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-31мп

Павський Д. О.

*(П.І.Б.)*

Перевірив: Демчишин А. А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(П.І.Б., підпис)*

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. Графічна візуалізація та WebGL

Графічна візуалізація - це процес відображення даних у візуальній формі з метою аналізу та розуміння. WebGL, як API на основі OpenGL, є потужним інструментом для створення веб-графіки з використанням мови програмування JavaScript. Він забезпечує можливість маніпулювати графічними елементами, які відображають тривимірний об'єкт або сцену на екрані користувача. Для визначення того, як обробляються вершини та фрагменти використовуються шейдери, дозволяючи включати текстурні координати та перетворення.

2. Шейдери в WebGL

Шейдери у WebGL є основною складовою графічного програмування, які виконуються на графічному процесорі (GPU). Вони дозволяють програмістам контролювати процес обробки графічних об'єктів. У WebGL використовуються два типи шейдерів: вершинні (vertex shaders) та фрагментні (fragment shaders).

Вершинний шейдер (Vertex Shaders) виконується для кожної вершини графічного об'єкта. Його основна функція - обробка вершин, надання їм нового місцезнаходження, зміни їх позиції та властивостей (наприклад, кольору чи текстурних координат). Після обробки всі вершини збираються для створення графічних примітивів, таких як трикутники чи лінії.

Фрагментний шейдер (Fragment Shaders) відповідає за кожен піксель, який потрібно рендерити на екрані. Цей шейдер визначає кольори та властивості кожного пікселя, використовуючи вхідні дані, що надходять з вершинного шейдера, та інші параметри, такі як текстури, освітлення, тіні тощо.

У WebGL шейдери програмуються мовою GLSL (OpenGL Shading Language), спеціалізованою мовою програмування для шейдерів. Це мова з обмеженим функціоналом, яка дозволяє виконувати обчислення на графічному процесорі. Процес виконання шейдерів у WebGL включає кілька етапів:

1. Створення та компіляція шейдерів: Написання вершинних та фрагментних шейдерів, їх компіляція та приєднання до програми.
2. Передача даних у шейдери: Передача вхідних даних, таких як координати вершин, текстурні координати, кольори, матриці трансформації та інші параметри, з програми у шейдери.
3. Виконання шейдерів: Вершинні шейдери виконуються для кожної вершини, а фрагментні - для кожного пікселя.
4. Рендеринг: Коли всі шейдери виконані, графічна картинка рендериться на екрані згідно результатів обчислень, виконаних шейдерами.

Шейдери дають можливість контролювати кожну деталь графічної візуалізації, дозволяючи створювати складні та реалістичні візуальні ефекти у веб-середовищі за допомогою WebGL.

3. Нанесення текстур на поверхні

Текстури у WebGL використовуються для накладання зображень на поверхні об'єктів. Їхнє використання вимагає текстурних координат (u, v), які відповідають точкам текстури і використовуються для відображення зображення на відповідній поверхні.

Нанесення текстур дозволяє створювати візуальні ефекти, такі як текстурування, деталізацію та реалістичність об'єктів. У WebGL текстури використовуються для накладання зображень або візуальних шаблонів на поверхню тривимірного об'єкта. Ключовим елементом при роботі з текстурами є текстурні координати (також відомі як UV-координати). Кожна точка на поверхні об'єкта відповідає певним текстурним координатам (зазвичай у вигляді пари значень u та v), що вказують на конкретну точку на текстурі, яка буде відображена на цій точці поверхні.

Для нанесення текстури на поверхню у WebGL, спочатку потрібно завантажити текстуру до пам'яті GPU та налаштувати її параметри, такі як фільтри для згладжування зображення, режими повторення текстури, анізотропність тощо. Після цього текстурні координати пов'язуються з вершинами об'єкта, щоб визначити, яка частина текстури відображатиметься на кожній вершині.

**Деталі реалізації**

Для початку виконання розрахунково-графічної роботи потрібно було побудувати поверхню з лабораторної роботи №2, за 20 варіантом це - Surface of Revolution with Damping Circular Waves. Поверхню потрібно було вивести трикутниками.

