**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”

Виконав студент групи ТР-31мп

Кривов’язюк Михайло Олексійович

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 є інтерфейсом програмування за допомогою JavaScript для відтворення динамічної 2D- та 3D-графіки у веб-браузерах. Для втілення функцій, таких як відображення текстур, масштабування, обертання і інтерактивне переміщення точок, API WebGL надає можливості для обробки шейдерів, текстур та матричних перетворень.

* Шейдери використовуються для визначення обробки вершин і фрагментів, дозволяючи включати текстурні координати та здійснювати перетворення.
* Текстурні об'єкти створюються та пов'язуються з певними текстурними одиницями для використання в шейдерах.
* Матричні перетворення використовуються для керування положенням, масштабуванням і обертанням 3D-моделі.

Шейдери WebGL

У WebGL шейдери відіграють ключову роль у візуалізаційному конвеєрі, забезпечуючи гнучкий та програмований метод визначення вигляду 3D-графіки. В цьому контексті існують два основних види шейдерів: вершинні та фрагментні.

Вершинні шейдери відповідають за обробку кожної вершини 3D-моделі перед її відображенням на екрані. Вони опрацьовують атрибути вершин, такі як положення, колір і координати текстури, створюючи вихідні дані. Широко використовується застосування трансформацій до позицій вершин, таких як переміщення, обертання та масштабування, для розташування об'єктів у 3D-просторі. Вихід вершинного шейдера містить трансформовану позицію вершини та інтерпольовані значення, які передаються фрагментному шейдеру.

Фрагментні шейдери, часто відомі як піксельні, опрацьовують кожен піксель, що буде відображений на екрані. Вони отримують інтерпольовані значення від вершинного шейдера, такі як координати кольору та текстури, а також інші дані, наприклад, щодо освітлення. Завданням фрагментного шейдера є визначення кінцевого кольору для кожного пікселя, що може включати в себе вибірку текстур, розрахунок освітлення та інші ефекти. Остаточний колір використовується для відображення пікселя на екрані.

Як вершинний, так і фрагментний шейдери написані мовою GLSL (OpenGL Shading Language) і потребують компіляції перед їх використанням. Після компіляції ці шейдери об'єднуються в програму шейдера, яка повинна бути зв'язана перед відтворенням, надаючи WebGL можливість використовувати вказані вершинні та фрагментні шейдери у конвеєрі рендерингу.

Uniforms представляють собою значення, які залишаються постійними для всіх вершин або фрагментів під час візуалізації примітиву. Вони дозволяють передавати зовнішні дані у шейдери, такі як матриці трансформації чи інформація про глобальне освітлення.

Attributes представляють дані для кожної вершини, які варіюються між вершинами. Вони використовуються для передачі інформації, такої як положення вершин, їхні нормали 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 ці координати використовуються для вибірки кольорів із текстур, які визначають остаточний колір кожного пікселя.

Накладання текстур. Текстурні координати

Відображення текстури - це метод у комп'ютерній графіці, який використовується для створення реалістичних поверхонь шляхом накладання зображень або текстур на 3D-моделі. У контексті WebGL цей процес передбачає прив'язку кожної вершини 3D-об'єкта до координат текстури (u, v), які потім використовуються для вибірки кольорів із текстурного зображення. Це сприяє покращенню візуального вигляду об'єкта, дозволяючи детально передати інформацію про його поверхню.

Координати текстури (u, v) є ключовими параметрами, що визначають спосіб накладення текстури на поверхню. Змінюючись від 0 до 1, вони використовуються для вказівки конкретних точок на текстурному зображенні. 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.

1. **Деталі розробки**

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

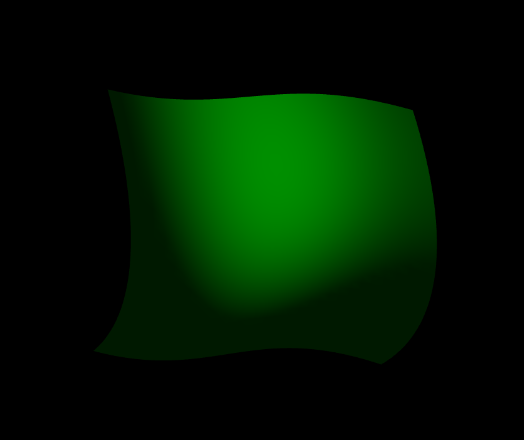


Рисунок 1 – Вигляд поверхні

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.

