Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інституту атомної та теплової енергетики

Кафедра ЦТЕ

**Візуалізація графічної та геометричної інформації**

Розрахунково-графічна робота

Варіант №20

Виконав: студент гр. ТР-32мп

Петричук В. В.

Перевірив:

Демчишин А. А.

Київ – 2023

**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.

**Теоретичні відомості**

1. *Візуалізація графіки та WebGL*

Процес представлення даних у візуальному форматі для аналізу та розуміння відомий як графічна візуалізація. WebGL, API, побудований на OpenGL, пропонує надійне рішення для створення веб-графіки за допомогою JavaScript. Це дозволяє маніпулювати графічними елементами, які демонструють тривимірні об’єкти або сцени на екрані користувача. Шейдери відіграють вирішальну роль у визначенні того, як обробляються вершини та фрагменти, дозволяючи включати координати текстури та перетворення.

1. *Роль шейдерів у WebGL*

Шейдери служать фундаментальним аспектом графічного програмування в WebGL, що виконується на графічному процесорі (GPU). Вони дають можливість програмістам контролювати обробку графічних об'єктів. WebGL використовує два типи шейдерів: вершинні шейдери та фрагментні шейдери.

Вершинні шейдери працюють з кожною вершиною графічного об’єкта, головним чином обробляючи завдання обробки вершин, такі як зміна позиції, зміна властивостей (наприклад, координат кольору або текстури) і агрегування вершин для створення графічних примітивів, таких як трикутники або лінії.

Фрагментні шейдери, з іншого боку, визначають кольори та властивості кожного пікселя, який потрібно відобразити на екрані. Вони використовують вхідні дані від вершинного шейдера та інші параметри, такі як текстури, освітлення та тіні.

Шейдери в WebGL програмуються за допомогою GLSL (OpenGL Shading Language), спеціальної мови для шейдерів. Він забезпечує обмежену функціональність для виконання обчислень на GPU. Процес виконання шейдера в WebGL складається з кількох етапів:

1) Створення та компіляція шейдерів: Написання та компіляція вершинних і фрагментних шейдерів і зв’язування їх у програму.

2) Передача даних до шейдерів: передача вхідних даних, таких як координати вершин, координати текстури, кольори, матриці перетворення та інші параметри, від програми до шейдерів.

3) Виконання шейдерів: Вершинні шейдери виконуються для кожної вершини, тоді як фрагментні шейдери виконуються для кожного пікселя.

4) Відтворення: після виконання шейдерів графічне зображення відображається на екрані на основі обчислень, виконаних шейдерами.

5) Шейдери пропонують точний контроль над відтворенням графіки, уможливлюючи створення складних і реалістичних візуальних ефектів у веб-середовищі за допомогою WebGL.

1. *Нанесення текстур на поверхні*

У WebGL текстури використовуються для накладання зображень на поверхні об’єктів, покращуючи такі візуальні ефекти, як текстурування, деталізація та реалістичність. Щоб застосувати текстури, потрібні координати текстури (u, v). Ці координати відповідають точкам текстури та визначають спосіб відображення зображення на відповідній поверхні.

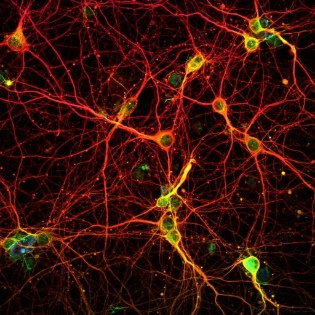
Процес передбачає нанесення текстур або візуальних візерунків на поверхню 3D-об’єкта. Координати текстури, часто представлені як пари значень u і v, призначаються кожній точці на поверхні об’єкта. Ці координати визначають конкретну точку на текстурі, яка буде зіставлена з цією конкретною точкою поверхні.

Щоб застосувати текстуру до поверхні в WebGL, необхідно завантажити текстуру в пам’ять GPU та налаштувати такі параметри, як фільтри згладжування, режими повторення текстури, анізотропія тощо. Потім координати текстури зв’язуються з вершинами об’єкта, визначаючи, скільки текстури відображається в кожній вершині.

**Деталі реалізації**

Для виконання розрахунково-графічної роботи потрібно побудувати поверхню з лабораторної роботи №2, за 20 варіантом (Surface of Revolution with Damping Circular Waves).