Далі потрібно нанести довільну текстуру на поверхню. Для цього потрібно завантажити в програму .jpeg/.png картинку вказавши URL посилання на неї. (рис.1)

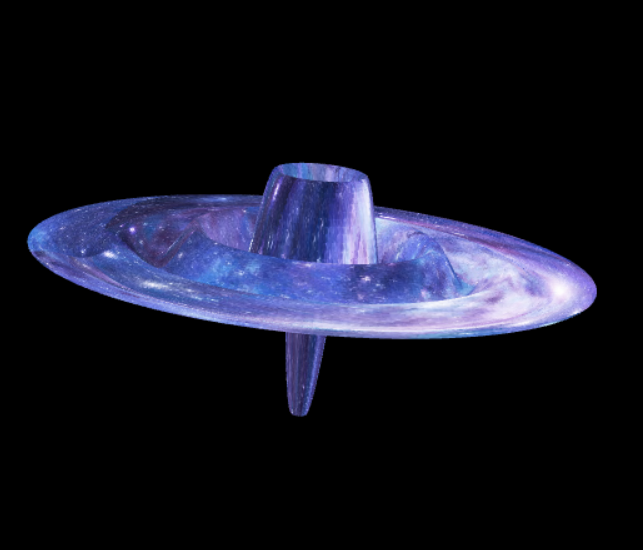
Рисунок 1. Текстура, яка використовувалась для нанесення на поверхню



Для того, щоб нанести текстуру потрібно переписати шейдер та основну програму, у шейдері вказати посилання на текстурний об’єкт та координати текстури. Також для обертання текстури потрібно занести у шейдер змінну uniform float u\_textureRotation, яка використовувалась для обертання текстури навколо точки, згідно варіанту та точку навколо якої робиться оберт u\_rotationPoint.

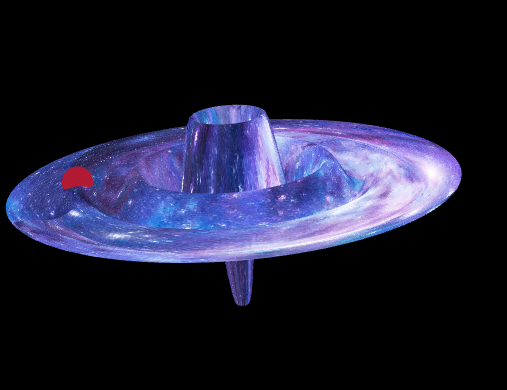
Для роботи з шейдером потрібно було переписати Draw метод в моделі, в ньому необхідно вказати координати поверхні, перенесені згідно оберту об’єкта в системи координат, також потрібно вказати змінні для нанесення текстури, такі як значення поточного оберту та необхідні примітиви. Також необхідно було створити функцію TextureBufferData, яка використовується для внесення координат текстури у програму. (рис. 2)

Рисунок 2. Поверхня з нанесеною текстурою



Останнім завдання потрібно було розробити функціонал для переміщення точки вздовж простору поверхні. Для цього було побудовано новий об’єкт у виді сфери. При зміні значення v(клавішами W/S) та u(клавішами A/D), ці значення передавались до обчислення поточної точки де знаходиться сфера та згідно рівняння поверхні вираховувалось значення у X,Y,Z координатах куди і переміщалась сфера та зміщувалась текстура. (рис. 3)

