**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”

на тему “Операції з текстурними координатами”

Виконав

студент групи

ТР-31мп

Турулько О. В.

Київ - 2023

1. **Завдання**

● Нанести текстуру на поверхню з практичного завдання №2.

● Реалізувати обертання текстури (координати текстури) обертання навколо визначеної користувачем точки – 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.

2. **Теоретичні відомості**

**WebGL** — це технологія, яка дозволяє веб-браузерам рендерити тривимірну графіку за допомогою програмного забезпечення. Вона заснована на OpenGL, стандартній API для тривимірної графіки, яка використовується в іграх та інших графічних додатках.

**WebGL** дозволяє веб-розробникам створювати інтерактивні тривимірні веб-додатки, такі як ігри, віртуальні тури та навчальні програми. Вона також може використовуватися для створення візуальних ефектів на веб-сайтах.

**WebGL** працює, надсилаючи команди до графічної карти комп'ютера користувача. Графічна карта відповідає за рендеринг тривимірних об'єктів на екрані.

**WebGL** має ряд переваг перед іншими технологіями тривимірної графіки для вебу. Вона:

* Легко інтегрується з веб-браузерами.
* Не вимагає додаткового програмного забезпечення.
* Підтримує широкий спектр графічних карт.

**WebGL** може використовуватися для створення широкого спектру тривимірних веб-додатків. Ось кілька прикладів:

* Ігри: **WebGL** можна використовувати для створення тривимірних ігор, таких як **Angry Birds**, **Minecraft** та **Fortnite**.
* Віртуальні тури: **WebGL** можна використовувати для створення віртуальних турів, таких як **Google Earth** та **Street View**.
* Навчальні програми: **WebGL** можна використовувати для створення навчальних програм, таких як **Khan Academy** та **3DMol.org**.
* Візуальні ефекти: **WebGL** можна використовувати для створення візуальних ефектів на веб-сайтах, таких як плавні анімації та складні переходи.

**WebGL** — це потужна технологія, яка може використовуватися для створення інноваційних тривимірних веб-додатків. Вона все ще розвивається, але вже зараз має широке застосування.

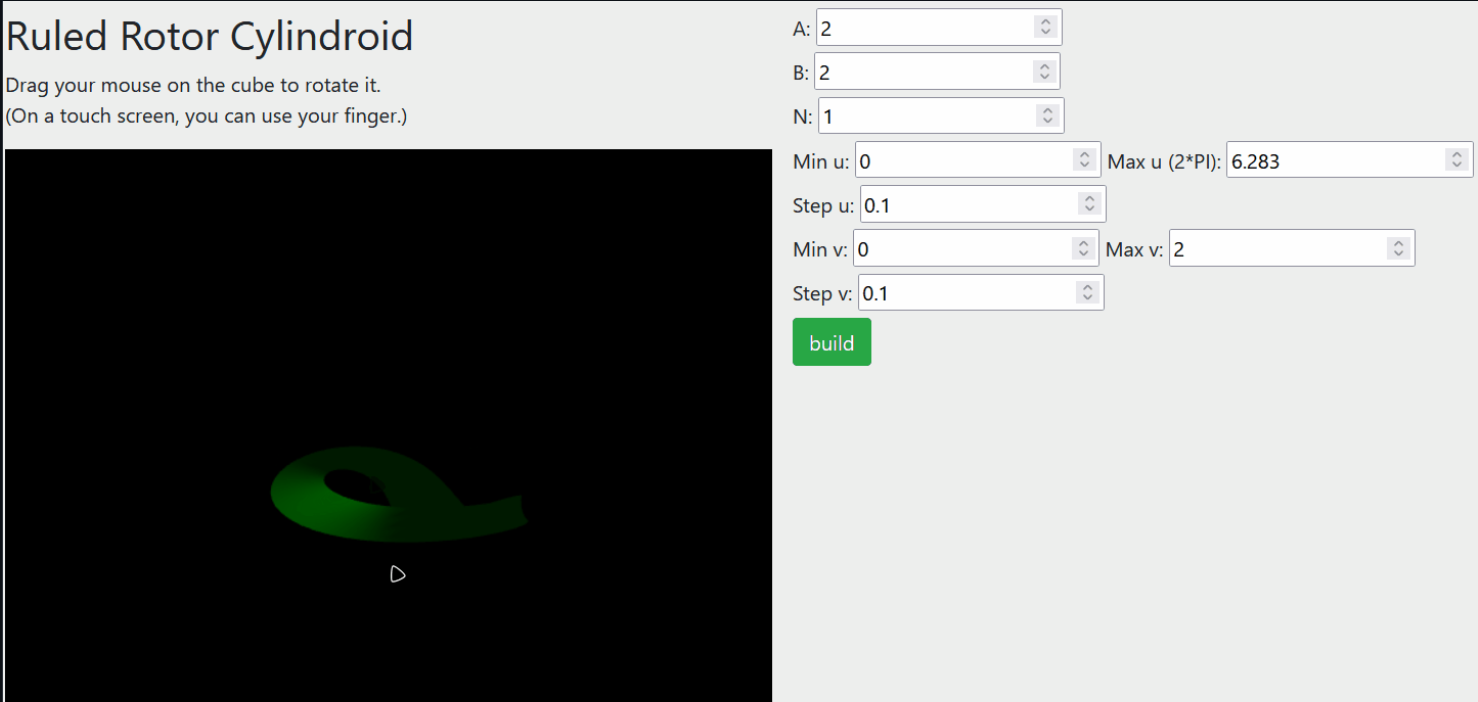
**Етапи ініціалізації проєкту з використанням можливостей WebGL:**

