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 17

Виконав: Осадчий Денис

Студентка групи ТР-23мп

Київ 2022

1. 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.

1. Теоретичні відомості

Текстурування є дуже важливою частиною процесу 3D-моделювання. Усі дрібніші візуальні характеристики у 3D-моделюванні, такі як зморшки та окремі нитки килима, є продуктом текстури, нанесеної 3D-художником. Зазвичай створювані 3D-моделі мають стандартний сірий колір програми. Щоб додати кольори, малюнки та текстури, 2D-фотографії потрібно розмістити на 3D-моделях. Додавання кольорів або властивостей поверхні та матеріалу до 3D-моделі вимагає ще одного кроку вперед у процесі 3D-моделювання, тобто 3D-текстурування. Цей підхід часто призводить до повного кольору та властивостей поверхні 3D-моделі.

Стандартна процедура текстурування така:

UV Mapping and Unwrapping

Щоб почати процес 3D-текстурування, необхідно спочатку розгорнути модель, що, по суті, те саме, що розгортання 3D-сітки. Коли художники-фактуристи отримають готові моделі від відділу 3D-моделювання, вони створять UV-карту для кожного 3D-об’єкта. UV-карта — це плоске зображення поверхні 3D-моделі, яке використовується для швидкого накладання текстур. Прямо пов’язуючи 2D-зображення (текстуру) з вершинами багатокутника, UV-відображення може допомогти обернути 2D-зображення (текстуру) навколо 3D-об’єкта, а згенеровану карту можна використовувати безпосередньо в процесі текстурування та затінення.

Більшість програмних систем 3D мають кілька інструментів або підходів для розгортання 3D-моделей. Коли справа доходить до створення UV-карт, це питання особистих уподобань. Якщо ви не збираєтеся використовувати процедурні текстури, майже завжди потрібно розгортати 3D-модель у компоненті текстурування. Це текстури, створені за допомогою математичних методів (процесів), а не безпосередньо записаних даних у 2D або 3D.

1. Виконання завдання

В ході другої лабораторної роботи було створено поверню під назвою

«Surface of revolution of general Sinusoid». Отриману поверхню з освітленням

можна побачити на рисунку 3.1.

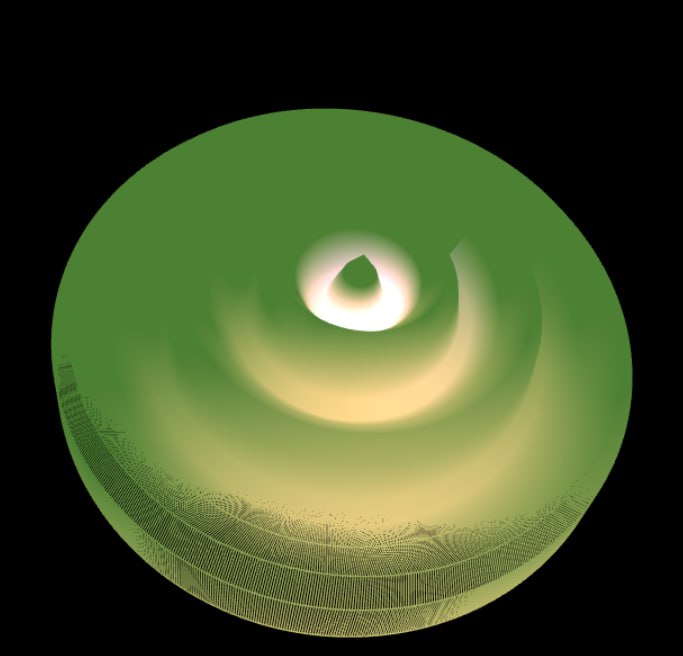


Рис. 3.1 «Surface of revolution of general Sinusoid» з освітленням

Для текстури було обрано картинку з інтернету формату «jpg». Після чого

завантажив її на github, щоб в подальшому використовувати посилання на

неї.

