МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

КАФЕДРА ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

**ВІЗУАЛІЗАЦІЯ ГРАФІЧНОЇ ТА ГЕОМЕТРИЧНОЇ ІНФОРМАЦІЇ**

РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА

“Операції з текстурними координатами”

Виконав:

студент групи ТР-32мп

Мар’яш Дмитро Ігорович

Київ – 2024

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. **Теоретичні відомості**

WebGL — це інтерфейс прикладного програмування JavaScript (API), який дозволяє використовувати графічний процесор (GPU) обчислювального пристрою у вікні веб-браузера для відтворення 3D-графіки комп’ютера в реальному часі. Це єдине міжплатформне середовище розробки для створення 3D-комп’ютерної графіки, яке сьогодні є в галузі програмного забезпечення. Вивчення WebGL — це найкращий спосіб зрозуміти основи комп’ютерної графіки, маючи при цьому «набір інструментів», який можна використовувати для створення власної комп’ютерної графіки.

Типова веб-сторінка, яка містить тривимірну комп’ютерну графіку, згенеровану програмою WebGL, складається з таких основних частин:

* HTML (мова розмітки гіпертексту) опис веб-сторінки.
* CSS (каскадна таблиця стилів), яка описує форматування кожного елемента опису HTML.
* Елемент HTML-полотна на веб-сторінці, який надає прямокутну область, у якій можна відтворювати тривимірну комп’ютерну графіку.
* Графічні дані, які визначають тривимірні об’єкти, які потрібно відобразити.
* Програми JavaScript, які завантажують ваші графічні дані, налаштовують ваші графічні дані, відображають ваші графічні дані та код, який реагує на події користувача.
* Програми OpenGL Shader, які виконують критичні частини графічного рендерингу.

Усі ці базові частини спочатку знаходяться на веб-сервері. Коли користувач (клієнт) запитує веб-сторінку, яка містить 3D-графіку, дані, описані вище, повинні бути передані на комп’ютер клієнта, збережені в правильних місцях, а потім оброблені відповідними процесорами. Сучасні обчислювальні пристрої мають кілька процесорів різних типів.

У візуалізації WebGL вершинні шейдери працюють як математичні сутності, яким доручено просторово маніпулювати геометричними примітивами. Їх основною функцією є перетворення однорідних координат, що представляють вершини в тривимірному просторі, у двовимірні координати простору кліпу. Це перетворення включає лінійні та афінні операції, що охоплюють трансляції, повороти, масштабування та перспективні проекції.

Вершиний шейдер обробляє вхідні атрибути, пов’язані з кожною вершиною, включаючи позиційну інформацію, дані кольору та координати текстури. Результатом є уточнений набір позицій вершин в однорідних координатах, готовий для подальшої обробки на наступних етапах конвеєра візуалізації. Крім того, вершинні шейдери полегшують інтерполяцію атрибутів між примітивами, такими як трикутники, забезпечуючи візуальну безперервність під час растеризації.

У безперервному процесі візуалізації WebGL фрагментні шейдери з’являються як обчислювальні сутності, відповідальні за детальне визначення значень пікселів у растеризованих примітивах. Працюючи над деталізацією окремих пікселів, ці шейдери враховують інтерпольовані атрибути вершинного шейдера, такі як колір, координати текстури та нормалі, щоб обчислити остаточний колір кожного пікселя.

Фрагментні шейдери допомагають виконувати такі операції, як відображення текстур, обчислення освітлення та моделі затінення. Завдяки цим обчисленням вони надають відтвореній сцені певного рівня реалізму та естетичної витонченості. Крім того, фрагментні шейдери відіграють ключову роль у тестуванні глибини, перевіряючи видимість фрагментів у випадках просторового перекриття.

Ефективний зв’язок між вершинними та фрагментними шейдерами має важливе значення для узгодженого відтворення. Цей зв’язок між шейдерами забезпечується змінними, які служать каналами для передачі інформації від вершинних до фрагментних шейдерів. Ці змінні проходять інтерполяцію на примітивних поверхнях під час процесу растеризації, забезпечуючи плавний перехід атрибутів.