1. **Створення Контексту WebGL:**
   * Початковий етап - отримання контексту WebGL за допомогою об'єкта canvas та методу getContext.
   * Перевірка наявності контексту перед продовженням роботи.
2. **Завантаження та Компіляція Шейдерів:**
   * Визначення шейдерів - вершинний та фрагментний, написаних на мові GLSL.
   * Створення, компіляція та лінковка програми шейдерів для використання в WebGL.
3. **Створення та Завантаження Буферів та Текстур:**
   * Визначення вершин та інших даних для обробки.
   * Створення буферів та завантаження в них даних.
   * Завантаження текстур для подальшого використання.
4. **Налаштування Камери та Відображення:**
   * Встановлення параметрів камери для визначення точки спостереження.
   * Налаштування параметрів проекції для визначення області відображення.
5. **Використання WebGL API для Малювання:**
   * Використання методів WebGL API для малювання графічних об'єктів.
   * Організація циклу малювання для анімації та оновлення сцени.
6. **Обробка Подій та Взаємодія з Користувачем:**
   * Додавання обробників подій для взаємодії з мишкою, клавіатурою та іншими введеннями.
   * Реалізація відгуків на події для створення інтерактивних елементів.
7. **Оптимізація та Налагодження:**
   * Вдосконалення шейдерів та алгоритмів для оптимізації продуктивності.
   * Використання інструментів розробника для відладки та аналізу продуктивності.

З цими етапами розробники можуть створювати вражаючі веб-застосунки з тривимірною графікою, використовуючи потужні можливості WebGL.

3. Деталі розробки

Згідно з поставленим завданням, моєю метою було вдосконалити поверхню, створену на основі другої лабораторної роботи, додавши до неї текстуру. У цій лабораторній роботі була реалізована поверхня Ruled Rotor Cylindroid (рис. 1). Для її створення використовувався метод, який базується на побудові фігур за допомогою трикутників.

  
рисунок 1 – інтерфейс виконання 2-ї ЛР

Для виконання РГР в якості текстури було обрано зображення формату . jpg розміром 512х512 пікселів.

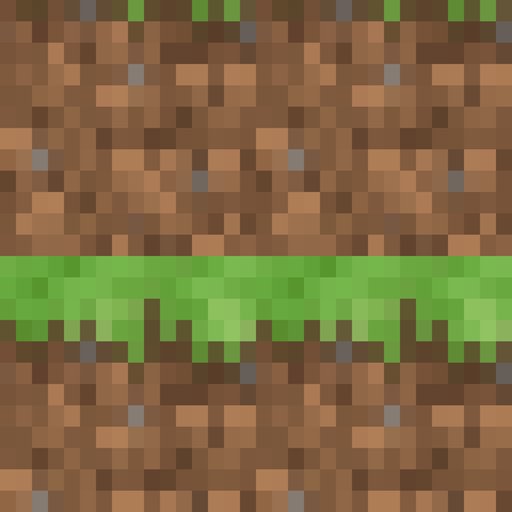


Рисунок 2 - Зображення текстури

Для того щоб нанести текстуру на поверхню, було створено буфер текстурних координат, де кожна координата відповідає відповідному елементу масиву вершинного буфера. У програмі шейдера для визначення кольору пікселя фігури, на яку застосована текстура, необхідно використовувати функцію texture2D(). Перший аргумент цієї функції - об'єкт класу sampler2D, який зберігає дані про зображення, а другий - текстурна координата.

