**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 32

Виконав студент групи ТР-31мп

Яковенко Олександр Ігорович

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.

**WebGL**

WebGL (Web Graphics Library) - це технологія, яка надає можливість використовувати 3D-графіку у веб-браузерах. Вона є JavaScript API для створення високоефективних інтерактивних графічних застосунків прямо в браузері без використання додаткових плагінів.

Основні риси і можливості WebGL:

* 1. Графіка на рівні апаратного забезпечення: WebGL використовує можливості графічного апаратного забезпечення комп'ютера, що дозволяє створювати швидкі та ефективні 3D-зображення.
* 2. Вбудована веб-стек: WebGL інтегрована в стек веб-технологій, таких як HTML, CSS і JavaScript, що дозволяє легко взаємодіяти з іншими веб-елементами.
* 3. Крос-платформеність: WebGL підтримується багатьма сучасними веб-браузерами, такими як Google Chrome, Mozilla Firefox, Safari та іншими, що робить його крос-платформеним і доступним на різних пристроях і операційних системах.
* 4. 3D-графіка та візуалізація: WebGL особливо корисний для створення веб-застосунків, які використовують 3D-графіку, візуалізацію даних, ігри, ефекти та інші графічні елементи.
* 5. Використання шейдерів: WebGL дозволяє використовувати шейдери, які є програмами, що використовуються для обробки графіки на рівні графічного процесора (GPU). Це дозволяє створювати складні графічні ефекти та контролювати відображення сцени.

WebGL є потужним інструментом для розробки веб-застосунків, що використовують 3D-графіку, і відкриває широкі можливості для створення захоплюючих інтерактивних веб-середовищ.

**Шейдери WebGL**

Шейдери у WebGL - це невеликі програми, які використовуються для обробки графіки прямо на графічному процесорі (GPU). Вони складаються з двох основних типів: вершинний і фрагментний. Вершинний шейдер працює з вершинами графічних об'єктів, визначаючи їхнє положення та кольори. Фрагментний шейдер відповідає за кожен піксель на екрані, визначаючи його кольорові характеристики.

Шейдери написані мовою GLSL (OpenGL Shading Language) і використовуються для створення реалістичних графічних ефектів, таких як тіні, відбиття та освітлення. Вони дозволяють розробникам контролювати відображення сцени, створюючи складні візуальні враження. Шейдери гнучкі та потужні, і вони грають ключову роль у створенні тривимірної графіки в WebGL, дозволяючи створювати захоплюючі та реалістичні веб-застосунки.

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.

У світі WebGL шейдери є невід'ємною частиною створення вражаючих веб-графічних візуалізацій. Вершинний шейдер, як диригент, керує рухом та положенням вершин об'єктів, реалізуючи їхню трансформацію в тривимірному просторі. Фрагментний шейдер, в свою чергу, відповідає за кожен піксель на екрані, визначаючи його кольорові характеристики та текстурні властивості.

Мова GLSL, в якій написані ці шейдери, дозволяє розробникам виразно та ефективно визначати логіку обчислень. Шейдери взаємодіють, утворюючи злагоджену симфонію графічного представлення. Вони забезпечують можливість реалізації різноманітних графічних ефектів, включаючи тіні, блики та поверхневі текстури.

Що важливо, шейдери дозволяють створювати захоплюючі тривимірні сцени, які раніше були доступні переважно в ігровій індустрії. Їх потужність полягає в здатності програмно контролювати кожен елемент графічного відображення, вносячи творчість та індивідуальність у веб-розробку. Шейдери у WebGL є інструментом, який розширює межі можливостей для візуалізації у веб-середовищі, викликаючи захоплення та зацікавленість розробників у створенні захоплюючих та інтерактивних веб-додатків.

Накладання текстур. Текстурні координати

Накладання текстур у комп'ютерній графіці - це процес додавання зображень (текстур) на поверхні 3D-моделей, щоб зробити їх більш деталізованими та реалістичними. Текстурні координати - це пари чисел, які визначають точки на текстурі, які відповідають кожній точці на поверхні моделі. Це дозволяє "приклеїти" текстуру на модель так, щоб кожна її частина відображалася правильно.

Накладання текстур є важливим елементом у 3D-графіці, де воно додає деталі та реалізм. Текстурні координати визначають, як саме текстура буде розміщена на поверхні об'єкта, і вони вимірюються від 0 до 1. Це дозволяє точно визначити місце кожної частини текстури на поверхні, незалежно від розмірів самої текстури чи геометрії об'єкта.

Накладання текстур та використання текстурних координат є важливим для створення різних ефектів, таких як рельєф, тіні та зміна кольору на поверхні. Це сприяє покращенню візуальної якості графіки та надає можливість реалістично відтворювати різноманітні матеріали, такі як дерево, метал чи текстиль, на поверхнях 3D-моделей.