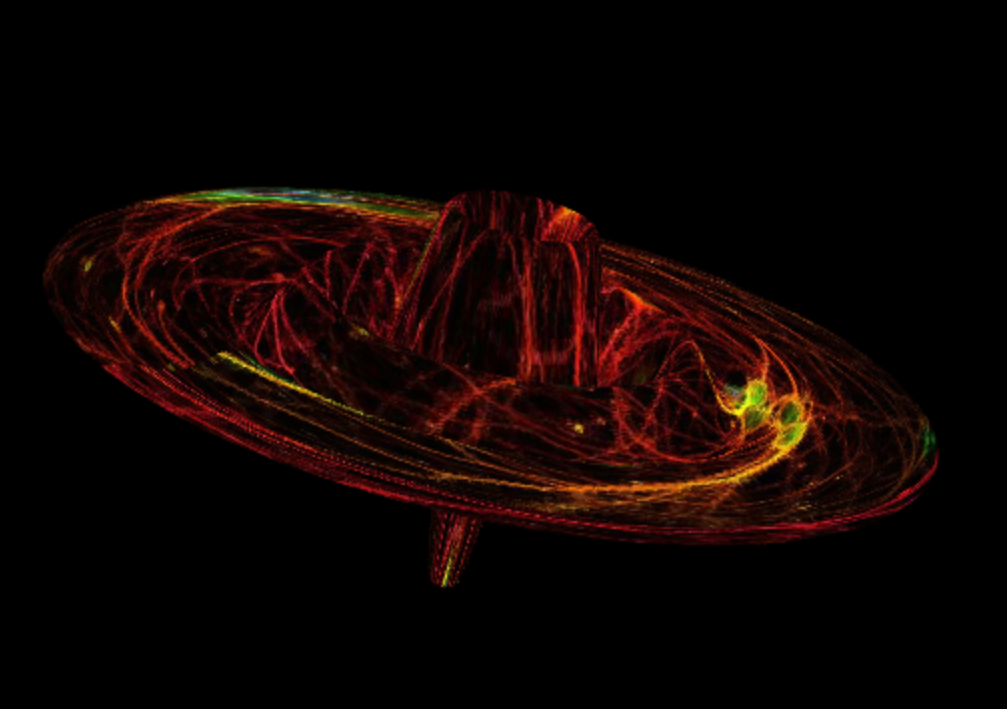
Далі потрібно нанести довільну текстуру на поверхню. (рис.1)



*Рисунок 1. Використана текстура для нанесення на поверхню*

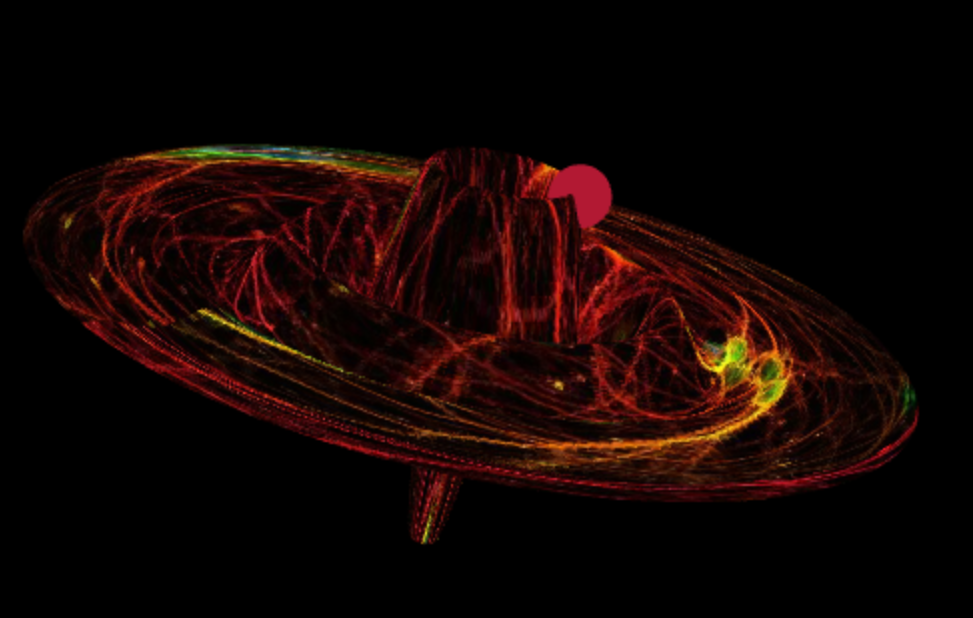
Щоб застосувати текстуру, потрібно переписати шейдер і основну програму, вказати в шейдері посилання на об'єкт текстури і координати текстури. Також для обертання текстури потрібно в шейдері ввести змінну uniform float u\_textureRotation, яка використовувалася для обертання текстури навколо точки, відповідно до опції.