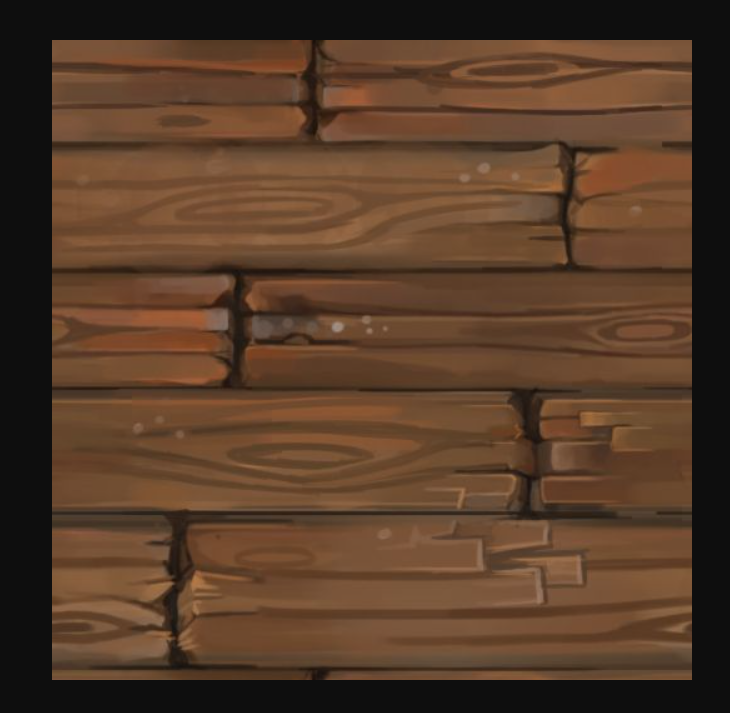


Рисунок 2 – Зображення текстури

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 масштабуватися, тому було створено відповідний uniform, який визначатиме фактор масштабування. 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.

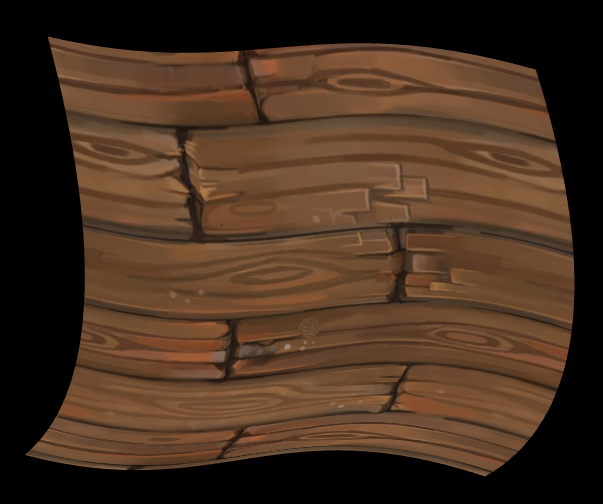


Рисунок 3 – Текстура накладена на поврехню

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

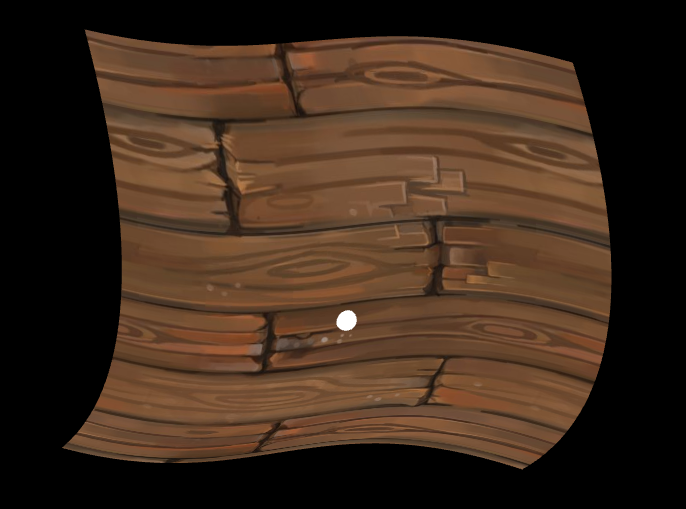


Рисунок 4 – Точка на текстурованій поверхні

Згідно варіанту було імплементовано масштабування текстури відносно точки на поверхні.