Уніфіковані змінні, з іншого боку, служать константами, спільними для всіх вершин або фрагментів, і залишаються незмінними під час послідовності візуалізації. Ці константи, встановлені програмою, забезпечують механізм для глобально синхронізованого обміну даними між двома шейдерами.

За своєю суттю відображення текстур включає асоціацію кожної вершини на 3D-моделі з відповідною точкою на 2D-текстурі. Сенс цього відображення полягає у визначенні координат текстури, двовимірних значень, позначених як «u» та «v». «u» відповідає горизонтальній осі, вирівнюючись із шириною текстури, тоді як «v» відповідає вертикальній осі, віддзеркалюючи висоту текстури. Ці координати нормалізуються в діапазоні від 0,0 до 1,0, де (0,0, 0,0) позначає нижній лівий кут текстури, а (1,0, 1,0) позначає верхній правий кут.

У парадигмі WebGL атрибути вершин інкапсулюють дані, пов’язані з кожною вершиною, і в цій структурі координати текстур є важливою підмножиною. Кожна вершина в 3D-моделі наділена набором координат текстури, що точно визначає конкретне місце на текстурі.

Координати текстури піддаються інтерполяції по всій поверхні 3D-об’єкта, що є ключовим механізмом, що забезпечує безперебійне відображення текстури навіть на нерівних або вигнутих поверхнях. Ця інтерполяція сприяє візуальній цілісності текстури, коли вона поширюється на об’єкт.

Роль координат текстури поширюється на визначення поведінки обтікання текстурою, коли поверхня виходить за межі текстури. Режими обтікання, такі як повторення або затискання, визначають, як керувати координатами текстури за межами нормалізованого діапазону [0,0, 1,0].

Координати текстури також відіграють важливу роль у виборі відповідних рівнів mipmap на основі відстані об’єкта від камери. Рівні Mipmap, попередньо обчислені та зменшені версії текстури підвищують ефективність візуалізації та пом’якшують проблеми з псевдонімами.

Крім своєї основної ролі, координати текстури можуть зазнавати таких трансформацій, як масштабування, переміщення або обертання. Ці перетворення, часто представлені через матриці, створюють динамічні та різноманітні візуальні ефекти шляхом маніпулювання відображенням між 3D-геометрією та 2D-текстурами.

1. **Деталі розробки**

У якості поверхні, для виконання практичного завдання №2, Мені було запропоновано створити Ding-Dong Surface. Для реалізації мети практичного завдання, була створена програма, яка генерує поверхню у вигляді суцільних трикутників.

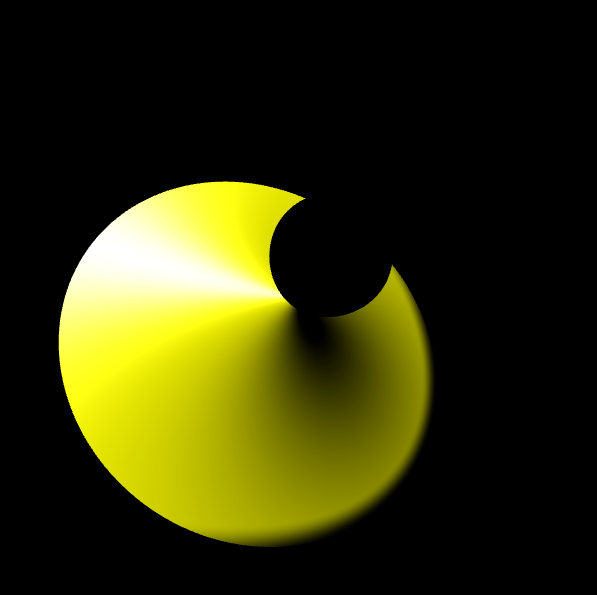


Рисунок 1 – Вигляд Ding-Dong Surface, практичне завдання №2

Для розрахунково-графічної роботи, було обрано зображення, що можна побачити на Рисунку 2. Розмір обраного зображення - 512х512 пікселів, формат - .jpg..

