**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 12

Виконав студент групи ТР-31мп

Кравчук Дмитро Володимирович

Київ - 2023

1. **Завдання**
2. Нанести текстуру на поверхню з практичного завдання №2.
3. Реалізувати масштабування/обертання текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання.
4. 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.
5. **Теорія**

**WebGL**

WebGL (Web Graphics Library) - це технологія, яка дозволяє вбудовувати 3D та 2D графіку у веб-браузери без необхідності встановлення додаткових плагінів. Вона базується на OpenGL ES (OpenGL for Embedded Systems) і забезпечує взаємодію між JavaScript та графічними прискорювачами, що є частиною веб-сторінки.

* WebGL використовує шейдери для обробки вершин і фрагментів, надаючи можливість задавати операції з текстурними координатами та здійснювати потрібні перетворення.
* Також, в рамках WebGL можна створювати та використовувати текстурні об'єкти для конкретних текстурних одиниць у шейдерах.
* Додатково, матричні перетворення використовуються для контролю положення, масштабування та обертання 3D-моделей.

**Шейдери WebGL**

WebGL — це потужний інструмент, який використовує шейдери для створення 3D-графіки у веб-середовищі. Шейдери мають ключове значення в процесі відтворення, надаючи можливість гнучкої настройки вигляду графіки.

У WebGL існують два основних типи шейдерів: вершинні та фрагментні. Вершинні шейдери обробляють кожну вершину 3D-моделі перед її відображенням. Вони відповідають за трансформації позицій вершин та передачу оброблених даних фрагментному шейдеру. Фрагментні шейдери, у свою чергу, працюють з кожним пікселем, що відображається на екрані, визначаючи його кінцевий колір на основі текстур, освітлення та інших ефектів.

Мова GLSL використовується для написання обох типів шейдерів. Їх потрібно компілювати та об'єднувати в програми шейдерів перед використанням. Під час відтворення, WebGL використовує вказані програми шейдерів для відображення графіки.

Uniforms використовуються для передачі постійних значень у шейдери, наприклад, матриць трансформації.

Attributes визначають дані, які відрізняються між вершинами, такі як їх положення чи координати текстур.

Координати текстур зазвичай передаються як атрибути у вершинний шейдер та інтерполюються для використання у фрагментному шейдері. Останній використовує ці координати для вибірки кольорів із текстур, визначаючи остаточний колір кожного пікселя на екрані.

**Накладання текстур. Текстурні координати**

Техніка відображення текстур у комп’ютерній графіці дозволяє наносити зображення або текстури на 3D-моделі, щоб створити реалістичні поверхні. У WebGL це досягається шляхом прив'язки кожної вершини 3D-об'єкта до координат текстури (u, v), які використовуються для вибірки кольорів із зображення текстури. Це значно поліпшує зовнішній вигляд об’єкта, надаючи йому деталізовану поверхню.

Координати текстури (u, v) відіграють ключову роль, визначаючи, як саме текстура накладається на поверхню. Вони знаходяться в межах від 0 до 1 і використовуються для вказівки точних місць на зображенні текстури. У WebGL ці координати пов’язані з кожною вершиною 3D-моделі та інтерполюються по всій поверхні в процесі відтворення. Це гарантує правильне нанесення текстури на всю модель, забезпечуючи її адекватне відображення.

1. **Деталі розробки**

Для роботи обрано поверхню Монжа з циліндричною поверхнею прямих і синусоїдою як меридіаном, згідно з моїм варіантом 12. Програма яка її виводить реалізована у практичному завданні №2 (Рисунок 1). Поверхня виводиться у вигляді суцільних трикутників.

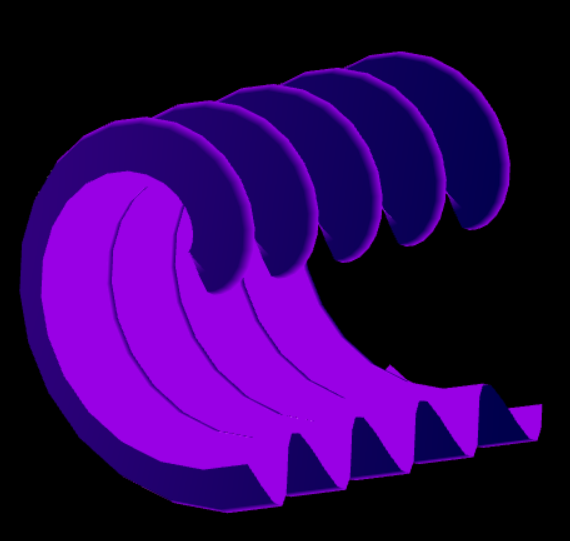


Рисунок 1 – Вигляд поверхні

Для нанесення на фігуру обрано текстуру, яка має вигляд води (Рисунок 2). Для того щоб була краща якість на різних операційних системах та браузерах обрано зображення розміром 512х512 пікселів та із розширенням .jpg.