1. **Інструкції користувача**

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

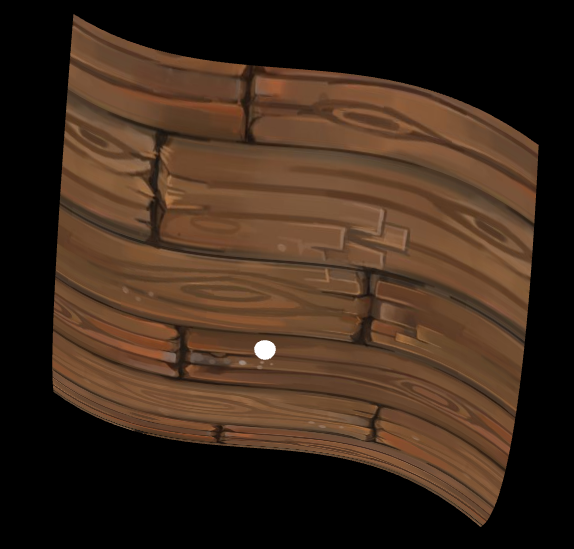
 

Рисунок 5 – Вигляд фігури до та після обертання

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.

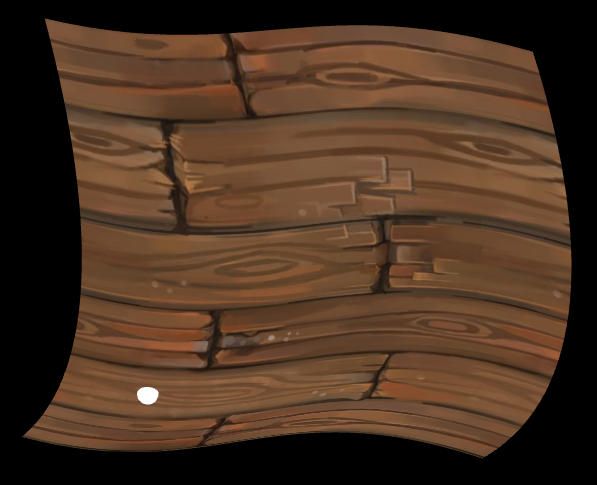
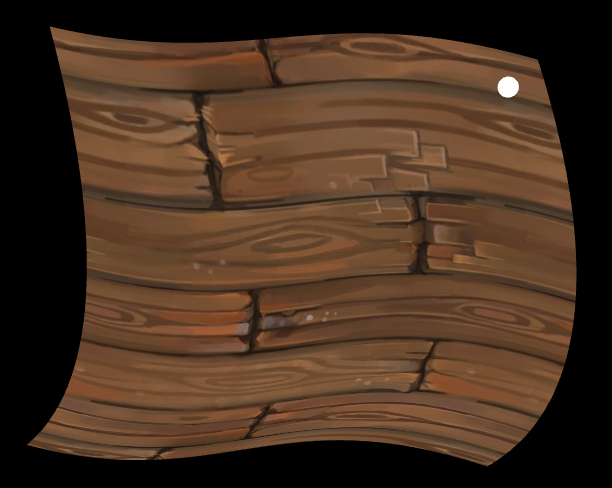
 

Рисунок 6 – Демонстрація переміщення точки відносно нерухомої фігури

За допомогою слайдера з підписом “Величина масштабування” можна змінювати фактор масштабування відносно умовної точки на поверхні. Можна помітити, що при переміщенні точки зміщується і текстура, адже масштабування відбувається відносно іншої точки на поверхні , яка в свою чергу відповідає іншій текстурній координаті.

