**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 30

Виконала студентка групи ТР-31мп

Цюрупович Катерина Валеріївна

Київ - 2023

1. **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.

1. **Теорія**

WebGL

WebGL — це API JavaScript для відтворення інтерактивної 2D- і 3D-графіки у веб-браузері. Щоб реалізувати відображення текстур, масштабування, обертання та інтерактивне переміщення точок, WebGL API надає функції для обробки шейдерів, текстур і перетворень матриці.

* Шейдери використовуються для визначення того, як обробляються вершини та фрагменти, дозволяючи включати текстурні координати та перетворення.
* Текстурні об’єкти створюються та прив’язуються до певних текстурних одиниць для вибірки в шейдерах.
* Матричні перетворення застосовуються для керування положенням, масштабуванням і обертанням 3D-моделі.

Шейдери WebGL

У WebGL шейдери відіграють вирішальну роль у конвеєрі візуалізації, забезпечуючи гнучкий і програмований спосіб визначення вигляду 3D-графіки. У WebGL є два основних типи шейдерів: вершинні та фрагментні.

Вершинні шейдери відповідають за обробку кожної вершини 3D-моделі перед її відображенням на екрані. Вершинні шейдери беруть атрибути з даних вершин, такі як положення, колір і координати текстури, і обробляють їх для створення виходу. Одним із поширених способів використання вершинних шейдерів є застосування трансформацій до позицій вершин. Це включає переміщення, обертання та масштабування, які є вирішальними для позиціонування об’єктів у 3D-просторі. Вихідні дані вершинного шейдера містять перетворену позицію вершини та будь-які інтерпольовані значення, які будуть передані фрагментному шейдеру.

Фрагментні шейдери, які іноді називають піксельними шейдерами, працюють з кожним пікселем, який буде намальовано на екрані. Фрагментні шейдери отримують інтерпольовані значення від вершинного шейдера, наприклад координати кольору та текстури, а також інші дані, як-от інформацію про освітлення. Основним завданням фрагментного шейдера є визначення остаточного кольору кожного пікселя. Це може включати вибірку текстури, обчислення освітлення та інші ефекти. Кінцевий вихідний колір потім використовується для малювання пікселя на екрані.

І вершинний, і фрагментний шейдери написані мовою GLSL (OpenGL Shading Language) і їх потрібно скомпілювати, перш ніж їх можна буде використовувати. Після компіляції вони об’єднуються в програму шейдера. Програма шейдера повинна бути зв’язана перед відтворенням, дозволяючи WebGL використовувати вказані вершинні та фрагментні шейдери для конвеєра відтворення.

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

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

Координати текстури зазвичай 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 із текстур. Вибрані кольори потім використовуються для визначення остаточного кольору кожного пікселя.

Накладання текстур. Текстурні координати

Відображення текстури — це техніка в комп’ютерній графіці, яка дозволяє відтворювати реалістичні поверхні шляхом нанесення зображень або текстур на 3D-моделі. У контексті WebGL відображення текстур передбачає асоціювання кожної вершини 3D-об’єкта з координатами текстури (u, v), які потім використовуються для вибірки кольорів із зображення текстури. Це покращує візуальний вигляд об’єкта, надаючи детальну інформацію про поверхню.

Координати текстури (u, v) є важливими параметрами, які визначають спосіб нанесення текстури на поверхню. Вони варіюються 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.

1. **Деталі розробки**

## За варіантом мені було надано «Conical Surface with a Directrix Curve in the Form of Cartesian Folium». При виконанні практичного завдання №2 було розроблено програму, що виводить поверхню у вигляді суцільних трикутників.

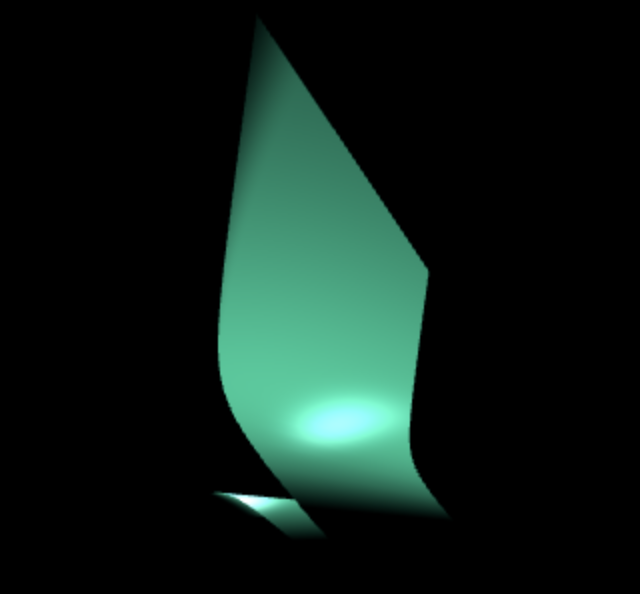


Рисунок 1 - Вигляд поверхні

Я обрала наступне зображення для подальшого виконання розрахунково-графічної роботи. Для підтримки на більшій кількості операційних систем та браузерів було обрано зображення розміром 512х512 пікселів. Формат зображення .png