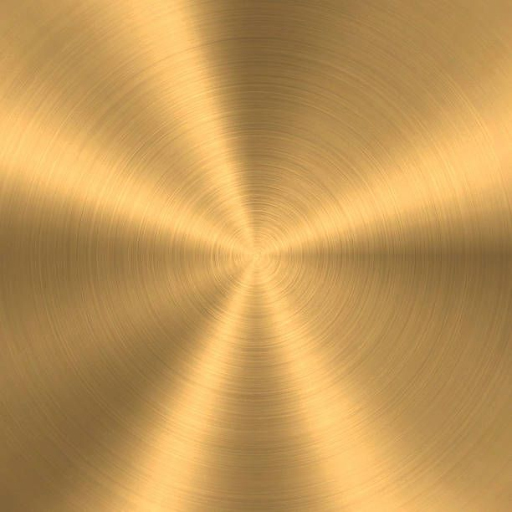


Рисунок 2 – Зображення текстури

Текстура була застосована до поверхні з використанням буфера текстурних координат, який відповідає кожному елементу масиву вершин.

Згідно варіанту текстура має масштабуватися, і для цього було введено відповідний параметр масштабування. У програмі шейдера для визначення кольору пікселя фігури з текстурою використовується функція texture2D().

Ця функція першим аргументом приймає обʼєкт класу sampler2D, який зберігає дані про зображення, а другим текстурну координату.

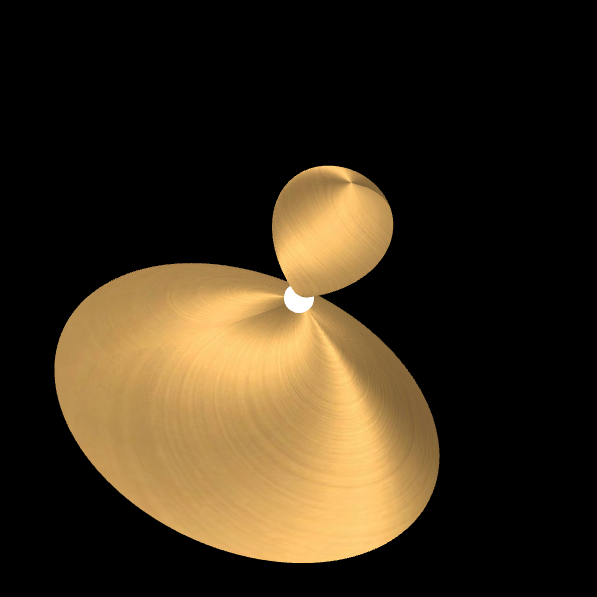
****

Рисунок 3 – Вигляд Ding-Dong Surface з текстурою

Було створено об'єкт класу Model для відображення точки, відносно якої відбувається трансформація текстури. Цей об'єкт графічно представлений у вигляді сфери, яка розташована на поверхні.

A gold circle with a white circle in the middle

Description automatically generated

Рисунок 4 – Точка на текстурованій Ding-Dong Surface

1. **Інструкція користувача**

Поверхню можна обертати, тримаючи ліву клавішу миші та пересуваючи в потрібному напрямку.

A yellow circle with a white light in the middle

Description automatically generated A gold circle with a black background

Description automatically generated

Рисунок 5 – Положення Ding-Dong Surface при обертанні

Переміщення точки, навколо якої відбувається обертання, можливе за допомогою клавіш WASD. Кожне натискання зсуває точку на поверхні на визначений крок, і переміщення обмежується певним лімітом.

