Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

**Візуалізація графічної та геометричної інформації**

Розрахунково-графічна робота

Варіант №13

Виконав:

студент 5-го курсу ІАТЕ

групи ТР-32мп

Лиштван В.В.

Перевірив: Демчишин А. А.

Київ - 2023

**Завдання**

* Нанести текстуру на поверхню з практичного завдання №2.
* Реалізація масштабування текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання
* Повинна бути можливість переміщати точку 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

**Теорія**

WebGL (Web Graphics Library) – це технологія, яка дозволяє створювати тривимірну графіку в браузері без встановлення додаткових плагінів чи розширень. Основна ідея полягає в тому, щоб використовувати OpenGL ES з рядом доповнень для взаємодії з елементами веб-сторінки. Завдяки WebGL розробники можуть створювати вражаючі та інтерактивні веб-додатки, ігри та візуалізації, використовуючи високоякісну тривимірну графіку.

Текстура в цьому контексті – це зображення, яке можна призначити тривимірній моделі для додаткової деталізації, кольору, рельєфу та інших візуальних характеристик. Використання текстур дозволяє надавати об'єктам тривимірного простору деталізований вигляд, зберігаючи при цьому оптимізацію продуктивності та забезпечуючи високу якість візуалізації.

Створення текстур в WebGL зазвичай виконується за допомогою функції createTexture(). Далі ініціалізується об'єкт Image, який вказує на URL-адресу зображення, що буде використовуватись як текстура. Кожен вершинний піксель на 3D-моделі має відповідні координати текстури, які визначають, який фрагмент текстури буде використовуватись для цього пікселя. Застосування різних методів фільтрації, таких як білінійна або трилінійна фільтрація, допомагає оптимізувати використання текстур.

Існують різні види текстур, такі як 2D-текстури для загальних завдань, текстури висот (heightmaps) для рельєфу на поверхні об'єктів та куб-текстури для створення оточення чи інших ефектів, які вимагають 360-градусного зображення. 2D-текстури є найбільш поширеними і використовуються для розміщення зображень, кольорів та іншої графіки на поверхнях об'єктів у двовимірному просторі. Це може включати текстури для стін, ґрунту, обличчя персонажів чи будь-яких інших плоских або покритих поверхонь. Зазвичай вони представлені у форматі зображення, такому як JPEG або PNG. Текстури висот використовуються для створення рельєфу на поверхнях об'єктів, надаючи їм вимірюваність та об'єм. Зазвичай це представлено у вигляді чорно-білих зображень, де різниця в яскравості вказує на висоту об'єкта. Використання таких текстур може створити реалістичний ефект тривимірності без додаткового геометричного об'єму. Куб-текстури - це шестипласне зображення, призначене для створення враження просторовості в об'єктах у віртуальному середовищі. Кожен з шести боків куб-текстури відображає зображення з певного напрямку або орієнтації. Вони часто використовуються для створення оточення (environment mapping) або для імітації взаємодії об'єкта з оточуючим світом. Це може включати відблиски, тіні та інші ефекти, що змінюються в залежності від точки огляду.

Обертання в графіці – це важлива операція, що дозволяє змінювати орієнтацію об'єктів чи сцени навколо певної осі або точки. Використовуються математичні формули, зокрема матриці обертання, для опису трансформацій координат об'єктів у просторі. Обертання може бути реалізоване навколо довільної точки за допомогою техніки зсуву та відповідних матриць.

Усе це разом дозволяє створювати реалістичні сцени та візуалізації веб-додатків за допомогою WebGL, оптимізуючи продуктивність та ефективне використання ресурсів.

Накладання текстур та використання текстурних координат є важливою складовою у комп'ютерній графіці. Цей процес включає прив'язку кожної вершини 3D-об'єкта до координат текстури (u, v), що визначають спосіб накладання текстури. У контексті WebGL ці координати пов'язані з кожною вершиною 3D-моделі та інтерполюються по всій поверхні під час візуалізації. Це гарантує точне охоплення текстурою форми об'єкта та покращує візуальний вигляд, передаючи деталі про його поверхню.