В графічному редакторі було налаштувано розмір картинки так, щоб

ширина і висота були рівні, а також, аби сторона мала розмір 2n 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 UV 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 «Snail Surface» з накладеною текстурою

Для відображення умовної точки відносно якої буде виконватися

трансформація текстури, в класі моделі було сворено відповідну функцію.

Замість відображення точки було прийнято рішення відобжати сферу,

адже працюємо в 3д-просторі. Для відорбраження сфери необхідно було

створити функцію, яка б створювала геометрію для неї. Модель з

умовною точкою зображено на рисунку 3.4.



Рис. 3.4 Поверхня з умовною точкою

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.

Для реалізації переміщення точки по поверхні та обертання текстури було

додано відповідні функції на відповідні вхідні дані від користувача.

1. Вказівки користувачу

Користувач може керувати переміщенням умовної точки по поверхні, обертанням текстури відносно умовної точки, а також орієнтацією поверхні в просторію При чому останні два пункти здійснюються в один і той же спосіб.

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.

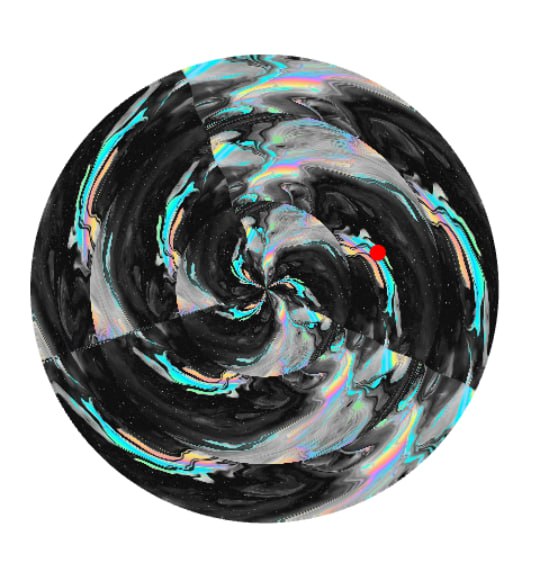
 

Рис. 4.1. Переміщення умовної точки

Орієнтація поверхні в просторі, а також трансформація текстури(рисунок 4.2) здійснюється за допомогою введення з миші: необхідно затиснути лівою клавішею миші у області відображення поверхні та потягнути в будь-яку сторону. Варто зауважити, що обертання текстури залежить від переміщення миші по горизонталі, тобто, при зміщенні затиснутої мишу тільки вгору або тільки вниз буде здійснюватись лише зміна орієнтації поверхні в просторі(рисунок 4.3)





Рис. 4.2. Трансформація текстури

На рисунку 4.3 бачим що точка та текстура залишились на одному і тому самому місці відносно поверхні. Змінилась лише орієнтація поверхні в просторі.





Рис. 4.3. Лише орієнтація поверхні в просторі

Код

let gl; // *The webgl context.*

let surface; // *surface model*

let shaderProgram; // *shader program*

let rotateBall; // *object for rotate the view by mouse.*

let magnit;

function numToRad(angle) {

return (angle \* Math**.**PI) / 180;

}

const R = 1;

const m = 1;

const n = 1;

const x = (r, B) => r \* Math**.**cos(B);

const y = (r, B) => r \* Math**.**sin(B);

const z = (r) => m \* Math**.**cos((n \* Math**.**PI \* r) / R);

function sphereSurfaceDate(r, u, v) {

let x = r \* Math**.**sin(u) \* Math**.**cos(v);

let y = r \* Math**.**sin(u) \* Math**.**sin(v);

let z = r \* Math**.**cos(u);

return { x: x, y: y, z: z };

}

let position = 0.0;

// *initialize Model*

function Model(name) {

this**.**name = name;

this**.**iVertexBuffer = gl**.**createBuffer();

this**.**iVertexTextureBuffer = gl**.**createBuffer();

this**.**count = 0;

this**.**textureCount = 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**.**TextureBufferData = function (vertices) {

gl**.**bindBuffer(gl**.**ARRAY\_BUFFER, this**.**iVertexTextureBuffer);