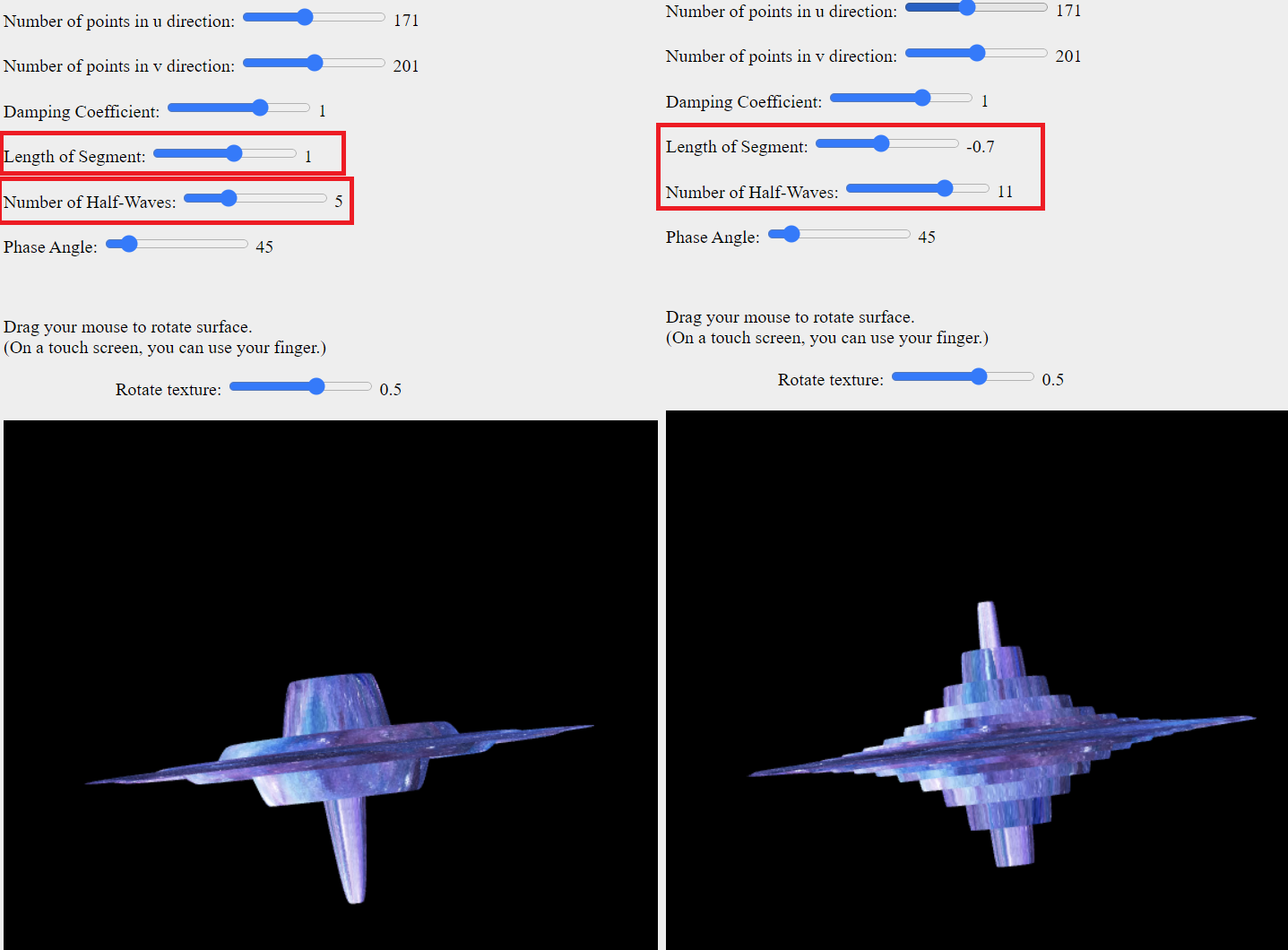
Рисунок 3. Сфера на поверхні



**Інструкція користувача**

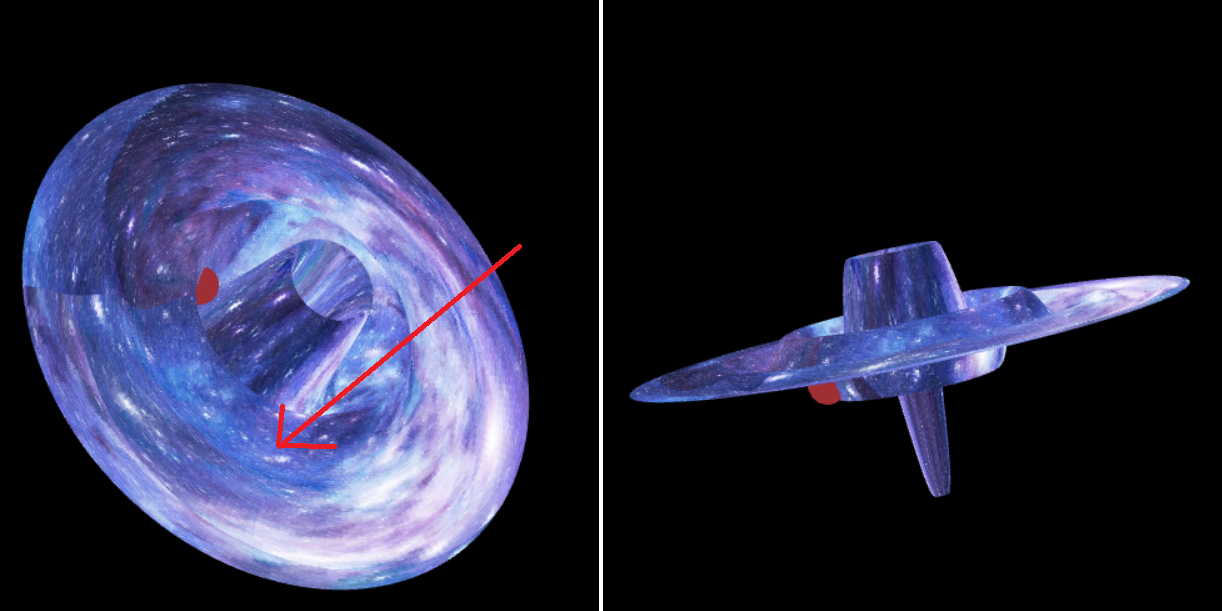
Параметри поверхні можна змінювати, корегуючи відповідні значення над зображенням поверхні. (рис. 4)

Рисунок 4. Зміна параметрів поверхні



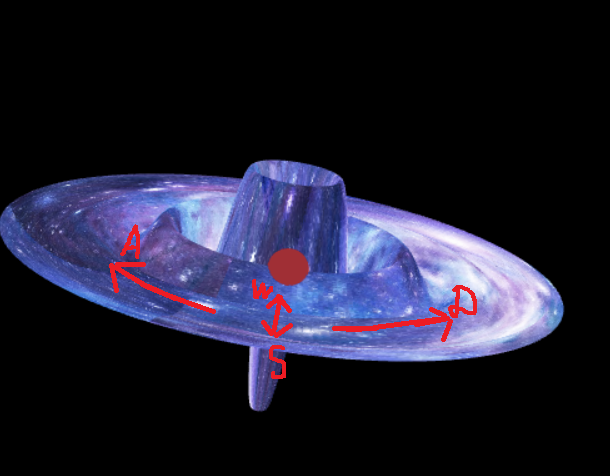
Також поверхню можна обертати, затиснувши ліві кнопку миші та переміщаючи курсор у потрібну сторону. (рис. 5)

