**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”

Варіант 29

Виконав:

студент групи ТР-31мп

Уваровський Олександр Дмитрович

Київ - 2023

1. **Завдання**

* Нанести текстуру на поверхню з практичного завдання #2.
* Реалізувати масштабування текстури (координат текстури) масштабування навколо визначеної користувачем точки.
* Імплементувати переміщення точки вздовж простору поверхні (u,v) використовуючи клавіатуру клавішами A і D, які переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.

1. **Опис завдання**

**WebGL**

WebGL - це JavaScript API, яке використовується для створення інтерактивної 2D- та 3D-графіки у веб-браузері. Для досягнення ефектів, таких як відображення текстур, масштабування, обертання та інтерактивне переміщення точок, WebGL API має ряд функцій для обробки шейдерів, текстур і матричних перетворень.

Шейдери використовуються для налаштування обробки вершин і фрагментів, дозволяючи включати текстурні координати та здійснювати перетворення. Текстурні об'єкти створюються та прив'язуються до конкретних текстурних одиниць для використання в шейдерах.

Матричні перетворення застосовуються для управління положенням, масштабуванням і обертанням 3D-моделі.

**Шейдери у WebGL**

У програмуванні WebGL шейдери виконують ключову роль у візуалізаційному конвеєрі, забезпечуючи гнучкий та програмований підхід до визначення зовнішнього вигляду 3D-графіки. У WebGL існують два основних типи шейдерів: вершинні та фрагментні.

Вершинні шейдери відповідають за обробку кожної вершини 3D-моделі перед її відображенням на екрані. Вони обробляють атрибути вершин, такі як положення, колір і координати текстури, для створення вихідних даних. Зазвичай вершинні шейдери використовуються для застосування трансформацій до позицій вершин, таких як переміщення, обертання та масштабування. Ці трансформації є важливими для позиціонування об'єктів у 3D-просторі. Вихідні дані вершинного шейдера включають перетворену позицію вершини та інтерпольовані значення для передачі фрагментному шейдеру.

Фрагментні шейдери, іноді називані піксельними шейдерами, працюють з кожним пікселем, який буде відображений на екрані. Вони отримують інтерпольовані значення від вершинного шейдера, такі як координати кольору та текстури, а також інші дані, наприклад, про освітлення. Основне завдання фрагментного шейдера - визначити остаточний колір кожного пікселя, що може включати вибірку текстури, обчислення освітлення та інші ефекти. Кінцевий колір використовується для малювання пікселя на екрані.

Як вершинний, так і фрагментний шейдери написані мовою GLSL (OpenGL Shading Language) і потребують компіляції перед використанням. Після компіляції вони об'єднуються в програму шейдера, яку необхідно зв'язати перед відтворенням. Це дозволяє WebGL використовувати вказані вершинні та фрагментні шейдери для відтворення графіки.

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

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

Координати текстури, як правило, 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 з текстур, які потім використовуються для визначення остаточного кольору кожного пікселя.

**Накладання текстур. Текстурні координати**

Техніка відображення текстур у комп'ютерній графіці дозволяє створювати реалістичні поверхні, застосовуючи зображення або текстури до 3D-моделей. В контексті WebGL відображення текстур передбачає прив'язку кожної вершини 3D-об'єкта до координат текстури (u, v), які використовуються для вибірки кольорів із зображення текстури. Це покращує візуальний вигляд об'єкта, забезпечуючи детальну інформацію про його поверхню.

Координати текстури (u, v) є ключовими параметрами, які визначають, як текстура наноситься на поверхню. Вони змінюються 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, надаючи їй реалістичний вигляд.

1. **Деталі розробки**

За варіантом 29 фігура: конічна поверхня з кривою директриси на сфері. При виконанні практичного завдання №2 було модифіковано програму, що відмальовує поверхню у вигляді трикутників.

A blue and black logo

Description automatically generated

**Рисунок 1 - конічна поверхня з кривою директриси на сфері**

Для накладення обрав зображення дерева формату jpg, і розміром 512x512.