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

this**.**textureCount = vertices**.**length / 2;

};

this**.**Draw = function () {

gl**.**bindBuffer(gl**.**ARRAY\_BUFFER, this**.**iVertexBuffer);

gl**.**vertexAttribPointer(

shaderProgram**.**iAttribVertex,

3,

gl**.**FLOAT,

false,

0,

0

);

gl**.**enableVertexAttribArray(shaderProgram**.**iAttribVertex);

gl**.**bindBuffer(gl**.**ARRAY\_BUFFER, this**.**iVertexTextureBuffer);

gl**.**vertexAttribPointer(

shaderProgram**.**iAttribVertexTexture,

2,

gl**.**FLOAT,

false,

0,

0

);

gl**.**enableVertexAttribArray(shaderProgram**.**iAttribVertexTexture);

gl**.**drawArrays(gl**.**TRIANGLE\_STRIP, 0, this**.**count);

};

this**.**DrawSphere = function () {

gl**.**bindBuffer(gl**.**ARRAY\_BUFFER, this**.**iVertexBuffer);

gl**.**vertexAttribPointer(

shaderProgram**.**iAttribVertex,

3,

gl**.**FLOAT,

false,

0,

0

);

gl**.**enableVertexAttribArray(shaderProgram**.**iAttribVertex);

gl**.**drawArrays(gl**.**TRIANGLE\_STRIP, 0, this**.**count);

};

}

// *Constructor*

function ShaderProgram(name, program) {

this**.**name = name;

this**.**prog = program;

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

this**.**iAttribVertex = -1;

this**.**iAttribVertexTexture = -1;

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

this**.**iModelViewProjectionMatrix = -1;

this**.**iTMU = -1;

this**.**iUserPoint = -1;

this**.**iMagnit = 1;

this**.**iTranslateSphere = -1;

this**.**Use = function () {

gl**.**useProgram(this**.**prog);

};

}

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

function draw() {

gl**.**clearColor(1, 1, 1, 1);

gl**.**clear(gl**.**COLOR\_BUFFER\_BIT | gl**.**DEPTH\_BUFFER\_BIT);

/\* *Set the values of the projection transformation* \*/

let projection = m4**.**orthographic(-4, 4, -4, 4, 0, 4 \* 4);

/\* *Get the view matrix from the SimpleRotator object.*\*/

let modelView = rotateBall**.**getViewMatrix();

let rotateToPointZero = m4**.**axisRotation([0.707, 0.707, 0], 0.7);

let translateToPointZero = m4**.**translation(0, 0, -10);

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(

shaderProgram**.**iModelViewProjectionMatrix,

false,

modelViewProjection

);

gl**.**uniform1i(shaderProgram**.**iTMU, 0);

gl**.**enable(gl**.**TEXTURE\_2D);

gl**.**uniform2fv(shaderProgram**.**iUserPoint, [userPoint**.**x, userPoint**.**y]);

gl**.**uniform1f(shaderProgram**.**iMagnit, magnit);

gl**.**uniform1f(shaderProgram**.**iB, -1);

gl**.**uniform3fv(shaderProgram**.**iTranslateSphere, [-0, -0, -0]);

surface**.**Draw();

let translate = Sinus(

map(userPoint**.**x, 0, 1, 0, 5),

map(userPoint**.**y, 0, 1, 0, Math**.**PI \* 2)

);

gl**.**uniform3fv(shaderProgram**.**iTranslateSphere, [

translate**.**x,

translate**.**y,

translate**.**z,

]);

gl**.**uniform1f(shaderProgram**.**iB, 1);

sphere**.**DrawSphere();

}

function CreateSurfaceData() {

let vertexList = [];

const step = 0.05;

for (let i = 0; i < 5; i += step) {

for (let j = 0; j < Math**.**PI \* 2; j += step) {

let v1 = Sinus(i, j);

let v2 = Sinus(i + step, j);

let v3 = Sinus(i, j + step);

let v4 = Sinus(i + step, j + step);