Рисунок 5. Оберт поверхні використовуючи ліву кнопку миші



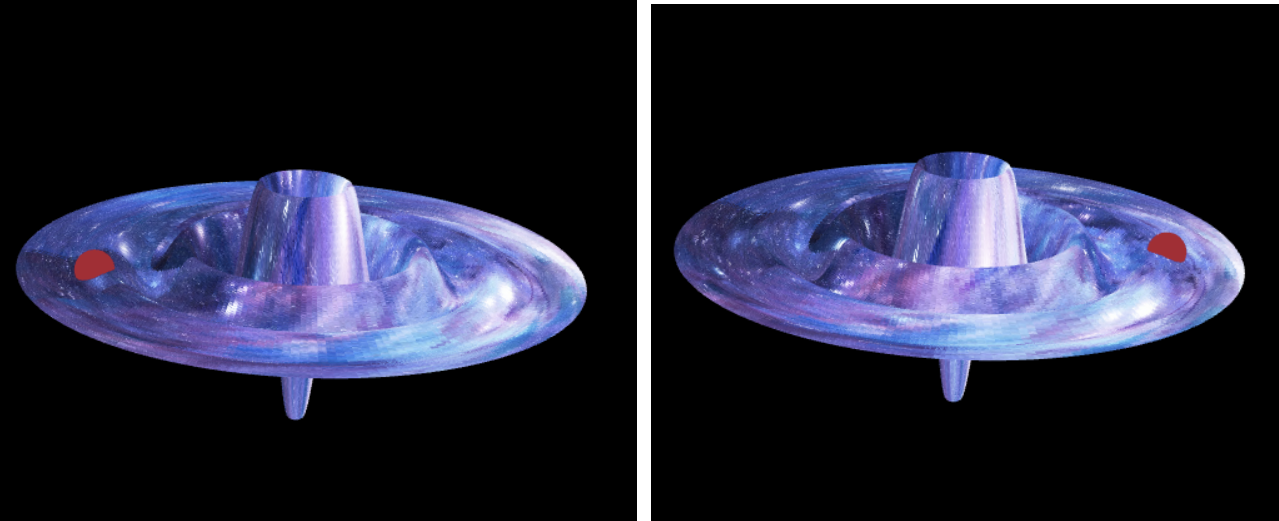
Для переміщення сфери по поверхні необхідно використовувати клавіші W/S для зміни координати v та клавіші A/D для зміни координати u. (рис. 6)

Рисунок 6. Інструкція для руху сфера по поверхні



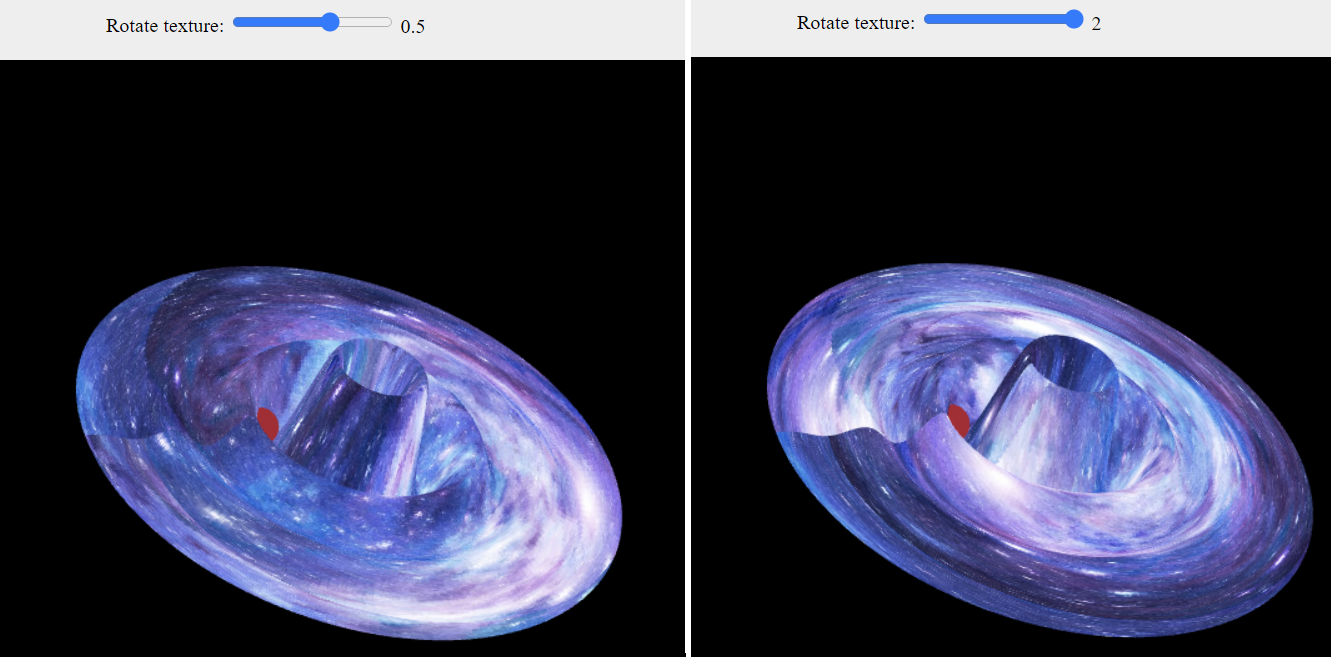
Наприклад, можемо натиснути клавішу D для зменшення значення u та після цього S для збільшення v і отримаємо наступний результат. (рис. 7)