Рисунок 2 - Зображення обраної текстури

Було накладено текстуру на поверхню. Для накладання текстури на поверхню було підготовлено буфер текстурних координат, кожна з яких відповідає елементу масива з буферу вершин. Згідно варіанту текстура має обертатися, тому було створено відповідний uniform, який визначатиме кут обертання. В програмі шейдера для визначення кольору пікселя фігури на яку накладено текстуру необхідно використовувати функцію texture2D(). Ця функція першим аргументом приймає обʼєкт класу sampler2D, який зберігає дані про зображення, а другим текстурну координату.

Було створено новий обʼєкт класу Model для відображення точки відносно якої буде здійснюватись трансформація текстури. Обʼєкт графічно відображається як сфера. Сфера перебуває на поверхні. Також згідно варіанту було імплементовано обертання текстури відносно точки на поверхні.

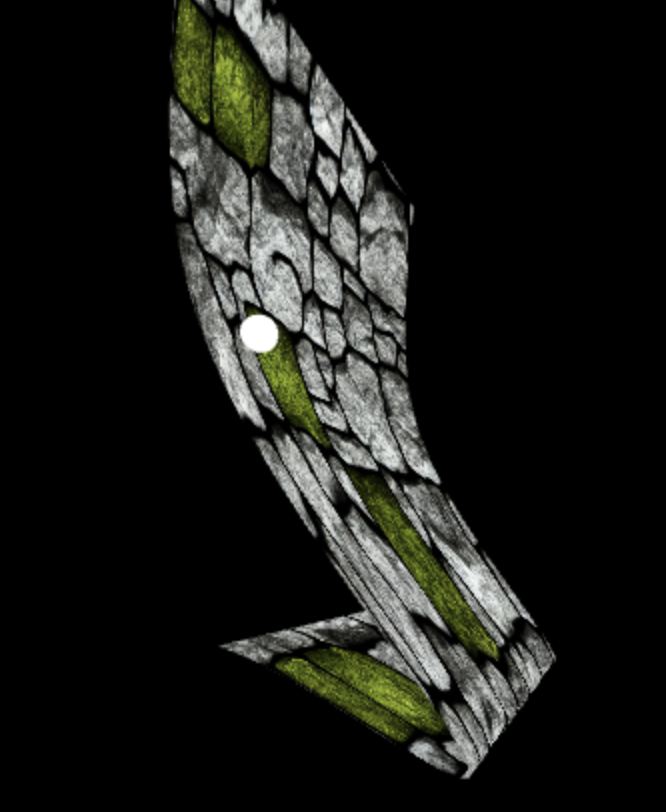


Рисунок 3 - Текстура накладена на поверхню

1. **Інструкції користувача**

Фігуру можна обертати відносно центру затиснувши ліву клавішу миші та потягнувши в сторону бажаного обертання.

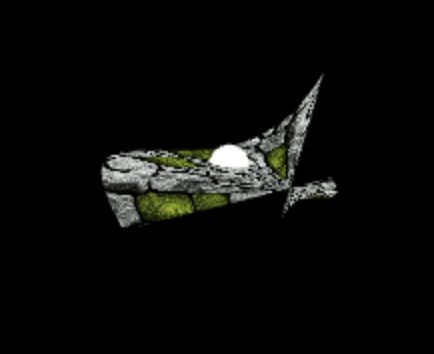


Рисунок 4 - Вигляд фігури до та після обертання

Переміщувати точку відносно якої здійснюється обертання можна за допомогою клавіш WASD. Кожне натискання переміщує точку по поверхні на визначений крок. Переміщення здійснюється до визначеної межі.

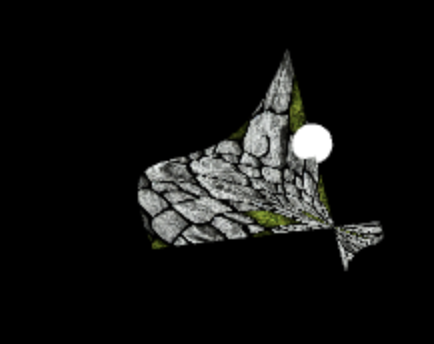
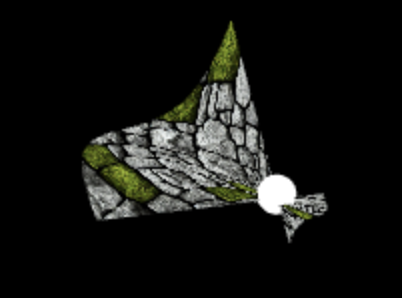


Рисунок 5 - Демонстрація переміщення точки відносно нерухомої фігури

За допомогою слайдера з підписом “Specify angle of texture rotation” можна змінювати кут обертання відносно умовної точки на поверхні. Якщо значення кута обертання не нульове Можна помітити, що при переміщенні точки зміщується і текстура, адже обертання відбувається відносно іншої точки на поверхні , яка в свою чергу відповідає іншій текстурній координаті.