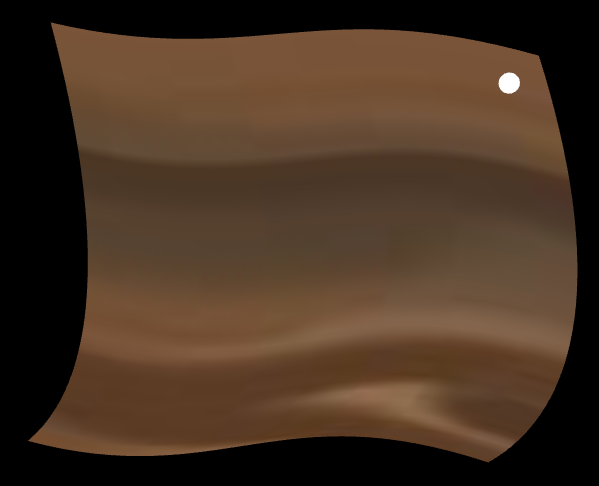
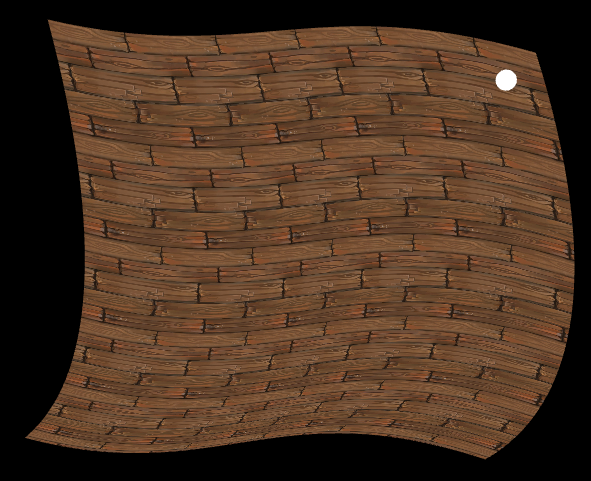
 

Рисунок 7 - Демонстрація зміни кута масштабування текстури

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.

1. **Код програми**

Код функції частини програми 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.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);

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

    const normal = m4.identity();

    m4.inverse(modelView, normal);

    m4.transpose(normal, normal);

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

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

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

    let shoe = function (a, b) {

        return (a \* a \* a) / 3 - (b \* b) / 2;

    }

    gl.uniformMatrix4fv(shProgram.iTranslationMatrix, false, m4.translation(...pointPos, shoe(...pointPos)));

    gl.uniform3fv(shProgram.iLightPosition, [0.5 \* Math.cos(Date.now() \* 0.001), 0.5 \* Math.sin(Date.now() \* 0.001), 0]);

    gl.uniform2fv(shProgram.iPointPos, [map(pointPos[0],x\_min,x\_max,0,1),map(pointPos[1],y\_min,y\_max,0,1)]);

    gl.uniform1f(shProgram.iScale, parseFloat(document.getElementById("scale\_Slider").value));

    gl.uniform3fv(shProgram.iPointTranslation, [...pointPos, shoe(...pointPos)])

*// console.log(...pointPos,shoe(...pointPos))*

    surface.Draw();

    gl.uniformMatrix4fv(shProgram.iModelViewProjectionMatrix, false, m4.multiply(modelViewProjection, m4.translation(...pointPos, shoe(...pointPos))));

    gl.uniform1i(shProgram.iLighting, true);

    lighting.Draw();

    gl.uniform1i(shProgram.iLighting, false);

}

Код функції частини програми GLSL

void main() {

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

vec4 sc = scaling(scaleFactor)\*s;

vec4 scl = translation(pointPos.x,pointPos.y,0.0)\*sc;

varTexture = vec2(scl.x,scl.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);

// 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(-vertPos); // Vector to viewer

// Compute the specular term

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

specular = pow(specAngle, 80.0);

}

vec3 ambientColor = vec3(0.0,0.1,0.0);

vec3 diffuseColor = vec3(0.0,0.7,0.0);

vec3 specularColor = vec3(0.0,1.0,0.0);;

varColor = vec3(ambientColor +

lambertian \* diffuseColor +

specular \* specularColor);

gl\_Position = vertPos4;

if(lighting){

gl\_Position = vertPos4;

}

}