Для роботи з шейдером довелося переписати в моделі метод Draw, в ньому потрібно вказати координати поверхні, що передаються за поворотом об'єкта в систему координат, також потрібно вказати змінні для малювання текстури, як-от значення поточного обертання та необхідні примітиви. Також необхідно було створити функцію TextureBufferData, яка використовується для введення координат текстури в програму. (Рис. 2)



*Рисунок 2. Поверхня з текстурою*

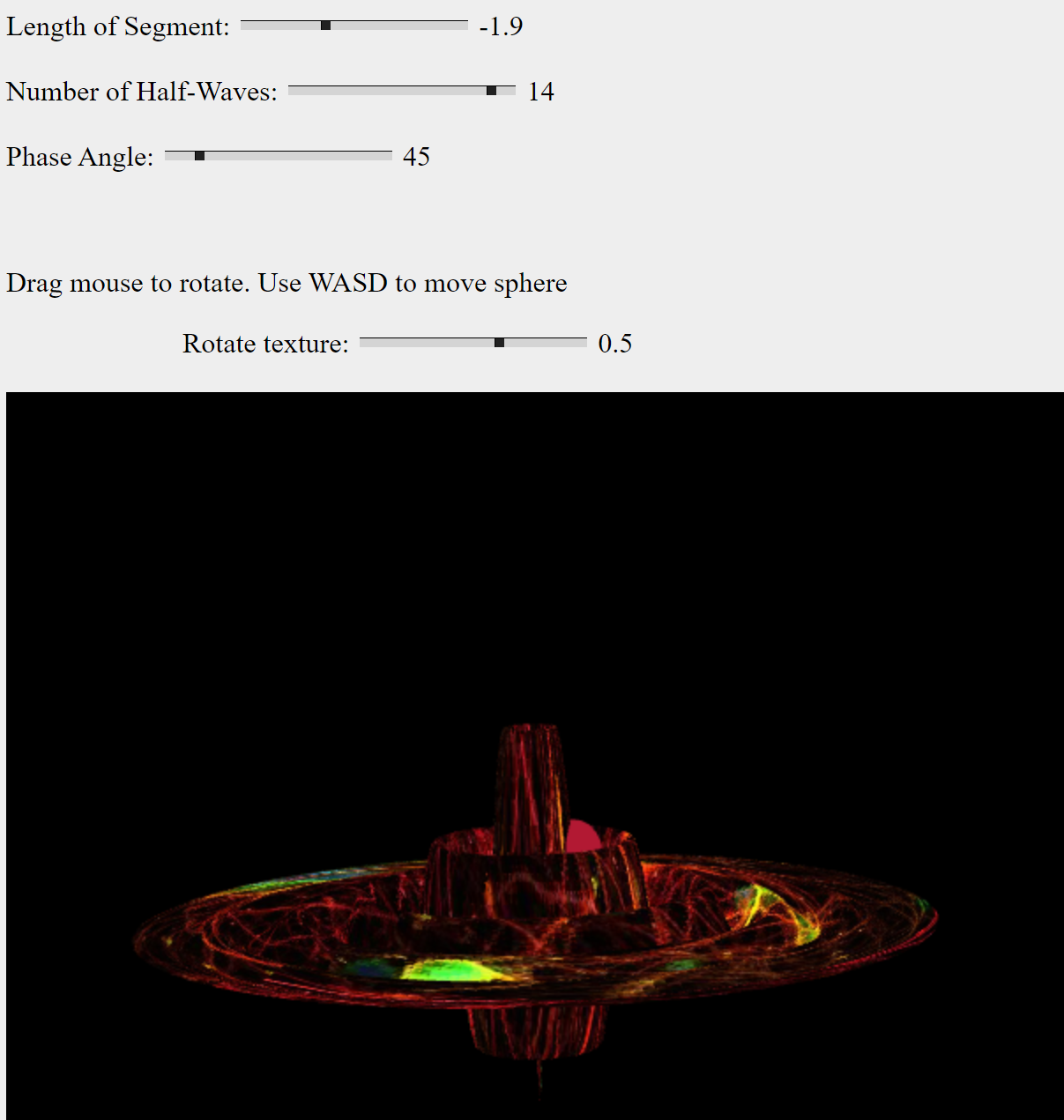