A yellow circle with a white light in the middle

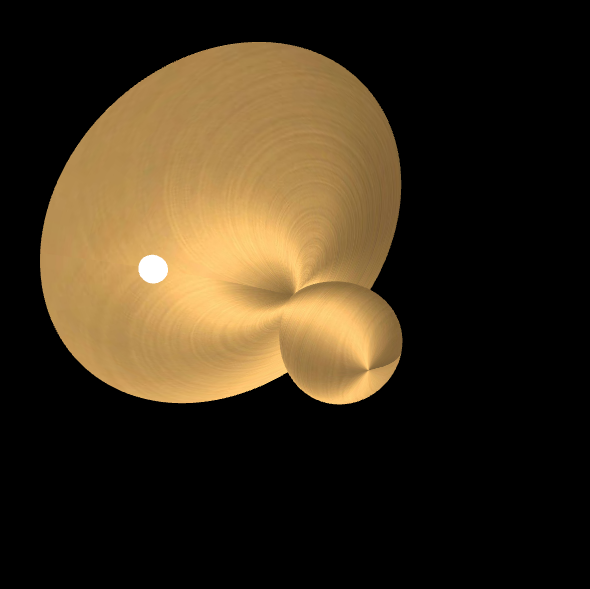
Description automatically generated 

Рисунок 5 – Переміщення точки на Ding-Dong Surface

Слайдер з підписом "Texture image scaling" дозволяє змінювати фактор масштабування відносно умовної точки на поверхні. Важливо відзначити, що при переміщенні точки також зміщується текстура, оскільки масштабування відбувається відносно іншої точки на поверхні, яка, в свою чергу, відповідає іншій текстурній координаті.

A yellow and white object with a black background

Description automatically generated A gold object with a round object on top

Description automatically generated

Рисунок 6 - Демонстрація переміщення точки відносно нерухомої фігури

При перезавантаженні сторінки обертання фігури та значення фактору масштабування будуть встановлені за замовчуванням. Положення точки на поверхні також повернеться до значення за замовчуванням.

1. **Лістинг**

**//** main.js

function CreateDingDongSurfaceData() {

let divisionsU = 150;

let divisionsV = 150;

let vertexList = [];

let normalList = [];

let texCoordList = [];

let u1 = 1 / (divisionsU - 1) \* 2 \* Math.PI

let v1 = 1 / (divisionsV - 1) \* 2

for (let u = 0; u <= divisionsU; u++) {

for (let v = 0; v <= divisionsV; v++) {

let u0 = u \* u1;

let v0 = v \* v1 - 1;

vertexList.push(...CreateDingDongVertex(u0, v0));

vertexList.push(...CreateDingDongVertex(u0 + u1, v0));

vertexList.push(...CreateDingDongVertex(u0, v0 + v1));

vertexList.push(...CreateDingDongVertex(u0, v0 + v1));

vertexList.push(...CreateDingDongVertex(u0 + u1, v0));

vertexList.push(...CreateDingDongVertex(u0 + u1, v0 + v1));

normalList.push(...CreateAverageNormal(u0, v0, u1, v1));

normalList.push(...CreateAverageNormal(u0 + u1, v0, u1, v1));

normalList.push(...CreateAverageNormal(u0, v0 + v1, u1, v1));

normalList.push(...CreateAverageNormal(u0, v0 + v1, u1, v1));

normalList.push(...CreateAverageNormal(u0 + u1, v0, u1, v1));

normalList.push(...CreateAverageNormal(u0 + u1, v0 + v1, u1, v1));

texCoordList.push(u / divisionsU, v / divisionsV);

texCoordList.push((u + 1) / divisionsU, v / divisionsV);

texCoordList.push(u / divisionsU, (v + 1) / divisionsV);

texCoordList.push(u / divisionsU, (v + 1) / divisionsV);

texCoordList.push((u + 1) / divisionsU, v / divisionsV);

texCoordList.push((u + 1) / divisionsU, (v + 1) / divisionsV);

}

}

return { v: vertexList, n: normalList, t: texCoordList };

}

function CreateAverageNormal(u0, v0, u1, v1) {

let vertex = CreateDingDongVertex(u0, v0)

let vertexA = CreateDingDongVertex(u0 + u1, v0)

let vertexB = CreateDingDongVertex(u0, v0 + v1)

let vertexC = CreateDingDongVertex(u0 - u1, v0 + v1)

let vertexD = CreateDingDongVertex(u0 - u1, v0)

let vertexE = CreateDingDongVertex(u0 - u1, v0 - v1)

let vertexF = CreateDingDongVertex(u0, v0 - v1)

vertex = m4.normalize(vertex)

vertexA = m4.normalize(vertexA)

vertexB = m4.normalize(vertexB)

vertexC = m4.normalize(vertexC)

vertexD = m4.normalize(vertexD)

vertexE = m4.normalize(vertexE)

vertexF = m4.normalize(vertexF)

let a = m4.subtractVectors(vertexA, vertex)

let b = m4.subtractVectors(vertexB, vertex)

let c = m4.subtractVectors(vertexC, vertex)

let d = m4.subtractVectors(vertexD, vertex)