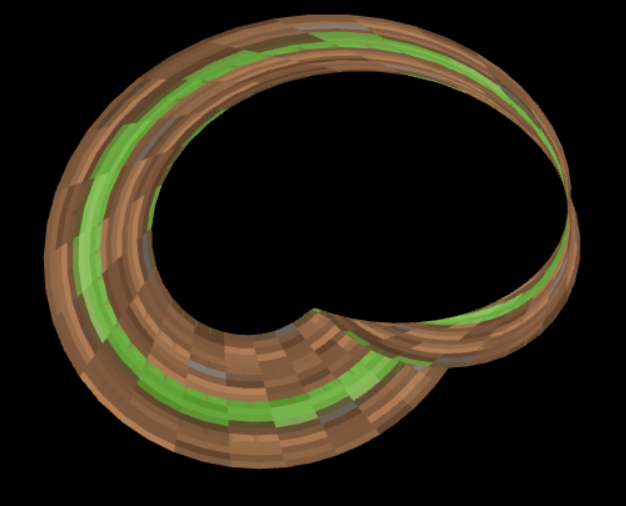


Рисунок 3 - фігура з накладеною текстурою

Після цього було створено новий екземпляр класу Model для відображення точки у формі сфери, що визначає місце, відносно якого буде застосована трансформація текстури. Ця точка призначена розташовуватися на поверхні фігури.



Рисунок 4 – точка на текстурованій поверхні

Згідно варіанту, було реалізовано обертання текстури навколо точки.

4. Інструкція користувача.

Інтерфейс користувача містить canvas та input типу range котрий забезпечує введення значення кута обороту текстури. Для того щоб обернути фігуру, потрібно затиснути будь-яку клавішу на миші.

При значенні кута 0 фігура має наступний вигляд:

