Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут ім. Ігоря Сікорського»
Інститут атомної та теплової енергетики
Кафедра цифрових технологій в енергетиці

Розрахунково-графічна робота

З дисципліни «Візуалізація графічної та геометричної інформації»
Варіант 7

Виконав:Дроздюк Олександр
Студент групи ТР-31мп

Київ 2023

Завдання

Тема роботи: Операціїї над тектурними координатами

Вимоги:

- Накласти текступу на поверхню отриману в результаті виконання лабораторної роботи №2.
- Імплементувати масштабування або обертання текстури(текстурних координат) згідно з варіантом: непарні - масштабування, парні - обертання.
- Запровадити можливість переміщення точки відносно якої відбувається трансформація текстури по поверхні за рахунок зміни параметрів в просторі текстури. Наприклад, клавіші A та D для переміщення по осі абсцис, змінюючи параметр u текстури, а клавіші W та S по осі ординат, змінюючи параметр v.

Теоретичні відомості

Карта текстур - це двовимірний файл зображення, який можна застосувати до поверхні 3D-моделі, щоб додати колір, текстуру або інші деталі поверхні, такі як блиск, відбивна здатність або прозорість. Текстурні карти розробляються для безпосередньої відповідності UV-координатам розгорнутої 3D-моделі і створюються на основі реальних фотографій або малюються вручну в графічних програмах, таких як Photoshop або Corel Painter.

Текстурні карти зазвичай малюються безпосередньо поверх UV-розкладки моделі, яку можна експортувати у вигляді квадратного растрового зображення з будь-якого програмного забезпечення для роботи з 3D. Художники-текстувальники зазвичай працюють з багатошаровими файлами, де UV-координати нанесені на напівпрозорий шар, який художник використовує як орієнтир для розміщення певних деталей.

Як випливає з назви, найочевидніше використання текстурної карти - це додавання кольору або текстури на поверхню моделі. Це може бути як просте застосування текстури текстури дерева до поверхні столу, так і складне, як карта кольорів для всього ігрового персонажа (включно з обладунками та аксесуарами).

Однак, термін "карта текстур", як його часто використовують, є дещо неправильним - карти поверхонь відіграють величезну роль у комп'ютерній графіці, не обмежуючись лише кольором і текстурою. У виробничих умовах карта кольору персонажа або середовища зазвичай є лише однією з трьох карт, які використовуються майже для кожної окремої 3D-моделі.

Інші два "основні" типи карт - це дзеркальні карти та карти рельєфу/зміщення, або нормальні карти.

Дзеркальні карти (також відомі як карти глянцю). Дзеркальна карта вказує програмі, які частини моделі мають бути блискучими або глянцевими, а також величину блиску. Свою назву дзеркальні карти отримали через те, що блискучі поверхні, такі як метали, кераміка і деякі пластмаси, мають сильний дзеркальний відблиск (пряме віддзеркалення від сильного джерела світла). Якщо ви не впевнені щодо дзеркальних відблисків, подивіться на біле відображення на обідку вашої кавової чашки. Іншим поширеним прикладом дзеркального відображення є крихітний білий відблиск у чиємусь оці, трохи вище зіниці.

Дзеркальна карта, як правило, є зображенням у відтінках сірого і є абсолютно необхідною для поверхонь, які не є рівномірно глянцевими. Наприклад, бронемашина потребує дзеркальної карти для того, щоб подряпини, вм'ятини та дефекти броні виглядали переконливо. Аналогічно, ігровий персонаж, виготовлений з різних матеріалів, потребує дзеркальної карти, щоб передати різні рівні блиску між шкірою персонажа, металевою пряжкою ременя та матеріалом одягу.

Трохи складніші, ніж будь-який з двох попередніх прикладів, карти рельєфу - це тип текстурної карти, який може допомогти більш реалістично відобразити нерівності або заглиблення на поверхні моделі.

Розглянемо цегляну стіну: Зображення цегляної стіни можна зіставити з плоскою багатокутною площиною і назвати його готовим, але, швидше за все, воно не буде виглядати дуже переконливо у фінальному рендері. Це тому, що пласка площина не реагує на світло так, як цегляна стіна з її тріщинами і грубістю.

Щоб посилити враження реалістичності, можна було б додати бамп або карту нормалей, щоб точніше відтворити грубу, зернисту поверхню цегли і посилити ілюзію того, що тріщини між цеглинами насправді зменшуються в просторі. Звичайно, можна було б досягти такого ж ефекту, моделюючи кожну цеглину вручну, але нормальна площина набагато ефективніша з точки зору обчислень. Неможливо переоцінити важливість нормального мапування в сучасній ігровій індустрії - ігри просто не могли б виглядати так, як вони виглядають сьогодні без нормальних карт.