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(v2**.**x, v2**.**y, v2**.**z);

vertexList**.**push(v4**.**x, v4**.**y, v4**.**z);

vertexList**.**push(v3**.**x, v3**.**y, v3**.**z);

}

}

return vertexList;

}

/\* *Initialize the WebGL context. Called from init()* \*/

function initGL() {

let prog = createProgram(gl, vertexShaderSource, fragmentShaderSource);

shaderProgram = new ShaderProgram("Basic", prog);

shaderProgram**.**Use();

shaderProgram**.**iAttribVertex = gl**.**getAttribLocation(prog, "vertex");

shaderProgram**.**iAttribVertexTexture = gl**.**getAttribLocation(

prog,

"vertexTexture"

);

shaderProgram**.**iModelViewProjectionMatrix = gl**.**getUniformLocation(

prog,

"ModelViewProjectionMatrix"

);

shaderProgram**.**iTMU = gl**.**getUniformLocation(prog, "TMU");

shaderProgram**.**iUserPoint = gl**.**getUniformLocation(prog, "userPoint");

shaderProgram**.**iMagnit = gl**.**getUniformLocation(prog, "magnit");

shaderProgram**.**iTranslateSphere = gl**.**getUniformLocation(

prog,

"translateSphere"

);

shaderProgram**.**iB = gl**.**getUniformLocation(prog, "b");

LoadTexture();

surface = new Model("Surface");

surface**.**BufferData(CreateSurfaceData());

surface**.**TextureBufferData(CreateSurfaceTextureData());

sphere = new Model("Sphere");

sphere**.**BufferData(CreateSphereSurface());

gl**.**enable(gl**.**DEPTH\_TEST);

}

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 = "anonymus";

image**.**src =

"https://raw.githubusercontent.com/denisosadchyi1/WebGL/CGW/background.jpg";

image**.**onload = () => {

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

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

draw();

};

}

function CreateSphereSurface(r = 0.1) {

let vertexList = [];

let lon = -Math**.**PI;

let lat = -Math**.**PI \* 0.5;

while (lon < Math**.**PI) {

while (lat < Math**.**PI \* 0.5) {

let v1 = sphereSurfaceDate(r, lon, lat);

let v2 = sphereSurfaceDate(r, lon + 0.5, lat);

let v3 = sphereSurfaceDate(r, lon, lat + 0.5);

let v4 = sphereSurfaceDate(r, lon + 0.5, lat + 0.5);

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(v2**.**x, v2**.**y, v2**.**z);

vertexList**.**push(v4**.**x, v4**.**y, v4**.**z);

vertexList**.**push(v3**.**x, v3**.**y, v3**.**z);

lat += 0.5;

}

lat = -Math**.**PI \* 0.5;

lon += 0.5;

}

return vertexList;

}

function CreateSurfaceTextureData() {

let vertexList = [];

const step = 0.05;

for (let i = 0; i < 5; i += step) {

for (let j = 0; j < Math**.**PI \* 2; j += step) {

let u = map(i, 0, 5, 0, 1);

let v = map(j, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i + step, 0, 5, 0, 1);

vertexList**.**push(u, v);

u = map(i, 0, 5, 0, 1);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i + step, 0, 5, 0, 1);

v = map(j, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i, 0, 5, 0, 1);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

}

}

return vertexList;

}

function CreateSurfaceTextureData() {

let vertexList = [];

const step = 0.05;

for (let i = 0; i < 5; i += step) {

for (let j = 0; j < Math**.**PI \* 2; j += step) {

let u = map(i, 0, 5, 0, 1);

let v = map(j, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i + step, 0, 5, 0, 1);

vertexList**.**push(u, v);

u = map(i, 0, 5, 0, 1);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i + step, 0, 5, 0, 1);

v = map(j, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

u = map(i, 0, 5, 0, 1);

v = map(j + step, 0, Math**.**PI \* 2, 0, 1);

vertexList**.**push(u, v);

}

}

return vertexList;