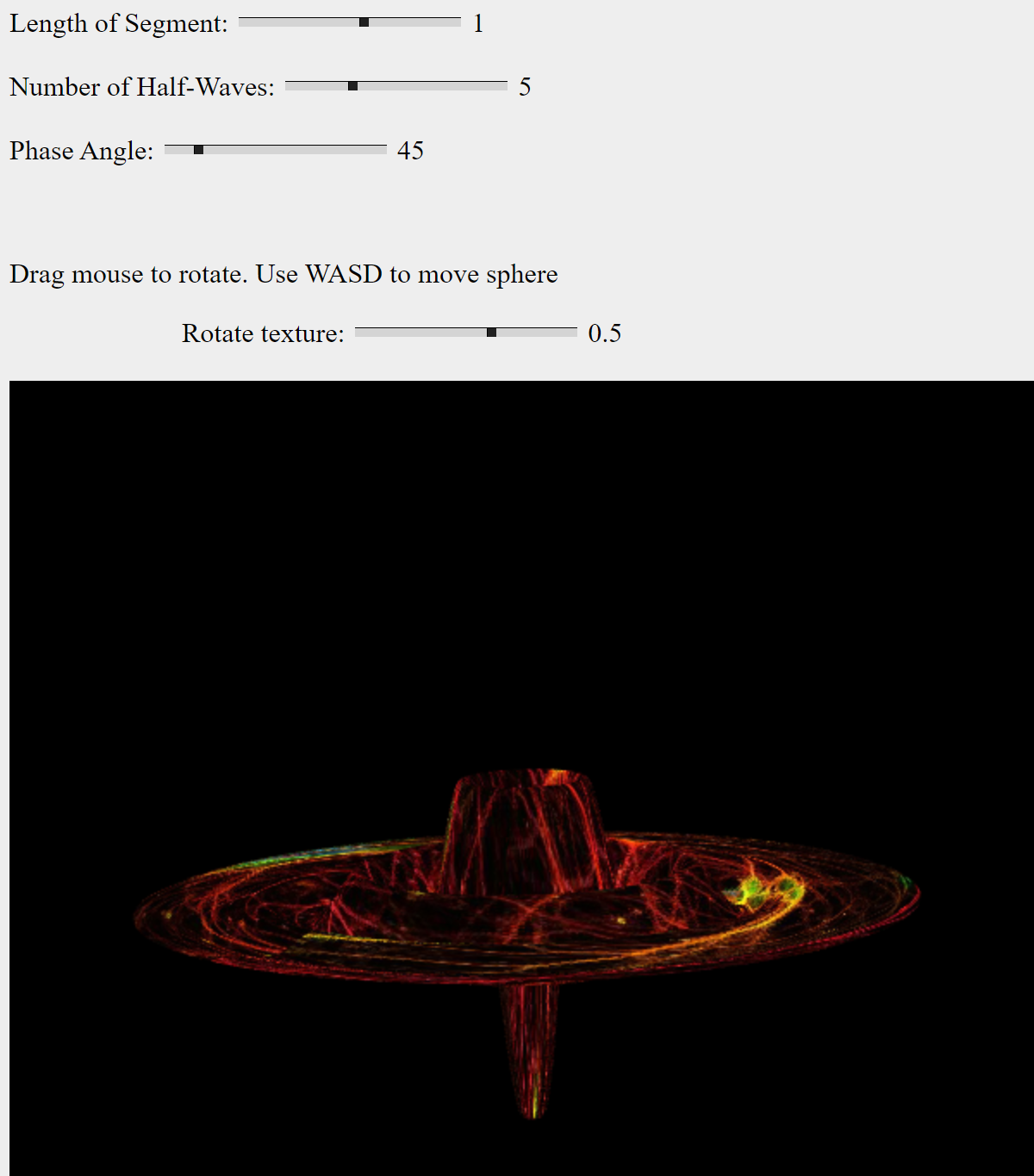
Кінцева мета передбачала створення функції, яка дозволяє точці рухатися по поверхні. Для цього був побудований об’єкт у формі сфери. Шляхом регулювання значень v (за допомогою клавіш W/S) і u (за допомогою клавіш A/D) ці значення використовувалися для обчислення поточної точки на поверхні сфери. Потім було використано рівняння поверхні для визначення відповідних координат X, Y і Z, де буде розташована сфера (рис. 3).