Бамп, зміщення та нормальні карти - це окрема тема для обговорення, але вони є абсолютно необхідними для досягнення фотореалізму в рендері. Чекайте на статтю, в якій ми докладно розповімо про них.

Виконання завдання

В ході другої лабораторної роботи було створено поверхню під назвою «Clover Knot». Отриману поверхню з освітленням можна побачити на рисунку 3.1.

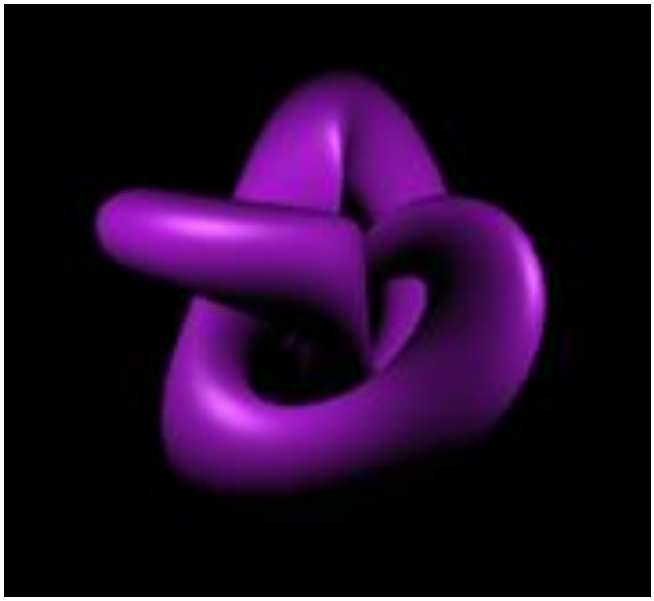

Рис. 3.1 «Clover Knot» з освітленням

Текстура була завантажена як картинка з інтернету формату «jpg». Вона була завантажена на github, щоб в подальшому використовувати посилання на неї і не стикатися з проблемою Cross-Origin Resource Sharing policy.

В графічному редакторі було налаштувано розмір картинки так, щоб ширина і висота були рівні, а також, аби сторона мала розмір 2^n в пікселях.

З метою накладання текстури на поверхню, в першу чергу було створено декілька змінних в коді шейдера. Після чого були створення

посилання на них в коді програми. Були також створені функції для генерації даних текстури.

Обрану картинку можна побачити на рисунку 3.2.