**Деталі розробки**

За варіантом мені було надано corrugated sphere. При виконанні практичного завдання №2 було розроблено програму, що виводить поверхню у вигляді суцільних трикутників.

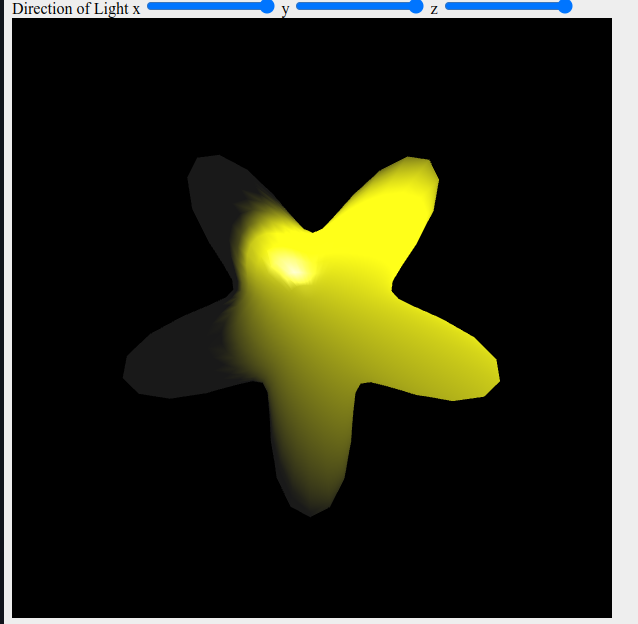


Рис.1 Сorrugated Sphere

Я обрав 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 .jpg.



Рисунок 2 - Зображення 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.

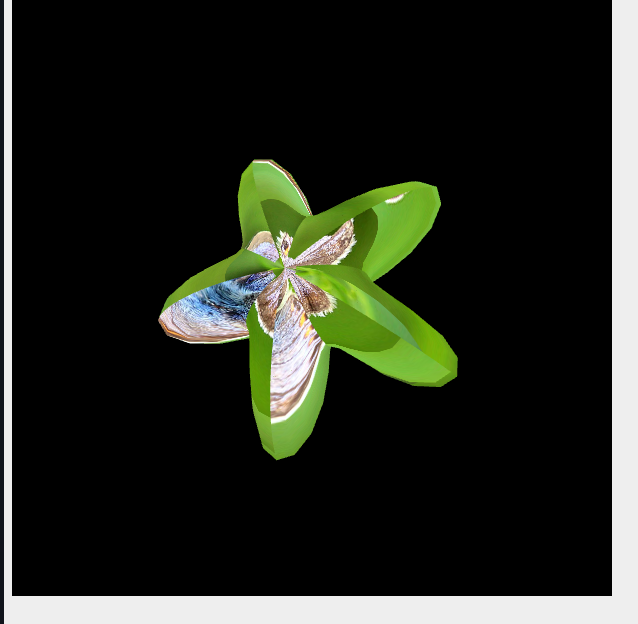


Рисунок 3 - Текстура накладена на поверхню

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

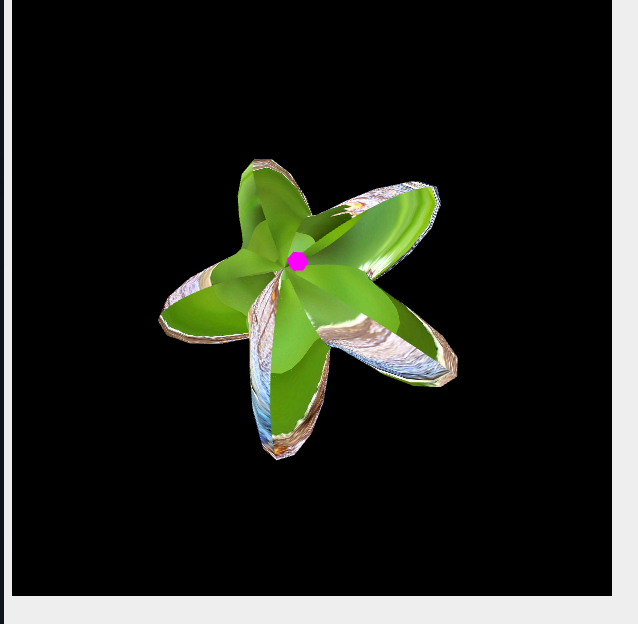


Рисунок 4 - Точка на текстурованій поверхні

Згідно варіанту було імплементовано обертання текстури відносно точки на поверхні.