*Рисунок 3. Сфера на поверхні*

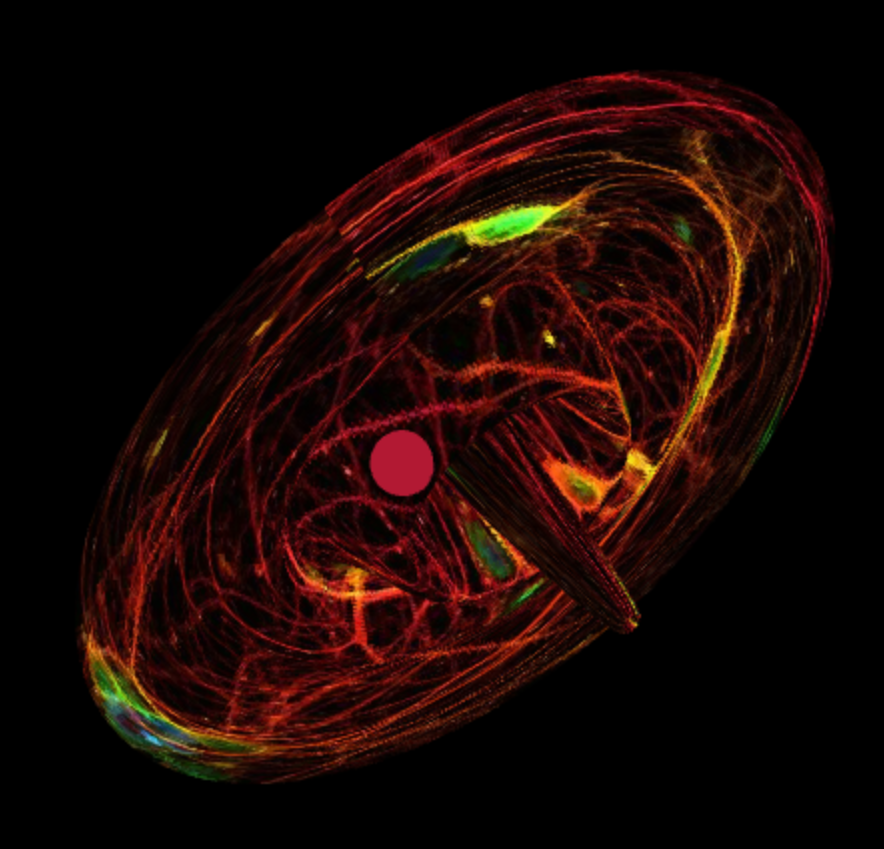
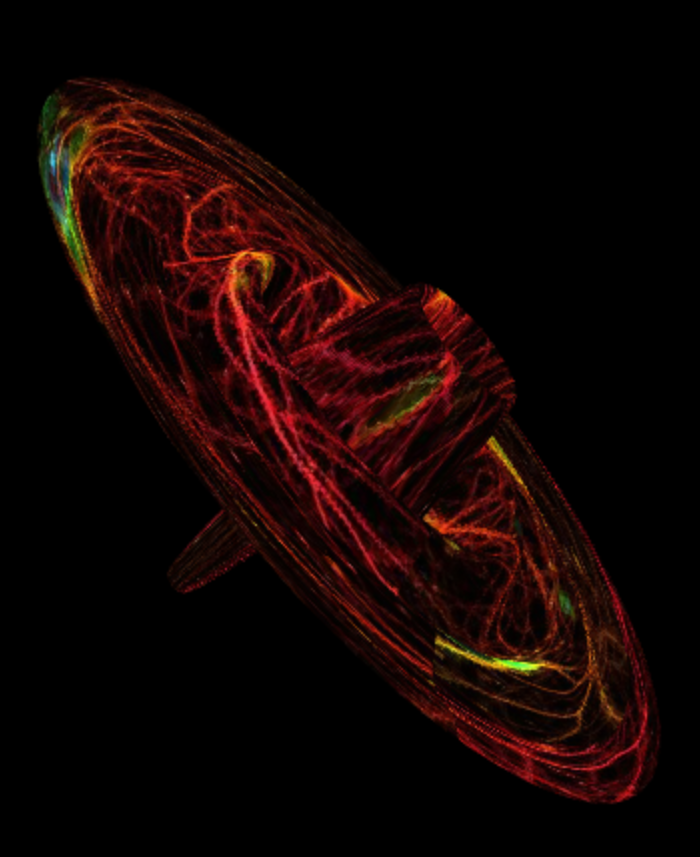
**Інструкція користувача**

Параметри поверхні можна змінювати, корегуючи значення над зображенням поверхні. (рис. 4)