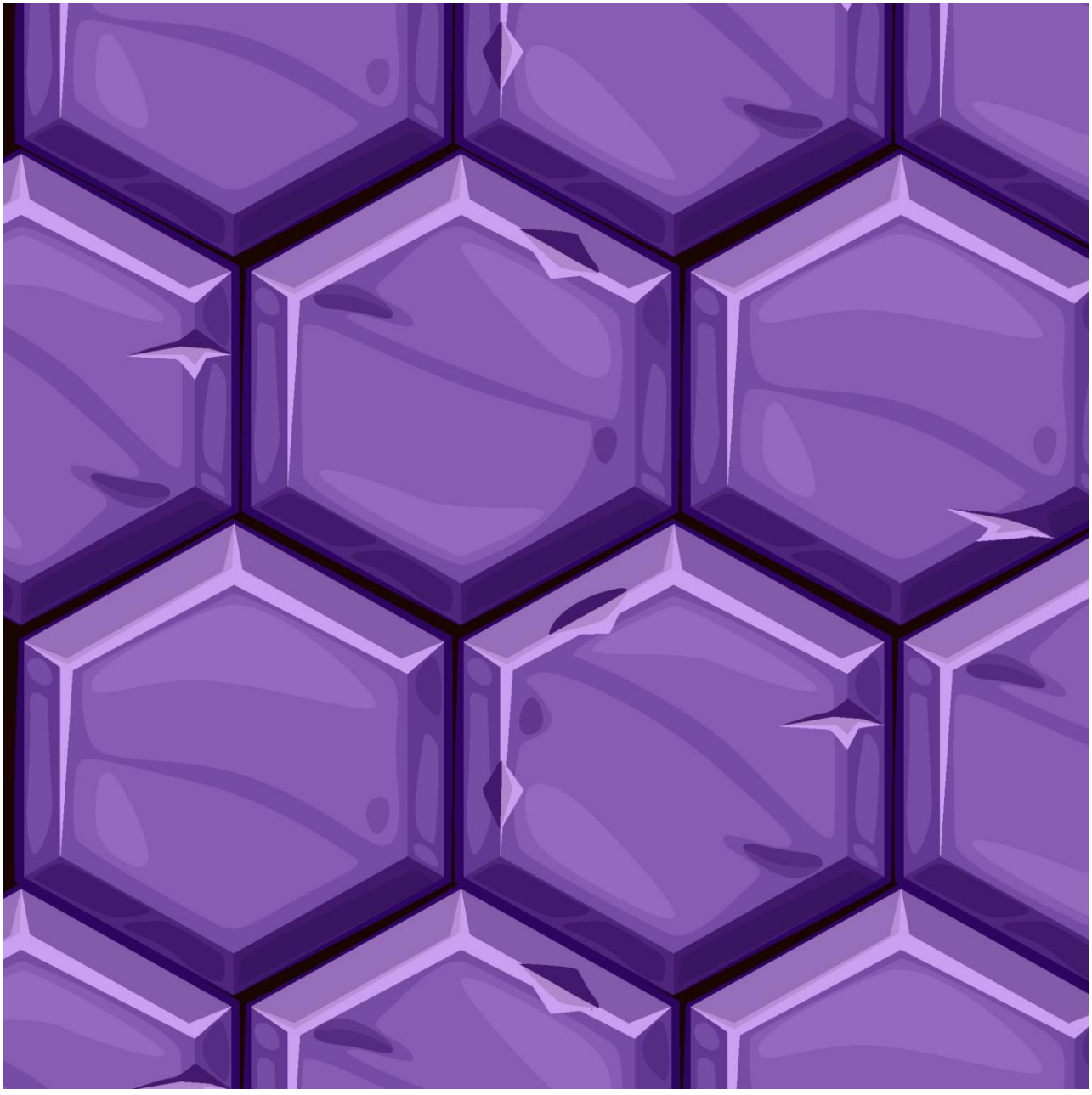

Рис. 3.2 Обрана текстура

Поверхню з накладеною текстурою можна побачити на рисунку 3.3.

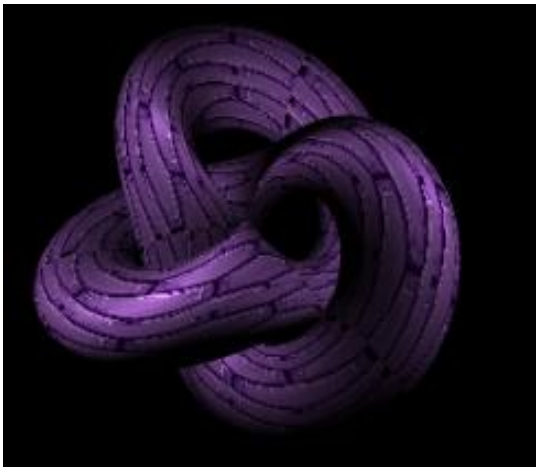

Рис. 3.3 «Corrugated Sphere» з накладеною текстурою

За допомогою шейдера була створена динамічна точка. Поверхню з точкою можна побачити на рис 3.4

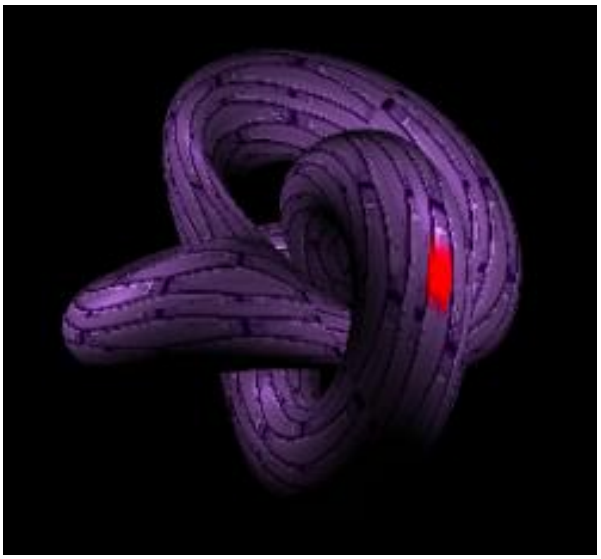

Рис. 3.4 «Corrugated Sphere» з накладеною текстурою та

Для роботи з текстурою було створено ще кілька змінних в коді шейдера:

обертання текстури, розташування умовної точки в (u,v) координатах, змінну для розташування сфери на відповідне місце поверхні в 3д-просторі.

Вказівки користувачу

Була додана можливість для користувача, керувати переміщенням умовних точок по поверхні, масштабування текстур щодо умовних точок і орієнтацією поверхні в просторі. Останні два пункти виконуються таким же чином.