**Інструкції користувача**

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

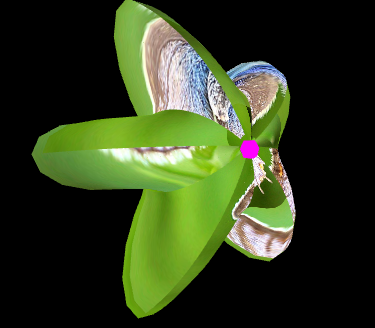


Рисунок 5 - Вигляд фігури до та після обертання

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.

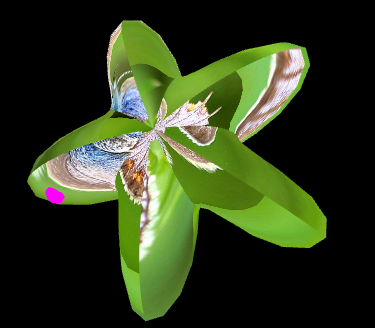




Рисунок 6 - Демонстрація переміщення точки відносно нерухомої фігури

За допомогою слайдера з підписом “Кут обертання текстури:” 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 7 - 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.

**Код програми**

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = gl.createBuffer();

this.iTextureBuffer = gl.createBuffer();

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.BufferNormalData = function (vertices) {

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

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

}

this.Draw = function () {

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

gl.enableVertexAttribArray(shProgram.iAttribTexCoord);

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

}

}

function CreateSphereList(step, r = 0.2) {

let vertexList = [];

let u = 0,

v = 0;

while (u < Math.PI \* 2) {

while (v < Math.PI) {

let v1 = getSphereVertex(u, v, r);

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

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

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

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);

v += step;

}

v = 0;

u += step;

}

return vertexList;

}

function getSphereVertex(long, lat, r) {

return {

x: r \* Math.cos(long) \* Math.sin(lat),

y: r \* Math.sin(long) \* Math.sin(lat),

z: r \* Math.cos(lat)

}

}

// Constructor

function ShaderProgram(name, program) {

this.name = name;

this.prog = program;

// Location of the attribute variable in the shader program.

this.iAttribVertex = -1;

// Location of the uniform specifying a color for the primitive.

this.iColor = -1;

// Location of the uniform matrix representing the combined transformation.

this.iModelViewProjectionMatrix = -1;

this.Use = function () {

gl.useProgram(this.prog);

}

}

/\* Draws a colored cube, along with a set of coordinate axes.

\* (Note that the use of the above drawPrimitive function is not an efficient

\* way to draw with WebGL. Here, the geometry is so simple that it doesn't matter.)

\*/

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, 20);

/\* 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, -14);

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 modelNormalMatrix = m4.identity();

m4.inverse(modelView, modelNormalMatrix);

modelNormalMatrix = m4.transpose(modelNormalMatrix, modelNormalMatrix);

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

gl.uniformMatrix4fv(shProgram.iModelNormalMatrix, false, modelNormalMatrix);

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

gl.uniform4fv(shProgram.iColor, [1, 1, 0, 0]);

gl.uniform2f(shProgram.iTexPoint, texPoint[0] / (Math.PI \* 2), texPoint[1] / Math.PI);

gl.uniform1f(shProgram.iRotate, document.getElementById('rotate').value);

surface.Draw();

gl.uniform4fv(shProgram.iColor, [1, 0, 1, 1]);

gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection, m4.translation(...CreateVertex(R, a, n, ...texPoint))));

sphere.Draw()

}

function CreateSurfaceData(R, a, n, segments) {

let vertexList = [];

let normalList = [];

for (let i = 0; i <= segments; i++) {

for (let j = 0; j <= segments1; j++) {

let phi = (j / segments1) \* Math.PI \* 2; // phi parameter (0 to 2pi)

let v = (i / segments) \* Math.PI; // v parameter (0 to pi)

let phiIncrement = (1 / segments1) \* Math.PI \* 2; // v parameter (0 to pi)

let vIncrement = (1 / segments) \* Math.PI; // v parameter (0 to pi)

vert++;

let vert1 = CreateVertex(R, a, n, phi, v)

let vert2 = CreateVertex(R, a, n, phi + phiIncrement, v)

let vert3 = CreateVertex(R, a, n, phi, v + vIncrement)

let vert4 = CreateVertex(R, a, n, phi + phiIncrement, v + vIncrement)

let tex1 = [phi / (Math.PI \* 2), v / Math.PI]

let tex2 = [(phi + phiIncrement) / (Math.PI \* 2), v / Math.PI]

let tex3 = [phi / (Math.PI \* 2), (v + vIncrement) / Math.PI]

let tex4 = [(phi + phiIncrement) / (Math.PI \* 2), (v + vIncrement) / Math.PI]

vertexList.push(

...vert1,

...vert2,

...vert3,

...vert3,

...vert2,

...vert4,

);

normalList.push(

...tex1,

...tex2,

...tex3,

...tex3,

...tex2,

...tex4,

);

}

}

return {

vertices: vertexList,

normals: normalList

};

}