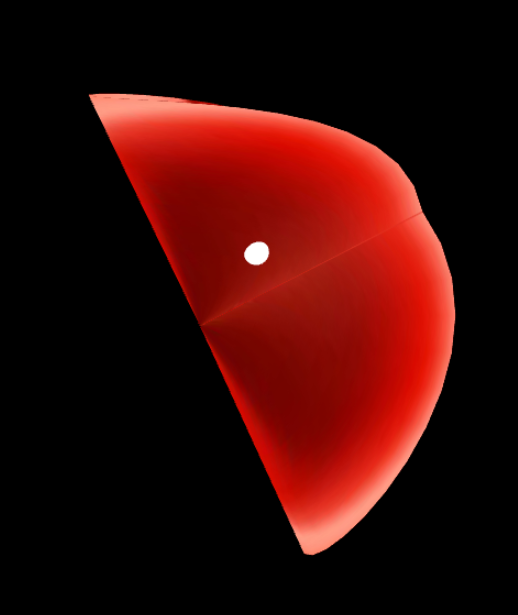
**Інструкція користувача**

Шляхом затиснення ЛКМ та руху миші фігуру можливо обертати навколо центру координат у бажаний бік. При «горизонтальному» положенні фігура приймає вигляд банту, що і було причиною вибору саме такої текстури.



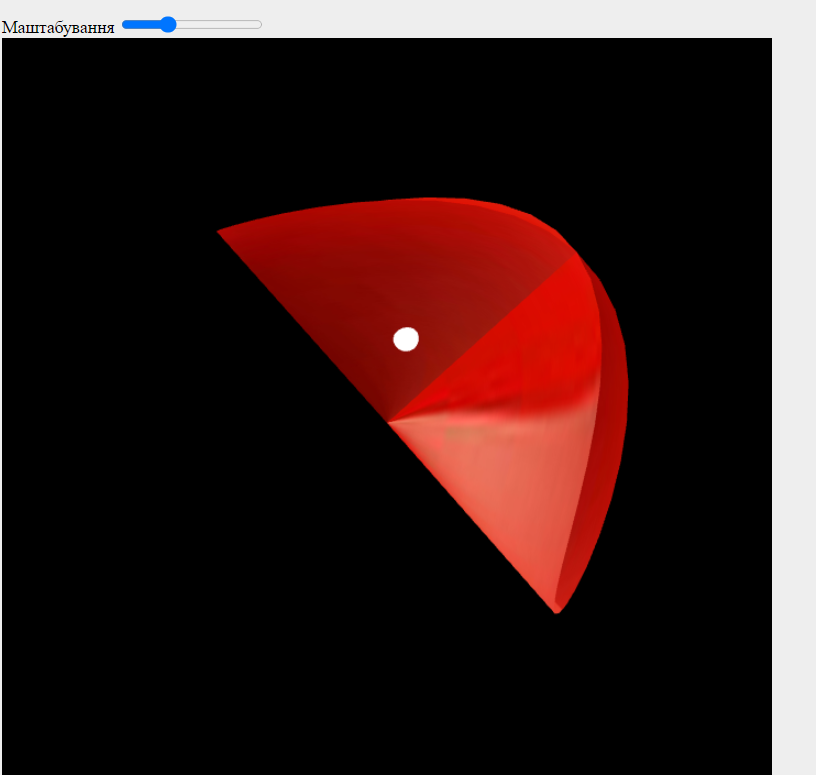
**Рисунок 5: Обертання до «форми» банту**

Для руху точки відносно фігури використовуються клавіші W, S, A, D. Одне натиснення переміщує точку на 1 крок по поверхні фігури та працює по всій площині фігури.

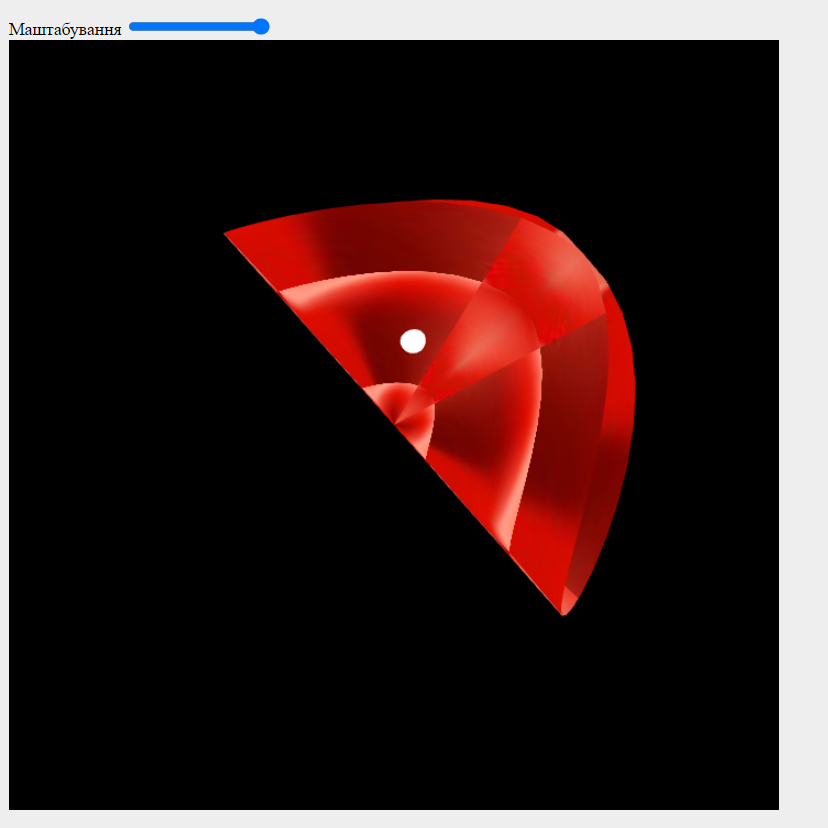


**Рисунок 6: Переміщення точки**

Через повзунок «Масштабування» можливо відповідно масштабувати текстуру відносно нинішнього положення точки. За замовчуванням значення знаходиться на середині повзунка, дозволяючи масштабування у обидва напрями.