Рисунок 2 – Зображення текстури

Для накладання текстури на поверхню (Рисунок 3) був створений буфер текстурних координат, де кожна координата відповідає вершині у буфері вершин. Згідно з варіантом, текстура мала обертатися, тому був визначений відповідний параметр uniform для встановлення кута обертання. У програмі шейдера для визначення кольору пікселя фігури, на яку накладено текстуру, використовувалася функція texture2D. Ця функція отримує об'єкт класу sampler2D як перший аргумент, який зберігає дані про зображення, та другим аргументом — текстурні координати.

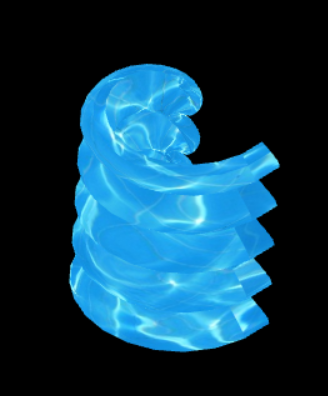
.

Рисунок 3 – Текстура накладена на поверхню

Створено новий об'єкт класу Model, який використовується для відображення точки, відносно якої застосовується трансформація текстури. Графічно цей об'єкт представлений у вигляді сфери, яка розташована на поверхні (Рисунок 4).

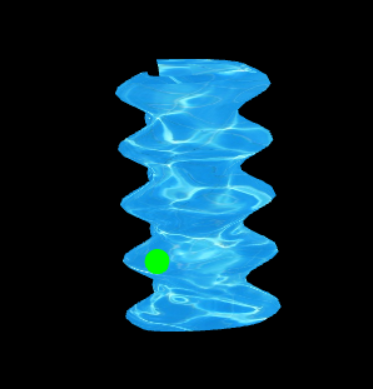


Рисунок 4 – Точка на текстурованій поверхні

Оскільки мій варіант парний (12), то реалізовано обертання текстури відносно точки на поверхні.

1. **Інструкції користувача**

Для обертання фігури (Рисунок 5) використовується ліва кнопка миші: утримуючи її, можна перетягнути фігуру у бажаному напрямку для обертання навколо центру.

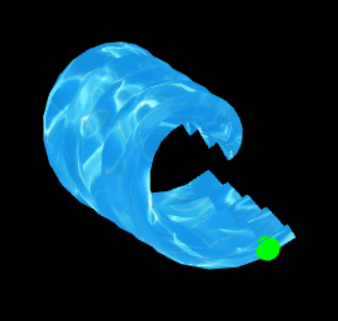


Рисунок 5 – Вигляд фігури до та після обертання

Для переміщення точки, навколо якої здійснюється обертання, використовуються клавіші W, A, S та D. Кожне натискання клавіші переміщує точку по поверхні на певний визначений крок (Рисунок 6). Рух здійснюється до досягнення встановленої межі.

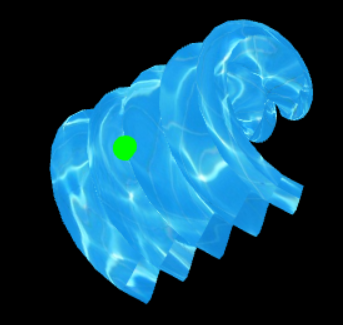
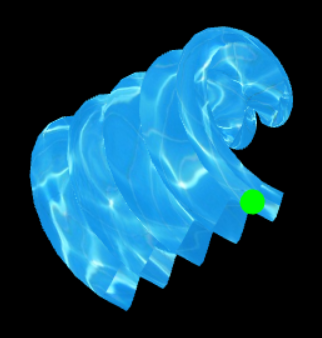


Рисунок 6 – Демонстрація переміщення точки відносно нерухомої фігури

Слайдер з назвою "angle of rotation" дозволяє користувачу змінювати кут, навколо якого відбувається обертання на поверхні. Якщо вибране значення відрізняється від нуля, спостерігається зсув текстури під час переміщення точки. Це стає можливим через обертання навколо іншої точки на поверхні, що відповідає іншим координатам текстури. Такий підхід дає можливість відтворювати зміни в текстурі при зміні кута обертання (Рисунок 7).

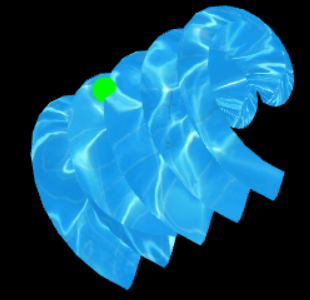
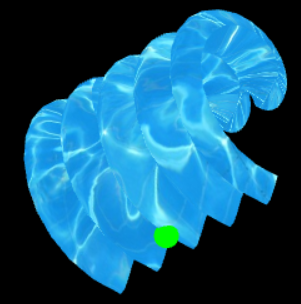


Рисунок 7 – Демонстрація зміни кута обертання текстури

При перезавантаженні сторінки, значення кута обертання та положення точки на поверхні будуть скинуті до їхніх значень за замовчуванням. Це означає, що обертання фігури та розташування точки повернуться до початкового стану при перезавантаженні сторінки.