**Деталі впровадження**

У практичному завданні №2 було дано поверхню варіанту 13, а саме Shoe Surface. Поверхня відображалась у вигляді TRAINGLE\_STRIP. Функція gl.drawArrays відповідала за відмалювання заданої поверхні. Створення даних для малювання відбувалось у функції CreateShoeSurfaceData.

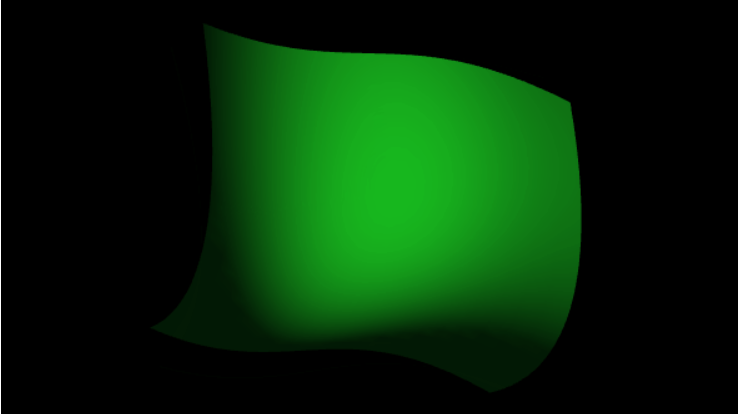


Рисунок 1 - Створена поверхня Shoe Surface

Для накладання текстури на задану поверхню було створено функцію LoadTexture. Обрана текстура зберігається на віддаленому сервері у форматі .jpg.

let texture = gl.createTexture(); - Спочатку створюється об'єкт текстури WebGL за допомогою методу createTexture().

gl.bindTexture(gl.TEXTURE\_2D, texture); - Встановлює активну текстуру для подальших операцій. В даному випадку, текстура типу gl.TEXTURE\_2D буде використовуватися.

gl.texParameteri(gl.TEXTURE\_2D,gl.TEXTURE\_MIN\_FILTER, gl.LINEAR); і gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.LINEAR); - Встановлює параметри фільтрації для мінімізації та максимізації текстури. У даному випадку використовується лінійна фільтрація.

const image = new Image(); - Створюється новий об'єкт зображення для завантаження текстурного зображення.

image.crossOrigin = "anonymous"; - Встановлює політику Cross-Origin Resource Sharing (CORS) для зображення. У цьому випадку використовується анонімний режим, що дозволяє завантаження зображення з інших доменів без передачі аутентифікаційної інформації.

image.src = url; - Задає URL зображення для завантаження.

image.onload = () => {...}; - Встановлює обробник події, який викликається, коли зображення успішно завантажено. У внутрішній частині обробника відбувається повторна прив'язка текстури і встановлення текстурного зображення за допомогою texImage2D. Після цього викликається функція draw(), яка, ймовірно, відповідає за відображення сцени з завантаженою текстурою.



Рисунок 2 – Обрана текстура

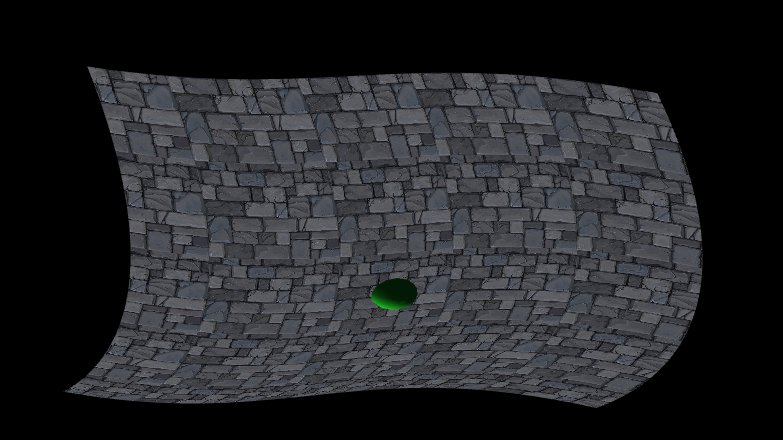


Рисунок 3 – Текстура накладена на поверхню та точка

У відповідності до варіанту було втілено можливість масштабування текстури щодо точки на поверхні.

**Інструкція користувача**

Для обертання поверхні відносно її центру слід натискати ліву клавішу миші та проводити рух у напрямку бажаного обертання. Точка на створеній поверхні рухається так само, як і задана поверхня.