Переміщення умовної точки реалізовано за допомогою введення з клавіатури(рисунок 4.1): клавіші W та S здійснюють переміщення точки за параметром v в додатньому та від’ємному напрамках відповідно, клавіші A та D здійснюють переміщення точки за параметром u у від’ємному та додатньому напрямках відповідно.

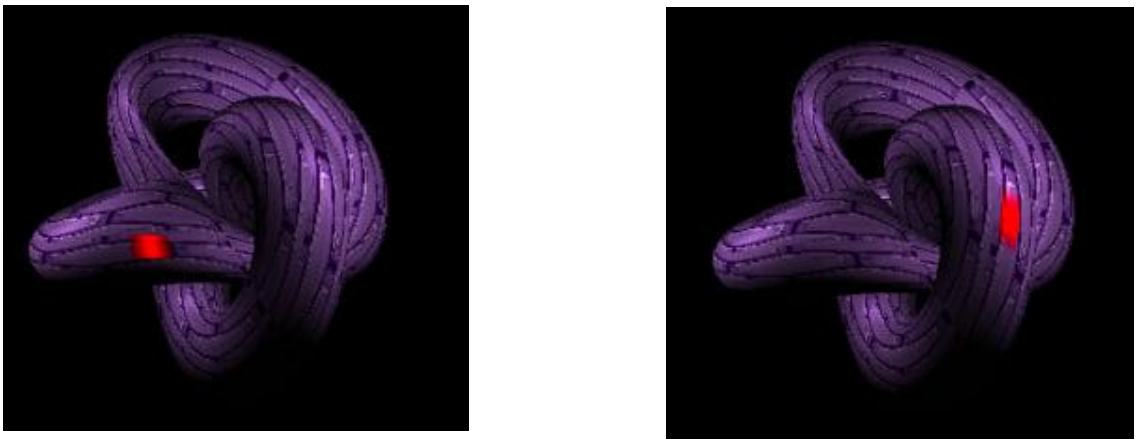

Рис. 4.1. Переміщення умовної точки

Орієнтація поверхні в просторі здійснюється за допомогою миші. На рисунку 4.2 можна помітити що точка та текстура залишились на одному і тому самому місці відносно поверхні. Змінилась лише орієнтації поверхні в просторі.

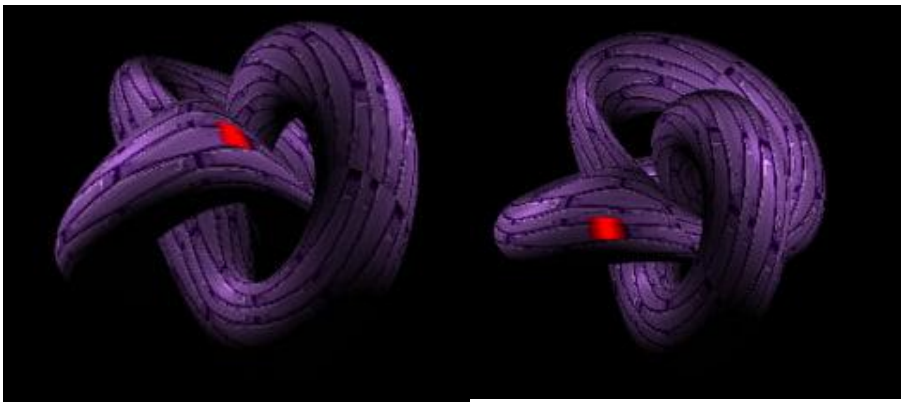

Рис. 4.2. Зміна орієнтація поверхні в просторі

Для зміни масштабу текстури на сторінку додане поле зображений на рис. 4.3.