let e = m4.subtractVectors(vertexE, vertex)

let f = m4.subtractVectors(vertexF, vertex)

let n1 = m4.normalize(m4.cross(a, b))

let n2 = m4.normalize(m4.cross(b, c))

let n3 = m4.normalize(m4.cross(c, d))

let n4 = m4.normalize(m4.cross(d, e))

let n5 = m4.normalize(m4.cross(e, f))

let n6 = m4.normalize(m4.cross(f, a))

let avgNormal = [

(n1[0] + n2[0] + n3[0] + n4[0] + n5[0] + n6[0]) / 6.0,

(n1[1] + n2[1] + n3[1] + n4[1] + n5[1] + n6[1]) / 6.0,

(n1[2] + n2[2] + n3[2] + n4[2] + n5[2] + n6[2]) / 6.0

]

avgNormal = m4.normalize(avgNormal);

return avgNormal;

}

// shader.gpu

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex, normal;

attribute vec2 texCoord;

uniform mat4 ModelViewProjectionMatrix;

varying vec3 vertexInterp, normalInterp;

varying vec2 tex;

uniform vec2 u\_location;

uniform float u\_scale;

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

mat4 dst;

dst[0][0] = 1.0;

dst[0][ 1] = 0.0;

dst[0][ 2] = 0.0;

dst[0][ 3] = 0.0;

dst[1][ 0] = 0.0;

dst[1][ 1] = 1.0;

dst[1][ 2] = 0.0;

dst[1][ 3] = 0.0;

dst[2][ 0] = 0.0;

dst[2][ 1] = 0.0;

dst[2][ 2] = 1.0;

dst[2][ 3] = 0.0;

dst[3][ 0] = tx;

dst[3][ 1] = ty;

dst[3][ 2] = tz;

dst[3][ 3] = 1.0;

return dst;

}

mat4 scaling(float s){

mat4 dst;

dst[0][0] = s;

dst[0][ 1] = 0.0;

dst[0][ 2] = 0.0;

dst[0][ 3] = 0.0;

dst[1][ 0] = 0.0;

dst[1][ 1] = s;

dst[1][ 2] = 0.0;

dst[1][ 3] = 0.0;

dst[2][ 0] = 0.0;

dst[2][ 1] = 0.0;

dst[2][ 2] = s;

dst[2][ 3] = 0.0;

dst[3][ 0] = 0.0;

dst[3][ 1] = 0.0;

dst[3][ 2] = 0.0;

dst[3][ 3] = 1.0;

return dst;

}

void main() {

vertexInterp = vertex;

normalInterp = normal;

vec4 tex1 = translation(-u\_location.x,-u\_location.y,0.0)\*vec4(texCoord,0.,1.);

vec4 tex2 = scaling(u\_scale)\*tex1;

vec4 tex3 = translation(u\_location.x,u\_location.y,0.0)\*tex2;

tex = tex3.st;

gl\_Position = ModelViewProjectionMatrix \* vec4(vertex,1.0);

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

precision highp float;

#else

precision mediump float;

#endif

uniform vec4 color;

varying vec3 vertexInterp, normalInterp;

uniform vec3 u\_position;

uniform vec3 u\_direction;

uniform float u\_limit;

uniform float u\_border;

varying vec2 tex;

uniform sampler2D tmu;

void main() {

vec3 toLight = normalize(u\_position-vertexInterp);

vec3 toView = normalize(-vertexInterp);

vec3 halfVector = normalize(toLight + toView);

vec3 N = normalize(normalInterp);

float dotFromDirection = dot(toLight,

-u\_direction);

float inLight = smoothstep(u\_limit,u\_limit+u\_border, dotFromDirection);

float light = inLight \* dot(N, toLight);

float specular = inLight \* pow(dot(N, halfVector), 150.0);

vec3 shadedColor = color.rgb\*light+specular;

vec4 texColor = texture2D(tmu,tex);

gl\_FragColor = color;

gl\_FragColor = texColor;

//gl\_FragColor = vec4(N,1.0);

//gl\_FragColor = vec4(shadedColor,1.0);

if(u\_border>100.0){

gl\_FragColor = vec4(1.0);

}

}`;