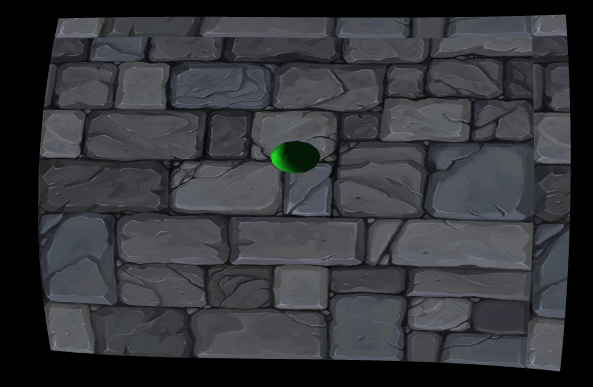
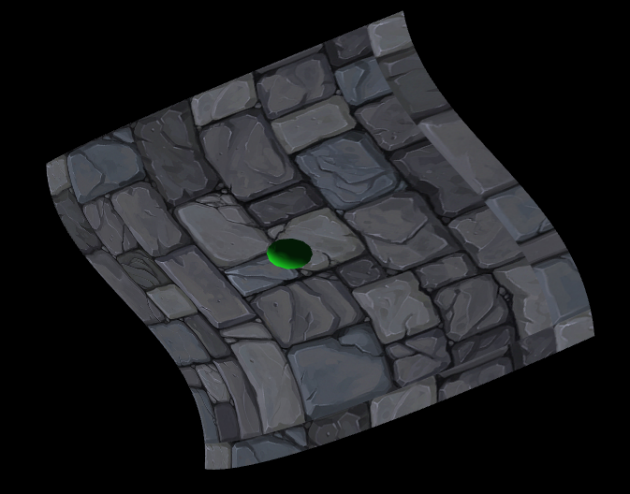
 

Рисунок 4 – Обертання заданої поверхні

Змінювати масштаб заданої текстури можна за допомогою зміни вхідного діапазону під назвою Texture Scaling. При переміщенні повзунка ліворуч, масштабування текстури зменшується, при переміщенні праворуч - збільшується. Найменше значення масштабування - 1, найбільше - 10.

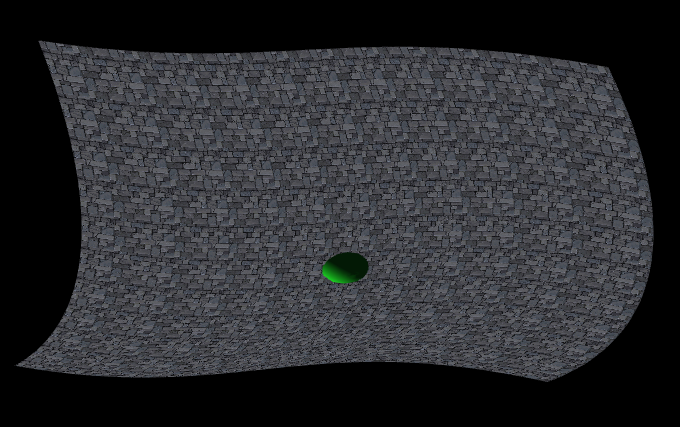
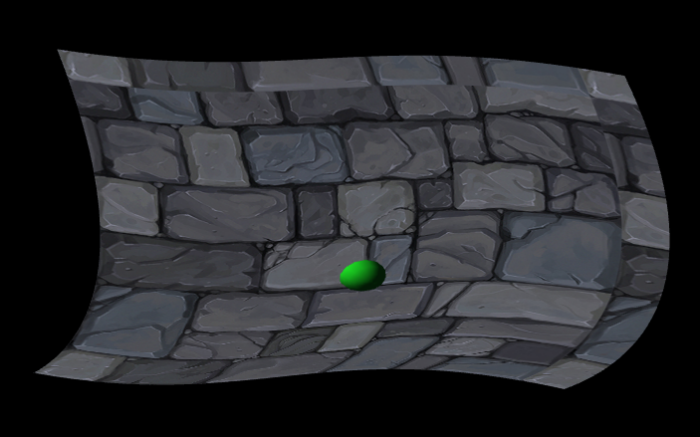


Рисунок 5 – Масштабування текстури

Точку, відносно якої здійснюється обертання, можна переміщувати за допомогою клавіш WASD. Кожне натискання клавіші призводить до зміщення точки на певний визначений крок по поверхні. Переміщення відбувається до досягнення визначеної межі.