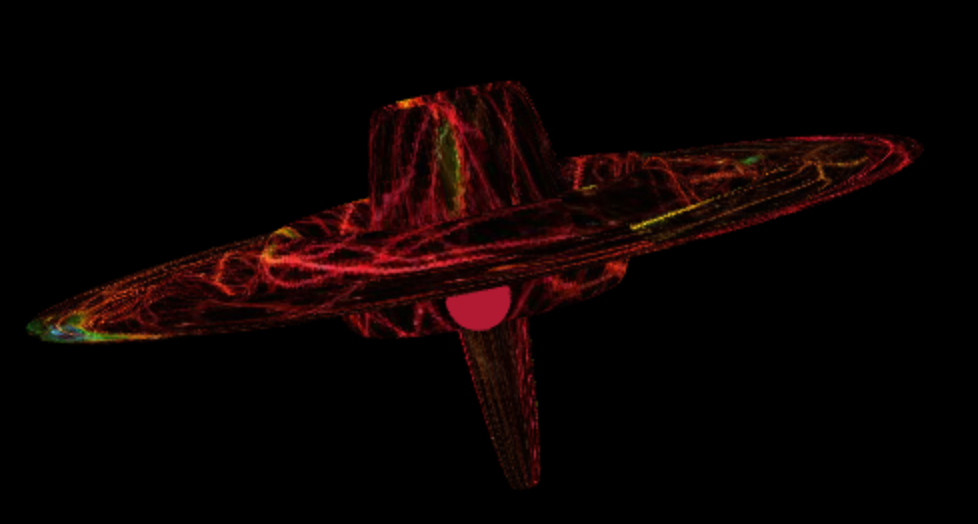
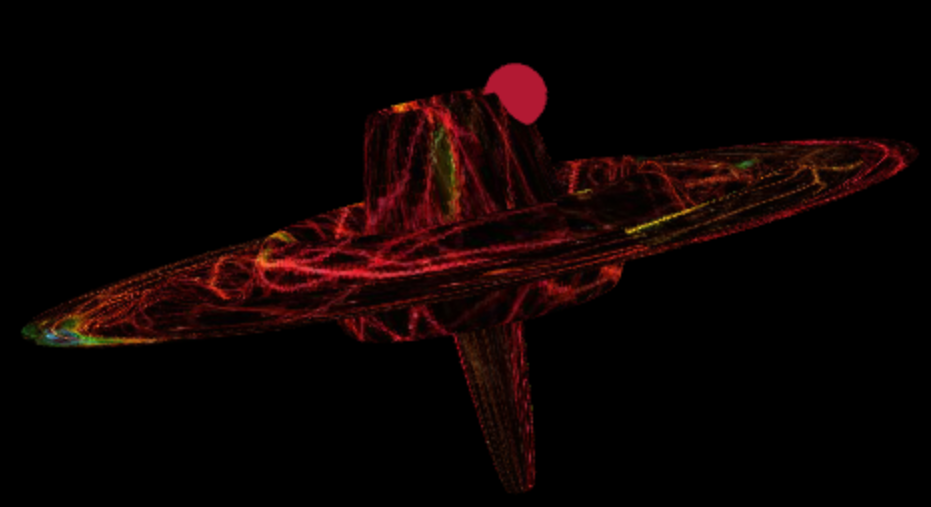
*Рисунок 4. Зміна параметрів поверхні*

Також поверхню можна обертати, використовуючи курсор миші, якщо затиснути ЛКМ та повертати курсор в потрібну сторону. (рис. 5)