}

function map(val, f1, t1, f2, t2) {

let m;

m = ((val - f1) \* (t2 - f2)) / (t1 - f1) + f2;

return Math**.**min(Math**.**max(m, f2), t2);

}

function Sinus(r, b) {

const a = 1;

const rr = 0.55;

const n = 1;

let x = r \* Math**.**cos(b);

let y = r \* Math**.**sin(b);

let z = a \* Math**.**sin((n \* Math**.**PI \* r) / rr);

return { x: 0.55 \* x, y: 0.55 \* y, z: 0.55 \* z };

}

/\* *Creates a program for use in the WebGL context gl* \*/

function createProgram(gl, vShader, fShader) {

let vsh = gl**.**createShader(gl**.**VERTEX\_SHADER);

gl**.**shaderSource(vsh, vShader);

gl**.**compileShader(vsh);

if (!gl**.**getShaderParameter(vsh, gl**.**COMPILE\_STATUS)) {

throw new Error("Error in vertex shader: " + gl**.**getShaderInfoLog(vsh));

}

let fsh = gl**.**createShader(gl**.**FRAGMENT\_SHADER);

gl**.**shaderSource(fsh, fShader);

gl**.**compileShader(fsh);

if (!gl**.**getShaderParameter(fsh, gl**.**COMPILE\_STATUS)) {

throw new Error("Error in fragment shader: " + gl**.**getShaderInfoLog(fsh));

}

let prog = gl**.**createProgram();

gl**.**attachShader(prog, vsh);

gl**.**attachShader(prog, fsh);

gl**.**linkProgram(prog);

if (!gl**.**getProgramParameter(prog, gl**.**LINK\_STATUS)) {

throw new Error("Link error in program: " + gl**.**getProgramInfoLog(prog));

}

return prog;

}

/\*\*

*\* initialization function that will be called when the page has loaded*

\*/

function init() {

userPoint = { x: 0.5, y: 0.5 };

magnit = 1.0;

let canvas;

try {

canvas = document**.**getElementById("webglcanvas");

gl = canvas**.**getContext("webgl");

if (!gl) {

// *Check condition support WebGL*

throw "Your browser does not support WebGL, pls check version of your browser and support WebGL";

}

} catch (e) {

console**.**log(e, "error loading webgl");

document**.**getElementById("canvas-holder")**.**innerHTML =

"<p>Could'nt get a WebGL context :(</p>";

return;

}

try {

initGL(); // *Check condition initialize the WebGL context*

} catch (e) {

console**.**log(e, "error loading webgl");

document**.**getElementById("canvas-holder")**.**innerHTML =

"<p>Could'nt initialize the WebGL context :( </p>";

return;

}

rotateBall = new TrackballRotator(canvas, draw, 0);

draw();

}

window**.**addEventListener("keydown", function (event) {

switch (event**.**keyCode) {

case 37:

handleChangeLeft();

case 39:

handleChangeRight();

break;

default:

return;

}

});

window**.**onkeydown = (e) => {

switch (e**.**keyCode) {

case 87:

userPoint**.**y -= 0.01;

break;

case 83:

userPoint**.**y += 0.01;

break;

case 65:

userPoint**.**x += 0.01;

break;

case 68:

userPoint**.**x -= 0.01;

break;

}

userPoint**.**x = Math**.**max(0.01, Math**.**min(userPoint**.**x, 0.999));

userPoint**.**y = Math**.**max(0.01, Math**.**min(userPoint**.**y, 0.999));

draw();

};

onmousemove = (e) => {

magnit = map(e**.**clientX, 0, window**.**outerWidth, 0, Math**.**PI);

draw();

};

const lightCoordinates = () => {

let coord = Math**.**sin(position) \* 1.1;

return [coord, -2, coord \* coord];

};

const handleChangeLeft = () => {

position -= 0.07;

reDraw();

};

const handleChangeRight = () => {

position += 0.07;

reDraw();

};

const reDraw = () => {

surface**.**BufferData(CreateSurfaceData());

draw();

};