A close-up of a wood grain

Description automatically generated

Рисунок 2 - Зображення текстури дерева обраної для завдання

Текстура була накладена на поверхню, і для цього було підготовлено буфер текстурних координат. Кожна текстурна координата відповідає елементу масиву з буферу вершин. Згідно з варіантом, текстура повинна масштабуватися, тому було створено відповідний параметр типу uniform, який визначатиме фактор масштабування. У програмі шейдера для визначення кольору пікселя фігури, на яку накладено текстуру, необхідно використовувати функцію texture2D(). Ця функція отримує об'єкт класу sampler2D як перший аргумент, який містить дані про зображення, та другий аргумент - текстурну координату.

A close up of a fan

Description automatically generated

Рисунок 3 – Накладена на поверхню текстура дерева

Створено новий екземпляр класу "Model" для представлення точки, відносно якої буде проводитися трансформація текстури. Графічне відображення цього об'єкта представлене у вигляді сфери, яка розташована на поверхні.

A brown bird with a white circle in the middle

Description automatically generated

Рисунок 4 - Точка на текстурованій поверхні

Згідно варіанту було розроблено масштабування текстури відносно точки на поверхні.

1. **Інструкція користувача**

Фігуру можна обертати відносно центру затиснувши ЛКМ та потягнувши в сторону бажаного обертання.

A close up of a piece of wood

Description automatically generated

A circular object with a white circle in the middle

Description automatically generated

Рисунок 5 - Вигляд фігури до та після масштабування

Для здійснення обертання точки використовуються клавіші WASD. Кожне натискання дозволяє перемістити точку по поверхні на певний крок. Переміщення відбувається до досягнення визначеної межі.

A circular object with a white circle in the middle

Description automatically generated

A wooden half-moon with a white circle

Description automatically generated

Рисунок 6 – Переміщення точки відносно площини

Використовуючи слайдер із підписом "Scaler", можна регулювати масштабування відносно умовної точки на поверхні. При ненульовому значенні масштабування помітно, що при переміщенні точки також зсувається текстура. Це пояснюється тим, що масштабування відбувається відносно іншої точки на поверхні, яка в свою чергу відповідає іншій текстурній координаті.

A computer screen shot of a wood half-moon

Description automatically generated

A screenshot of a computer

Description automatically generated

Рисунок 7 - Зміна фактору масштабування текстури

Після перезавантаження сторінки параметри обертання фігури і значення фактору масштабування будуть автоматично встановлені на мінімальні значення за замовчуванням. Положення точки відносно поверхні також буде скинуто до значень за замовчуванням.

1. **Код програми**

Код частини програми на JavaScript JavaScript — Википедия

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);

let projection = m4.orthographic(-3, 3, -3, 3, -3, 3);

/\* 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, -0);

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);

let normalMatrix = m4.identity();

m4.inverse(modelView, normalMatrix);

normalMatrix = m4.transpose(normalMatrix, normalMatrix);

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

gl.uniformMatrix4fv(shProgram.iNormalMatrix, false, normalMatrix);

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

let color = hexToRgb(document.getElementById("color").value)

gl.uniform4fv(shProgram.iColor, [color.r / 255.0, color.g / 255.0, color.b / 255.0, 1]);

gl.uniform1f(shProgram.iScale,

document.getElementById('scl').value)

gl.uniform2fv(shProgram.iTranslate, pointTr)

surface.Draw();

gl.uniform4fv(shProgram.iLight, [1, 1, 1, 1]);

gl.uniform1f(shProgram.iScale, -222)

gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false,

m4.multiply(modelViewProjection,

m4.translation(...conical(map(pointTr[0], 0, 1, -Math.PI, Math.PI), map(pointTr[1], 0, 1, -a, 0))))

);

pointViz.Draw();

main функція на GLSL

void main() {

vec3 N = normalize(normalVarying);

vec3 L = normalize(vec3(1.0));

// Lambert's cosine law

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

float specular = 0.0;

if(lambertian > 0.0) {

vec3 R = reflect(-L, N); // Reflected light vector

vec3 V = normalize(-vertexVarying); // Vector to viewer

// Compute the specular term

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

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);

}

}`;