Рисунок 7. Приклад переміщення сфери по поверхні



Також за допомогою зміни значення Rotate texture, можна обертати текстуру відповідно до значення оберту, яка потім буде використано для обрахування нових координат текстури (рис. 8)

Рисунок 8. Приклад оберту текстури



**Лістинг програмного коду**

1. Шейдер вершин для поверхні

// Vertex shader

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

// Fragment shader

const fragmentShaderSource = `

precision mediump float;

varying vec2 v\_texCoord;

uniform sampler2D u\_texture;

uniform float u\_textureRotation;

uniform vec2 u\_rotationPoint;

vec2 rotatePoint(vec2 point, vec2 center, float angleRadians) {

    vec2 translatedPoint = point - center;

    float s = sin(angleRadians);

    float c = cos(angleRadians);

    vec2 rotatedPoint;

    rotatedPoint.x = translatedPoint.x \* c - translatedPoint.y \* s;

    rotatedPoint.y = translatedPoint.x \* s + translatedPoint.y \* c;

    return rotatedPoint + center;

}

void main() {

    vec2 rotatedTexCoord = rotatePoint(v\_texCoord, u\_rotationPoint, u\_textureRotation);

    // Fetch texel from the texture

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

    // Output the color from the texture

    gl\_FragColor = texColor;

}`;

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 / 2;

    }

    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, deg2rad(rotate\_texture\_value));

        gl.enable(gl.TEXTURE\_2D);

        let u = map(point\_on\_surface.u, 0, u\_max, 0, 1);

        let v = map(point\_on\_surface.v, 0, v\_max, 0, 1);

        gl.uniform2fv(shProgram.uRotationPoint, [u, v]);

        gl.uniform1f(shProgram.iRotateValue, 0);

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

    }

}

1. Конструктор програми шейдера в main.js

// Constructor

function ShaderProgram(name, program) {

    this.name = name;

    this.prog = program;

    this.iSolidColor = -1;

    this.iAttribVertex = -1;

    this.iModelViewProjectionMatrix = -1;

    // Texture rotation uniform

    this.uTextureRotation = -1;

    // Get uniform locations

    this.uTexture = -1;

    // Get attribute location for texture coordinates

    this.aTexCoord = -1;

    this.Use = function() { gl.useProgram(this.prog);}}

1. Функція для розрахунку координат текстури

function CreateTexture() {

    let texture = [];

    let u = 0;

    let v = 0;

    let uMax = u\_max;

    let vMax = v\_max;

    let uStep = u\_max / numPointsU;

    let vStep = v\_max / numPointsV;

    for (let u = 0; u <= u\_max; u += uStep) {

        for (let v = 0; v <= v\_max; v += vStep) {

            let u1 = map(u, 0, uMax, 0, 1)

            let v1 = map(v, 0, vMax, 0, 1)

            texture.push(u1, v1)

            u1 = map(u + uStep, 0, uMax, 0, 1)

            texture.push(u1, v1)

            u1 = map(u, 0, uMax, 0, 1)

            v1 = map(v + vStep, 0, vMax, 0, 1)

            texture.push(u1, v1)

            u1 = map(u + uStep, 0, uMax, 0, 1)

            v1 = map(v, 0, vMax, 0, 1)

            texture.push(u1, v1)

            v1 = map(v + vStep, 0, vMax, 0, 1)

            texture.push(u1, v1)

            u1 = map(u, 0, uMax, 0, 1)

            v1 = map(v + vStep, 0, vMax, 0, 1)

            texture.push(u1, v1)

        }

    }

    console.log(texture.length);

    return texture;

}

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);

}