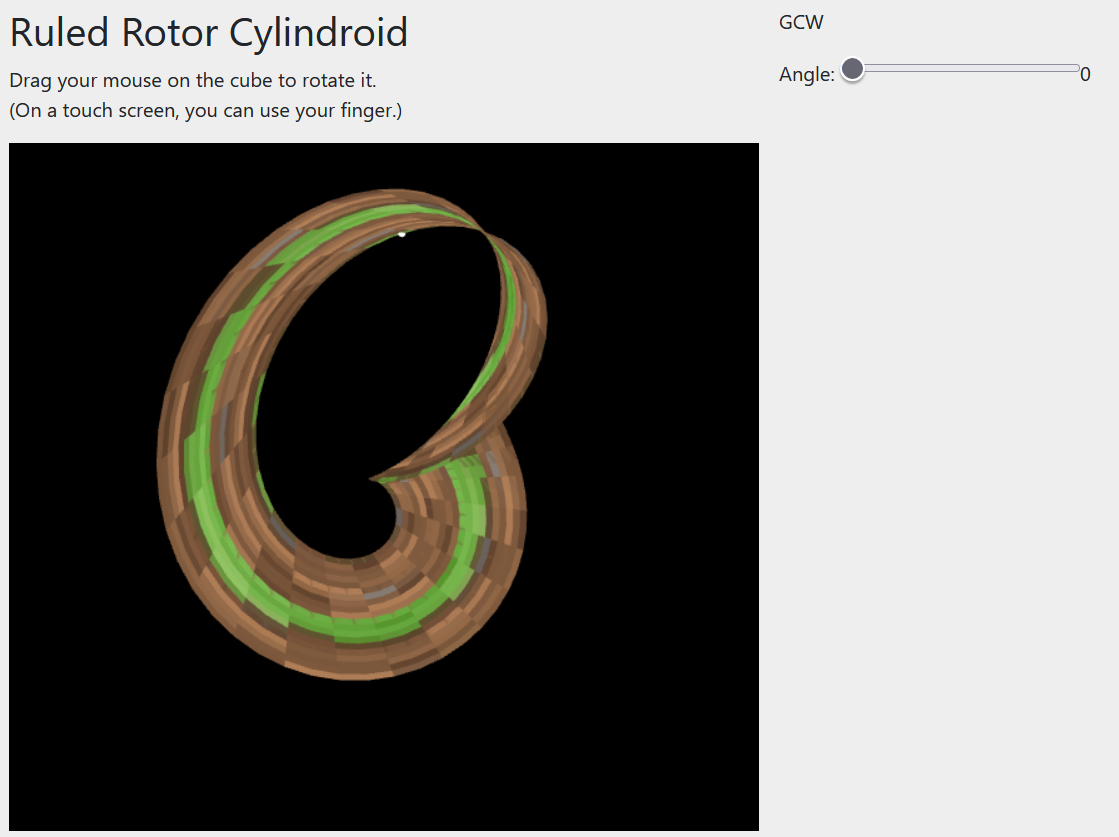


Рисунок 5 – фігура при значенні кута 0

І при значенні кута 90 фігура має наступний вигляд:

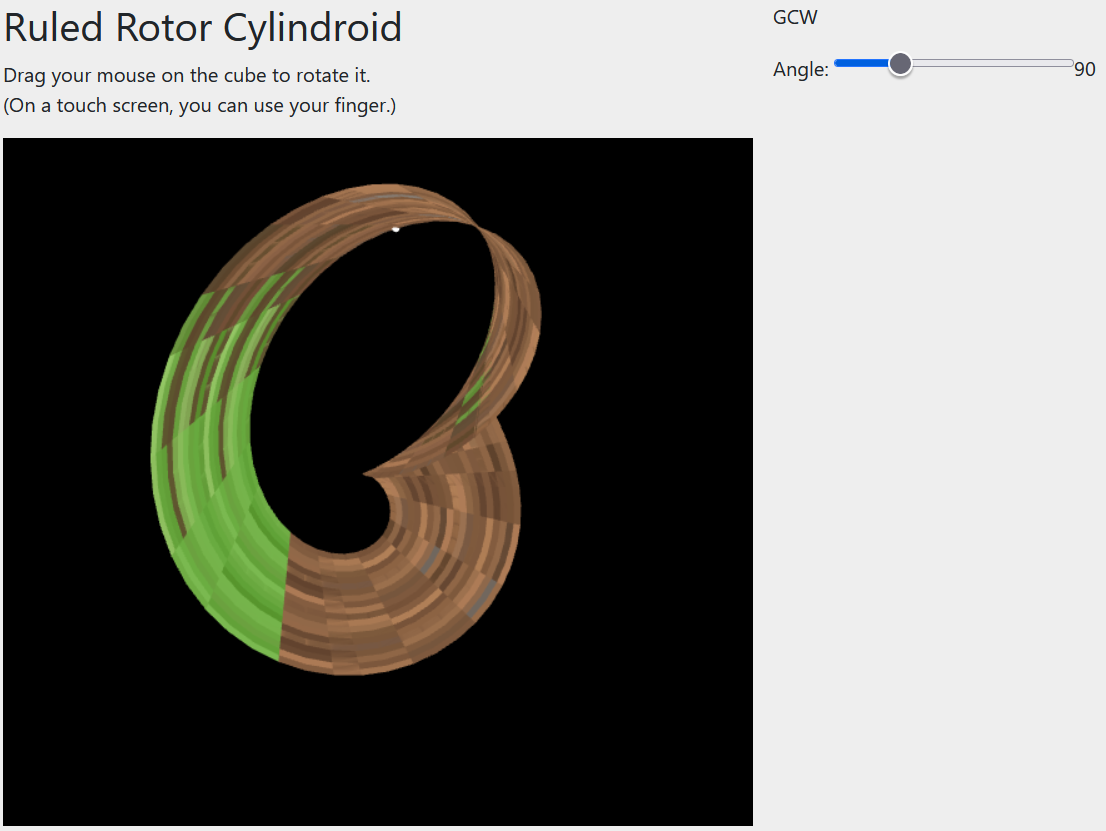


Рисунок 6 – фігура при значенні кута 90

Код програми:

const map = (val, fromRangeStart, fromRangeEnd, toRangeStart, toRangeEnd) =>Math.min(Math.max( (val - fromRangeStart) \* (toRangeEnd - toRangeStart) / (fromRangeEnd - fromRangeStart) + toRangeStart, toRangeStart),toRangeEnd);

function CreateSurfaceTextureData() {

let vertexTextureList = [];

for (let u = u\_min; u <= u\_max; u += step\_u) {

for (let v = v\_min; v <= v\_max; v += step\_v) {

let u1 = map(u, 0, u\_max, 0, 1)