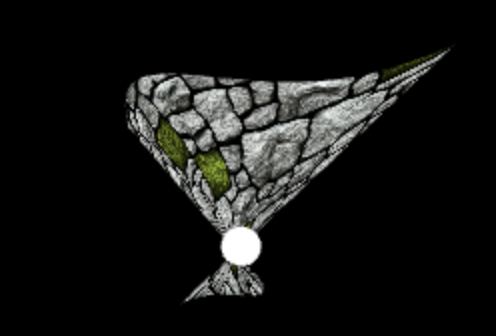
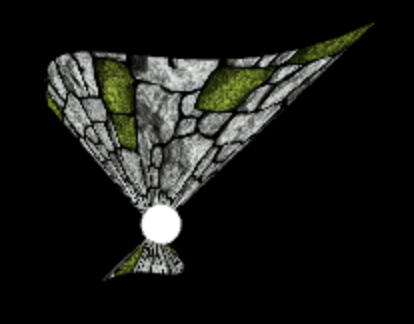


Рисунок 6 - Демонстрація зміни кута обертання текстури

При перезавантаженні сторінки обертання фігури, а також значення кута обертання буде встановлено по замовчуванню. Положення точки відносно поверхні також буде скинуто до значення по замовчуванню.

1. **Код програми**

Ластинг функціональної частини на javascript

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.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, -1);

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);

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

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

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

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

gl.uniform3fv(shProgram.iP, [-2 \* Math.sin(Date.now() \* 0.001) - 2, 0, -1]);

gl.uniform3fv(shProgram.iD, [1, 1, 0]);

gl.uniform1f(shProgram.iR, document.getElementById('r').value);

gl.uniform1f(shProgram.iF, document.getElementById('f').value);

gl.uniform2fv(shProgram.iT, position);

gl.uniform1f(shProgram.iA, document.getElementById('a').value);

surface.Draw();

gl.uniform1f(shProgram.iF, -1);

gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection,

m4.translation(...directrixSurface(position[0],(position[1]+0.3)/1.3))));

circle.Draw();

}

let L = 1

let T = 1

let B = 0.5

function CreateSurfaceData() {

L = document.getElementById('l').value

T = document.getElementById('t').value

B = document.getElementById('b').value

let vertexList = [];

let normalList = [];

let textureList = [];

const numb = 50;

const uE = 1 / numb;

const vE = 1.3 / numb;

for (let u = 0; u < 1; u += uE) {

for (let v = -0.3; v < 1; v += vE) {

let vertex1 = directrixSurface(u, v)

let vertex2 = directrixSurface(u + uE, v)

let vertex3 = directrixSurface(u, v + vE)

let vertex4 = directrixSurface(u + uE, v + vE)

let normal1 = directrixNormal(u, v)

let normal2 = directrixNormal(u + uE, v)

let normal3 = directrixNormal(u, v + vE)

let normal4 = directrixNormal(u + uE, v + vE)

vertexList.push(

...vertex1, ...vertex2, ...vertex3,

...vertex3, ...vertex2, ...vertex4,

)

normalList.push(

...normal1, ...normal2, ...normal3,

...normal3, ...normal2, ...normal4,

)

textureList.push(

u, v, u + uE, v, u, v + vE,

u, v + vE, u + uE, v, u + uE, v + vE,

)

}

}

return [vertexList, normalList, textureList];

}

function directrixSurface(u, v) {

let x = u

let y = (L - u) \* Math.tan(v)

let z = T \* (L - u) / L \* (((2 \* pow(B, 2)) / (4 \* pow(L, 2) \* pow(Math.tan(v), 2) + pow(B, 2))) - 1)

return [x, y, z]

}

const e = 0.001;

function directrixNormal(u, v) {

let uv = directrixSurface(u, v)

let ue = directrixSurface(u + e, v)

let ve = directrixSurface(u, v + e)

const dU = [(uv[0] - ue[0]) / e, (uv[1] - ue[1]) / e, (uv[2] - ue[2]) / e]

const dV = [(uv[0] - ve[0]) / e, (uv[1] - ve[1]) / e, (uv[2] - ue[2]) / e]

return m4.normalize(m4.cross(dU, dV))

}