1. **Код програми**

Частина коду на мові JavaScript

function draw() {

webGLCont.clearColor(0, 0, 0, 1);

webGLCont.clear(webGLCont.COLOR\_BUFFER\_BIT | webGLCont.DEPTH\_BUFFER\_BIT);

const projectionTransformation = 40;

let projection = m4.orthographic(-projectionTransformation, projectionTransformation,

-projectionTransformation, projectionTransformation,

-projectionTransformation, projectionTransformation);

let modelView = simpleRotator.getViewMatrix();

let rotateToPointZero = m4.axisRotation([0.707, 0.707, 1], 0.6);

let translateToPointZero = m4.translation(-1, 1.5, -17);

let matAccum0 = m4.multiply(rotateToPointZero, modelView);

let matAccum1 = m4.multiply(translateToPointZero, matAccum0);

let modelViewProjection = m4.multiply(projection, matAccum1);

let normMatr = m4.identity();

m4.inverse(modelView, normMatr);

normMatr = m4.transpose(normMatr, normMatr);

webGLCont.uniformMatrix4fv(shaderProgram.iModelViewProjectionMatrix, false, modelViewProjection);

webGLCont.uniformMatrix4fv(shaderProgram.iNormalMatrix, false, normMatr);

webGLCont.uniform4fv(shaderProgram.iColor, [0.6, 0, 0.9, 1]);

webGLCont.uniform1f(shaderProgram.iLimit, 0);

webGLCont.uniform1f(shaderProgram.iSmoothValue, 0);

webGLCont.uniform1f(shaderProgram.iAngle, parseFloat(document.getElementById('angle').value));

webGLCont.uniform2fv(shaderProgram.iTxTr, txTr);

surface.Draw();

webGLCont.uniform1f(shaderProgram.iSmoothValue, -1000);

let aMax = 360;

let tMax = 20;

let aStep = 15;

let tStep = 0.5;

let r = 1.4;

let c = 1.4;

let d = 1.4;

let phi = 3.14 / 2;

let alpha0 = 0;

let pi = 3.14;

let surfaceFormula = (a, t) => {

let x = r \* Math.cos(degreeToRad(a)) - (r \* (degreeToRad(alpha0) - degreeToRad(a)) + t \* Math.cos(phi)

- c \* Math.sin(d \* t) \* Math.sin(phi)) \* Math.sin(degreeToRad(a));

let y = r \* Math.sin(degreeToRad(a)) + (r \* (degreeToRad(alpha0) - degreeToRad(a)) + t \* Math.cos(phi)

- c \* Math.sin(d \* t) \* Math.sin(phi)) \* Math.cos(degreeToRad(a));

let z = t \* Math.sin(phi) + c \* Math.sin(degreeToRad(d \* t)) \* Math.cos(phi);

return [x, y, z];

}

webGLCont.uniformMatrix4fv(shaderProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection, m4.translation(

...surfaceFormula(txTr[0] \* aMax, txTr[1] \* tMax))));

sphere.Draw()

}

Частина коду програми на мові GLSL

void main() {

vec4 txTr1 = translation(vec3(-txTr,0.0))\*vec4(tx,0.,1.);

vec4 txTr2 = rotation(angle)\*txTr1;

vec4 txTr3 = translation(vec3(txTr,0.0))\*txTr2;

texCoord = txTr3.xy;

vec3 normalVector = normalize(mat3(MatrixOfNormals)\*nm);

vec3 vertexVector = mat3(MatrixOfViewProjection)\*vt;

vec3 vectorToLightPosition = positionOfLight-vertexVector;

vec3 vectorToViewPosition = -vertexVector;

vec3 halfVector = normalize(vectorToLightPosition + vectorToViewPosition);

float prodLDandLS = dot(vectorToLightPosition, normalize(-directionOfLight));

float smoothTransitionU\_limitAndU\_smoothValue = smoothstep(limit, limit + smooth, prodLDandLS);

float mirroringLight = smoothTransitionU\_limitAndU\_smoothValue \* pow(dot(normalVector, halfVector), 1.0);

normalInterp = vec3(MatrixOfNormals \* vec4(nm, 0.0));

vec3 normalToSurface = normalize(normalInterp);

float lambertianCos = max(dot(normalToSurface, vectorToLightPosition), 0.0);

if (lambertianCos > 0.0) {

vec3 lightVector = reflect(-vectorToLightPosition, normalToSurface);

vec3 vectorToObserver = normalize(-verticalPosition);

float mirroringAngle = max(dot(lightVector, vectorToObserver), 0.0);

mirroringLight = pow(mirroringAngle, 10.0);

}

fragmentColor = vec3(figureColor) + lambertianCos\*vec3(0.0, 0.0, 0.0) + mirroringLight\*vec3(1.0);

gl\_Position = MatrixOfViewProjection \* vec4(vt, 1.0);

}