let v1 = map(v, 0, v\_max, 0, 1)

vertexTextureList.push(u1, v1)

u1 = map(u + 0.1, 0, u\_max, 0, 1)

vertexTextureList.push(u1, v1)

u1 = map(u, 0, u\_max, 0, 1)

v1 = map(v + 0.1, 0, v\_max, 0, 1)

vertexTextureList.push(u1, v1)

u1 = map(u + 0.1, 0, u\_max, 0, 1)

v1 = map(v, 0, v\_max, 0, 1)

vertexTextureList.push(u1, v1)

v1 = map(v + 0.1, 0, v\_max, 0, 1)

vertexTextureList.push(u1, v1)

u1 = map(u, 0, u\_max, 0, 1)

v1 = map(v + 0.1, 0, v\_max, 0, 1)

vertexTextureList.push(u1, v1)

}

}

return vertexTextureList;

}

function CreateSphereSurface() {

let vertexList = [];

let step = 0.1;

let radius = 0.1;

const getSphereVertex = (u, v) =>{

return {

x: radius \* Math.cos(u) \* Math.sin(v),

y: radius \* Math.sin(u) \* Math.sin(v),

z: radius \* Math.cos(v)

}

};

for (let u = 0; u <= Math.PI; u += step) {

for (let v = 0; v <= Math.PI \* 2; v += step) {

let v1 = getSphereVertex(u, v);

let v2 = getSphereVertex(u + step, v);

let v3 = getSphereVertex(u, v + step);

let v4 = getSphereVertex(u + step, v + step);

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(v3.x, v3.y, v3.z);

vertexList.push(v2.x, v2.y, v2.z);

vertexList.push(v4.x, v4.y, v4.z);

}

}

return [vertexList,vertexList];

}

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 = "texture.jpeg";

image.onload = () => {

gl.bindTexture(gl.TEXTURE\_2D, texture);

gl.texImage2D(

gl.TEXTURE\_2D,

0,

gl.RGBA,

gl.RGBA,

gl.UNSIGNED\_BYTE,

image

);

draw()

}

}

Код шейдера:  
// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 vertexTexture;

uniform mat4 ModelViewProjectionMatrix;

varying vec2 textureCoord;

uniform float b;

uniform vec3 translateSphere;

uniform vec2 userPoint;

uniform float angle;

mat4 translateForPoint(float tx, float ty, float tz) {

return mat4(

vec4(1.0, 0.0, 0.0, 0.0),

vec4(0.0, 1.0, 0.0, 0.0),

vec4(0.0, 0.0, 1.0, 0.0),

vec4(tx, ty, tz, 1.0)

);

}

mat4 translationForVec(vec3 t) {

return mat4(

vec4(1.0, 0.0, 0.0, 0.0),

vec4(0.0, 1.0, 0.0, 0.0),

vec4(0.0, 0.0, 1.0, 0.0),

vec4(t, 1.0)

);

}

mat4 rotation(float angleInRadians) {

float c = cos(angleInRadians);

float s = sin(angleInRadians);

return mat4(

vec4(c, s, 0.0, 0.0),

vec4(-s, c, 0.0, 0.0),

vec4(0.0, 0.0, 1.0, 0.0),

vec4(0.0, 0.0, 0.0, 1.0)

);

}

void main() {

mat4 rMatrix = rotation(angle);

mat4 tMatrix1 = translateForPoint(userPoint.x, userPoint.y, 0.0);

mat4 tMatrix2 = translateForPoint(-userPoint.x, -userPoint.y, 0.0);

vec4 textureT1 = vec4(vertexTexture, 0.0, 0.0) \* tMatrix1;

vec4 textureR = textureT1 \* rMatrix;

vec4 textureT2 = textureR \* tMatrix2;

textureCoord = vec2(textureT2.x, textureT2.y);

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

if (b > 0.0) {

vec4 sphere = translationForVec(translateSphere) \* vec4(vertex, 2.0);

gl\_Position = ModelViewProjectionMatrix \* sphere;

}

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

precision highp float;

#else

precision mediump float;

#endif

varying vec2 textureCoord;

uniform sampler2D tmu;

uniform float b;

void main() {

vec4 texColor = texture2D(tmu, textureCoord);

gl\_FragColor = (b > 0.0) ? vec4(1.0) : texColor;

}`;