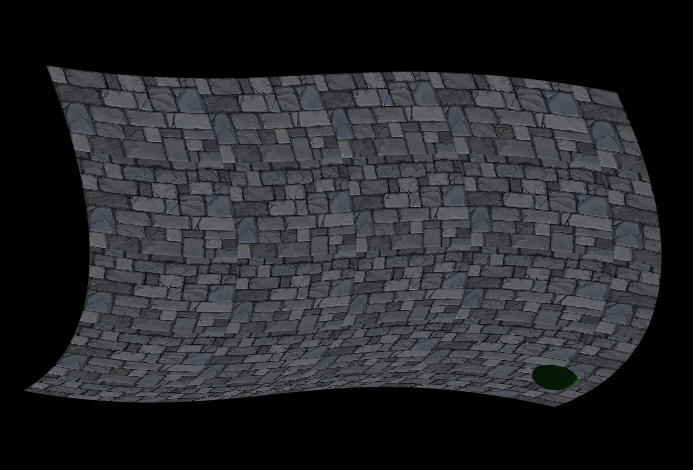


Рисунок 6 - Зміна положення точки

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

**Зразок вихідного коду програми**

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 = "anonymous";

  image.src = url;

  image.onload = () => {

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

    gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE, image);

    draw();

  };

}

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec3 normal;

attribute vec2 texture;

uniform mat4 ModelViewProjectionMatrix;

uniform mat4 TranslationMatrix;

uniform mat4 NormalMatrix;

varying vec4 color;

uniform bool lighting;

uniform vec3 lightPos;

uniform vec2 pointPos;

uniform float scaleFactor;

varying vec2 varTexture;

uniform vec3 pointTranslation;

vec3 ambientColor = vec3(0.01, 0.1, 0.02);

vec3 diffuseColor = vec3(0.11, 0.8, 0.13);

vec3 specularColor = vec3(0.0, 0.0, 0.0);

float shininess = 80.0;

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

    return mat4 (

        1.0, 0.0, 0.0, 0.0,

        0.0, 1.0, 0.0, 0.0,

        0.0, 0.0, 1.0, 0.0,

        tx,  ty,  tz, 1.0

    );

}

mat4 scaling(float s){

    return mat4 (

        s, 0.0, 0.0, 0.0,

        0.0, s, 0.0, 0.0,

        0.0, 0.0, s, 0.0,

        0.0,  0.0,  0.0, 1.0

    );

}

void main() {

    vec4 tranlatedP = translation(-pointPos.x, -pointPos.y, 0.0) \* vec4(texture, 0., 1.);

    vec4 scaledP = scaling(scaleFactor) \* tranlatedP;

    vec4 finalTransformedP = translation(pointPos.x, pointPos.y, 0.0) \* scaledP;

    varTexture = vec2(finalTransformedP.x,finalTransformedP.y);

    vec4 vertPos4 = ModelViewProjectionMatrix \* vec4(vertex, 1.0);

    vec3 vertPos = vec3(vertPos4) / vertPos4.w;

    vec4 norm = NormalMatrix \* vec4(normal,1.0);

    vec3 N = normalize(vec3(norm));

    vec3 L = normalize(lightPos - vertPos);

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

    float specular = 0.5;

    vec3 R = reflect(-L, N);

    vec3 V = normalize(-vertPos);

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

    specular = pow(specAngle, 80.0);

    color = vec4(ambientColor + lambertian \* diffuseColor + specular \* specularColor, 1.0);

    gl\_Position = vertPos4;

    if(lighting){

        gl\_Position = vertPos4;

    }

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

   precision highp float;

#else

   precision mediump float;

#endif

varying vec2 varTexture;

varying vec4 color;

uniform bool lighting;

uniform sampler2D tmu;

void main() {

    gl\_FragColor = texture2D(tmu,varTexture);

        if(lighting) {

        gl\_FragColor = color;

    }

}`;