**Рисунок 7: Масштабування**



**Рисунок 8: Масштабування**

**Код програми**

Функція CreateSurfaceData(),модифікована після практичного заняття №2:

function CreateSurfaceData() {

    //Побудова власне фігури

    let a = 2;

    let p = 1;

    let normalsList =[];

    let vertexList = [];

    let vertexTexCoordList = [];

    let numSteps = 50; // Кількість кроків

    // Значення параметра u

    const uMin = -Math.PI;

    const uMax = Math.PI;

    // Значення параметра v

    const vMin = -2;

    const vMax = 0;

    let delta = 0.0001;

    for (let i = 0; i < numSteps; i++) {

        const u1 = uMin + (uMax - uMin) \* (i / numSteps);

        const u2 = uMin + (uMax - uMin) \* ((i + 1) / numSteps);

        for (let j = 0; j < numSteps; j++) {

            const v1 = vMin + (vMax - vMin) \* (j / numSteps);

            const v2 = vMin + (vMax - vMin) \* ((j + 1) / numSteps);

            let ω1 = p \* u1;

            let [x1, y1, z1] = creating(a, ω1, u1, v1);

            let derivative1 = getDerivative1(a, ω1, u1, v1, delta);

            let derivative2 = getDerivative2(a, ω1, u1, v1, delta);

            let normal1 = m4.cross(derivative1,derivative2);

            let ω2 = p \* u2;

            let [x2, y2, z2] = creating(a, ω2, u2, v1);

            derivative1 = getDerivative1(a, ω2, u2, v1, delta);

            derivative2 = getDerivative2(a, ω2, u2, v1, delta);

            let normal2 = m4.cross(derivative1,derivative2);

            ω1 = p \* u1;

            let [x3, y3, z3] = creating(a, ω1, u1, v2);

            derivative1 = getDerivative1(a, ω1, u1, v2, delta);

            derivative2 = getDerivative2(a, ω1, u1, v2, delta);

            let normal3 = m4.cross(derivative1,derivative2);

            ω2 = p \* u2;

            let [x4, y4, z4] = creating(a, ω2, u2, v2);

            derivative1 = getDerivative1(a, ω2, u2, v2, delta);

            derivative2 = getDerivative2(a, ω2, u2, v2, delta);

            let normal4 = m4.cross(derivative1,derivative2);

            vertexList.push(x1, y1, z1, x2, y2, z2, x3, y3, z3, x3, y3, z3, x2, y2, z2, x4, y4, z4);

            normalsList.push(normal1[0],normal1[1],normal1[2], normal2[0],normal2[1],normal2[2],normal3[0],

                normal3[1],normal3[2],normal3[0],normal3[1],normal3[2], normal2[0],normal2[1],normal2[2], normal4[0],normal4[1],normal4[2]);

            vertexTexCoordList.push(map(u1, -Math.PI, Math.PI, 0, 1), map(v1, -a, 0, 0, 1));

            vertexTexCoordList.push(map(u2, -Math.PI, Math.PI, 0, 1), map(v1, -a, 0, 0, 1));

            vertexTexCoordList.push(map(u1, -Math.PI, Math.PI, 0, 1), map(v2, -a, 0, 0, 1));

            vertexTexCoordList.push(map(u1, -Math.PI, Math.PI, 0, 1), map(v2, -a, 0, 0, 1));

            vertexTexCoordList.push(map(u2, -Math.PI, Math.PI, 0, 1), map(v1, -a, 0, 0, 1));

            vertexTexCoordList.push(map(u2, -Math.PI, Math.PI, 0, 1), map(v2, -a, 0, 0, 1));

        }

    }

    return [vertexList, normalsList, vertexTexCoordList];

}

Функція main() вершинного шейдера:

void main() {

    mat4 t1 = translation(-translate);

    mat4 t2 = translation(translate);

    mat4 s1 = scaling(scale);

    vec4 tr1 = t1\*vec4(texture,0.,1.);

    vec4 sc1 = s1\*tr1;

    vec4 tr2 = t2\*sc1;

    textureVarying = tr2.xy;

    vertexVarying = mat3(ModelViewProjectionMatrix) \* vertex;

    normalVarying = mat3(NormalMatrix)\*normal;

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

}

Функція main() фрагментного шейдера:

void main() {

    vec3 N = normalize(normalVarying);

    vec3 L = normalize(vec3(1.0));

    float lambertian = max(dot(N, L), 0.0);

    vec3 R = reflect(-L, N);

    vec3 V = normalize(-vertexVarying);

    float specAngle = max(dot(R, vertexVarying), 0.0);

    float specular = pow(specAngle, 20.0);

    vec3 ambient = color.rgb\*0.1;

    vec3 diffuse = color.rgb;

    vec3 c = vec3(ambient +

                        lambertian \* diffuse +

                        specular \* vec3(1.0));

    gl\_FragColor = vec4(c,1.0);

    vec4 tC = texture2D(tmu, textureVarying);

    gl\_FragColor = tC;

    if(scale<-1.0){

        gl\_FragColor = vec4(1.0);

    }

}