Рис. 4.3. Поле зміни масштабу текстури

Масштабування текстури 1x та 2x зображено на рисунку 4.4

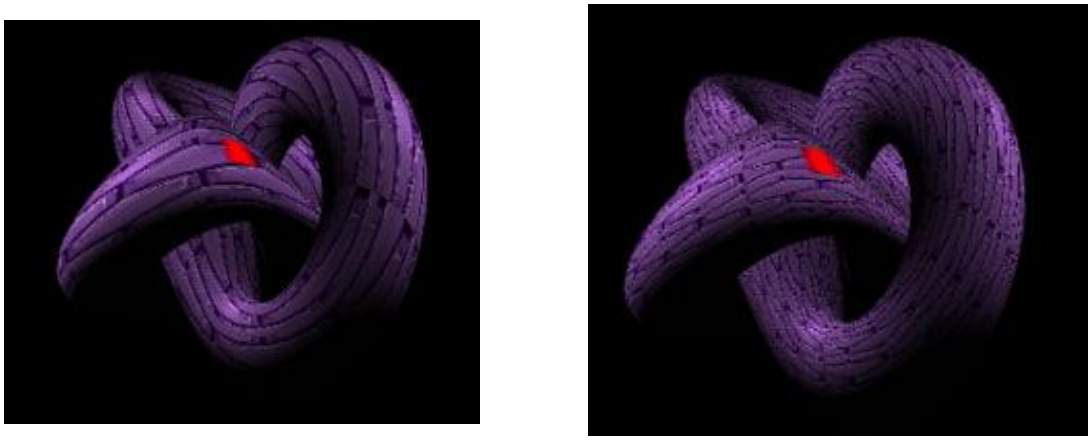

Рис. 4.4. Масштаб текстури

Висновок

В даній розрахунковій роботі ми дослідили, що таке текстурування об'єкту, а також вивчили, що таке розгортка та UV-mapping. Було реалізовано обертання текстури навколо визначеної користувачем точки. Також є можливість переміщати точку вздовж поверхні. Матеріал засвоєний.

Код шейдерів:

```
// Vertex shader
const vertexShaderSource = `
attribute vec3 vertex;
attribute vec3 normal;
attribute vec2 textureCoord;
uniform mat4 WorldInverseTranspose;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 pointPosition;

varying vec3 vertexVarying;
varying vec3 normalVarying;
varying vec2 v_texcoord;
varying vec4 pointColor;

void main() {
    normalVarying = (WorldInverseTranspose * vec4(normalize(normal), 0.0)).xyz;

    pointColor = vec4(0.0, 0.0, 0.0, 0.0);
    if(distance(pointPosition, vertex) < 0.1){
        pointColor.x = 1.0;
        pointColor.y = -1.0;
        pointColor.z = -1.0;
    }

    vec4 worldVertex = ModelViewProjectionMatrix * vec4(vertex,1.0);
    vertexVarying = worldVertex.xyz;
    v_texcoord = textureCoord;
    gl_Position = worldVertex;
}`;


// Fragment shader
const fragmentShaderSource = `
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif

uniform vec3 LightPosition;
uniform vec3 CamWorldPosition;

uniform vec3 lightAmbientColor;
uniform vec3 lightDiffuseColor;
uniform vec3 lightSpecularColor;
```

```
uniform vec3 matAmbientColor;
uniform vec3 matDiffuseColor;
uniform vec3 matSpecularColor;
uniform float matShininess;

float attenuationConstant = 1.0;
float attenuationLiner = 0.045;
float attenuationQuadratic = 0.0075;

uniform vec3 pointPosition;
uniform vec2 scalePoint;
uniform float textureScale;

varying vec3 vertexVarying;
varying vec3 normalVarying;
uniform sampler2D tmu;
varying vec2 v_texcoord;
varying vec4 pointColor;

varying vec4 color;
mat3 getScaleMatrix(float scale, vec2 center) {
    return mat3(
        scale, 0.0, 0.0,
        0.0, scale, 0.0,
        center.x * (1.0 - scale), center.y * (1.0 - scale), 1.0
    );
}

void main() {
    // Calculate Lambertian reflection
    vec3 N = normalize(normalVarying);
    vec3 L = normalize(vertexVarying - LightPosition);
    float lambertian = max(0.0, dot(N, L));

    // Calculate specular reflection
    vec3 reflectLighDir = normalize(reflect(L, N));
    vec3 dirToCamera = normalize(CamWorldPosition - vertexVarying);
    vec3 ambient = matAmbientColor * lightAmbientColor;

    // Apply Lambertian reflection
    vec3 diffuse = matDiffuseColor * lightDiffuseColor * lambertian;

    // Apply specular reflection
    vec3 specular = matSpecularColor * lightSpecularColor * pow(max(0.0,
dot(dirToCamera, reflectLighDir)), matShininess);
```

```
    // Calculate distance-based attenuation
    float distance = length(vertexVarying - LightPosition);
    float attenuation = 1.0 / (attenuationConstant + attenuationLiner * distance +
attenuationQuadratic * (distance * distance));

    // Create a scaling matrix
    mat3 scaleMatrix = getScaleMatrix(textureScale, scalePoint);

    // Apply scaling to UV coordinates
    vec3 scaledUV = vec3(v_texcoord, 1.0) * scaleMatrix;

    // Sample the texture and apply lighting calculations
    gl_FragColor = texture2D(tmu, scaledUV.xy) * vec4(ambient * attenuation +
diffuse * attenuation + specular * attenuation, 1) + pointColor;
}`;
```