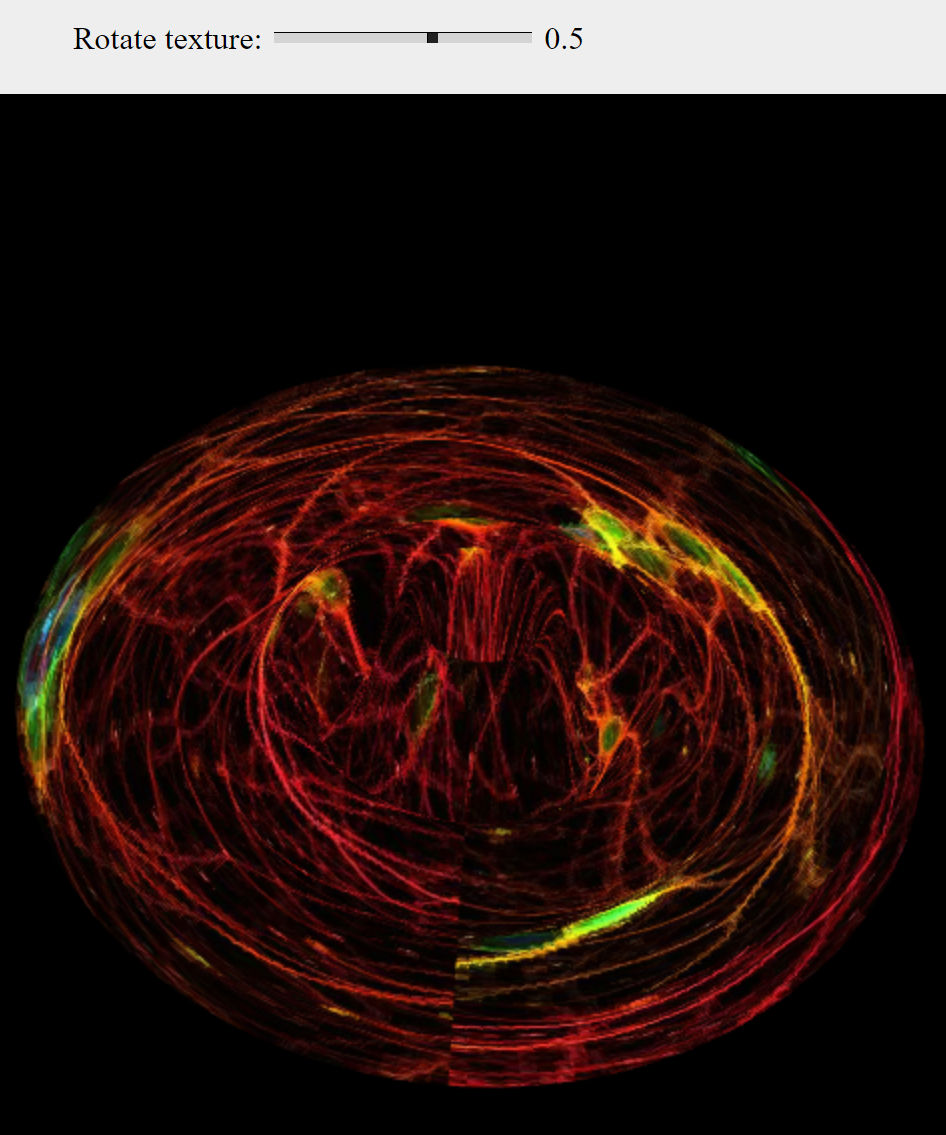
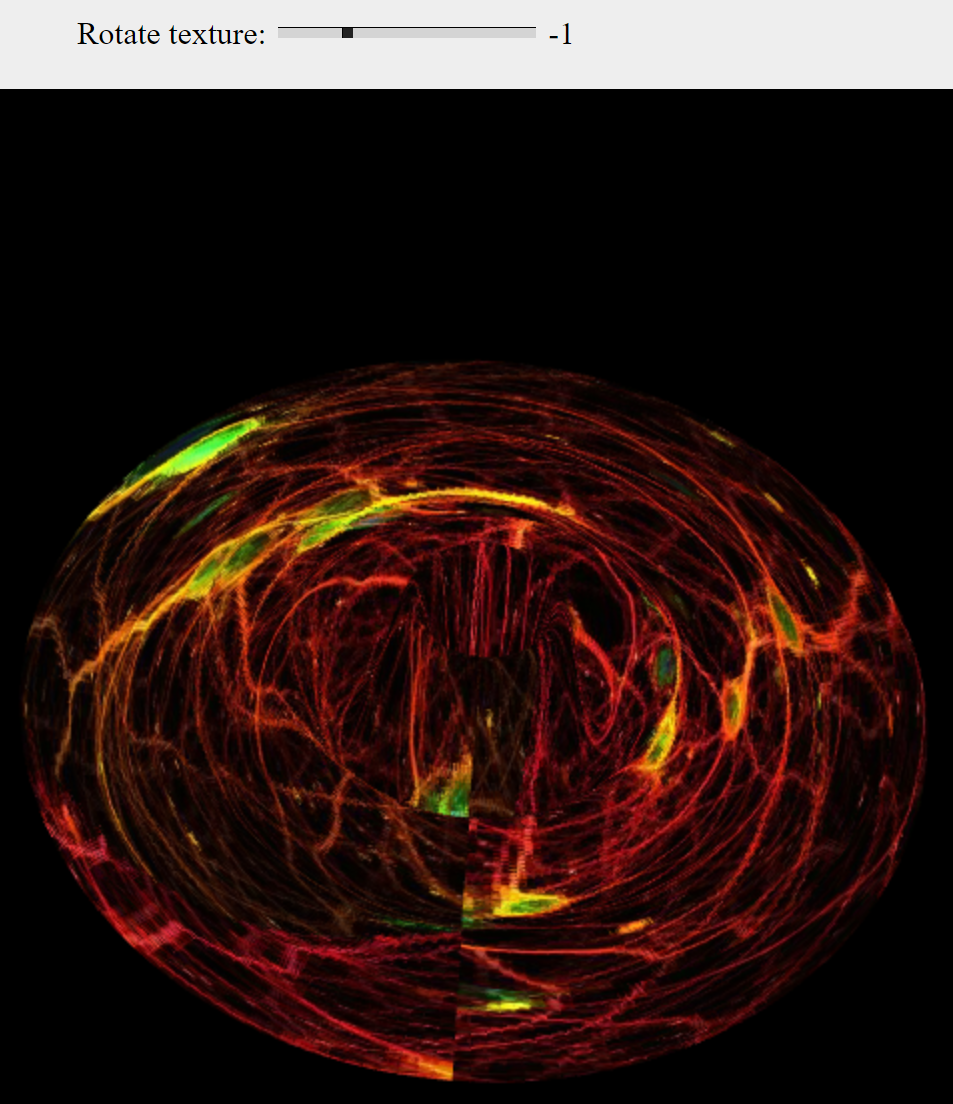
*Рисунок 5. Оберт поверхні використовуючи ліву кнопку миші*

Для переміщення сфери по поверхні необхідно змінювати координати v (використовуючи клавіші W/S) та координати u (використовуючи клавіші клавіші A/D). (рис. 6)

*Рисунок 6. Рух сфери по поверхні*

Також можна обертати текстуру за допомогою зміни значення Rotate texture відповідно до значення оберту, яка потім буде використано для обрахування нових координат текстури (рис. 7)

*Рисунок 7. Приклад оберту текстури*

**Лістинг програмного коду**

1. Шейдер вершин для поверхні

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 a\_texcoord;

uniform mat4 ModelViewProjectionMatrix;

varying vec2 v\_texCoord;

void main() {

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

    v\_texCoord = a\_texcoord;

}`;

1. Фрагментний шейдер для поверхні

const fragmentShaderSource = `

precision mediump float;

varying vec2 v\_texCoord;

uniform sampler2D u\_texture;

uniform float u\_textureRotation;

void main() {

    vec2 rotatedTexCoord = vec2(

        (v\_texCoord.x - 0.5) \* cos(u\_textureRotation) - (v\_texCoord.y - 0.5) \* sin(u\_textureRotation) + 0.5,

        (v\_texCoord.x - 0.5) \* sin(u\_textureRotation) + (v\_texCoord.y - 0.5) \* cos(u\_textureRotation) + 0.5

    );

    vec4 texColor = texture2D(u\_texture, rotatedTexCoord);

    gl\_FragColor = texColor;

};

1. Конструктор програми шейдера в main.js

function ShaderProgram(name, program) {

    this.name = name;

    this.prog = program;

    this.iSolidColor = -1;

    this.iAttribVertex = -1;

    this.iModelViewProjectionMatrix = -1;

    this.uTextureRotation = -1;

    this.uTexture = -1;

    this.aTexCoord = -1;

    this.Use = function() {

        gl.useProgram(this.prog);

    }

}

1. Конструктор об’єкта моделі поверхні

function Model(name) {

    this.name = name;

    this.iVertexBuffer = gl.createBuffer();

    this.iTextureBuffer = gl.createBuffer();

    this.texture = gl.createTexture();

    this.countText = 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 (normals) {

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

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

        this.countText = normals.length / 3;

    }

    this.Draw = function(projectionViewMatrix) {

        let rotation = spaceball.getViewMatrix();

        let translation = m4.translation(0, 0, 0);

        let modelMatrix = m4.multiply(translation, rotation);

        let modelViewProjection = m4.multiply(projectionViewMatrix, modelMatrix);

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

        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.aTexCoord, 2, gl.FLOAT, false, 0, 0);

        gl.enableVertexAttribArray(shProgram.aTexCoord);

        gl.uniform1i(shProgram.uTexture, 0);

        gl.uniform1f(shProgram.uTextureRotation, rotate\_texture\_value);

        gl.enable(gl.TEXTURE\_2D);

        gl.uniform1f(shProgram.iRotateValue, 0);

        gl.drawArrays(gl.TRIANGLES, 0, this.count);

    }

}