МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

КАФЕДРА ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

**Розрахунково-графічна робота**

з дисципліни «Візуалізація графічної та геометричної інформації»

**Виконав:**

Студент 5-го курсу

ІАТЕ

групи ТР-23мп

Рудик Володимир Іванович

**Перевірив:**

Демчишин Анатолій Анатолійович

Київ-2022

**Завдання**

**Варіант №18**

1. Нанести текстуру на поверхню з практичного завдання №2.
2. Реалізувати обертання текстури навколо вказаної користувачем точки.
3. Реалізувати можливість переміщати цю точку по поверхні за допомогою клавіш W, A, S, D (W, S відповідають за пересування вздовж параметра v, а A, D вздовж параметра u).
4. Створити gif зображення роботи розробленої програми.
5. Створити звіт з виконання розрахунково графічної роботи.
6. Завантажити код, gif та звіт в окрему гілку в репозиторії GitHub.

**Мета роботи**: засвоїти навички роботи з текстурами та бібліотеки WebGL.

**Основне завдання**: відобразити текстуру на поверхні та реалізувати обертання текстури навколо вказаної точки на поверхні.

**Теоретичні відомості**

**WebGL** (скорочення від Web Graphics Library) — це API JavaScript для відтворення інтерактивної 2D і 3D графіки в будь-якому сумісному веб-браузері без використання плагінів. WebGL повністю інтегровано з іншими веб-стандартами, що дозволяє використовувати фізику, обробку зображень і ефекти з прискореним графічним процесором як частину полотна веб-сторінки. Елементи WebGL можна змішувати з іншими елементами HTML і поєднувати з іншими частинами чи фоном сторінки.

Програми WebGL складаються з керуючого коду, написаного на JavaScript, і коду шейдера, написаного мовою OpenGL ES Shading Language (GLSL ES), мовою, схожою на C або C++, і виконується на графічному процесорі (GPU) комп’ютера. WebGL розроблено та підтримується неприбутковою компанією Khronos Group.

WebGL 1.0 базується на OpenGL ES 2.0 і надає API для 3D-графіки. Він використовує елемент canvas HTML5 і доступ до нього здійснюється за допомогою інтерфейсів Document Object Model (DOM).

WebGL 2.0 базується на OpenGL ES 3.0 і забезпечує гарантовану доступність багатьох додаткових розширень WebGL 1.0 і надає нові API. Автоматичне керування пам'яттю неявно забезпечується JavaScript.

Як і OpenGL ES 2.0, WebGL не має фіксованих функцій API, представлених у OpenGL 1.0 і застарілих у OpenGL 3.0. Цю функцію, якщо вона потрібна, має реалізувати кінцевий розробник, надавши код шейдера та налаштувавши прив’язки даних у JavaScript.

Шейдери в WebGL виражаються безпосередньо в GLSL і передаються в API WebGL як текстові рядки. Реалізація WebGL компілює ці інструкції шейдерів у код GPU. Цей код виконується для кожної вершини, надісланої через API, і для кожного пікселя, растеризованого на екрані.

**Текстура** (англ. Texture mapping) — це спосіб надання поверхні 3D деталей — полігону: кольору, фактури, блиску, матовості та інших фізичних властивостей (для імітації найчастіше якогось природного матеріалу, наприклад: паперу, дерева, каменю, металу тощо). Першим цю техніку запровадив Едвін Кетмул у 1974 році.

Відображення текстур спочатку називалося дифузним відображенням. Це був методо, який просто зображав пікселі текстури на 3D-поверхні («обгортаючи» об’єкт зображенням). В останні десятиліття поява багатопрохідного рендерингу, мультитекстурування, mipmaps і більш складних відображень, таких як height mapping, bump mapping, normal mapping, displacement mapping, reflection mapping, specular mapping, occlusion mapping та багато інших варіацій зробили можливим імітацію майже фото-реалістичних об’єктів в режимі реального часу.

Досягти цього вдалось шляхом значного зменшення кількості багатокутників, а також розрахунку освітлення, необхідного для створення реалістичної та функціональної 3D-сцени.

**Карта текстури** — це зображення, нанесене на поверхню фігури або багатокутника. Це може бути растрове зображення або процедурна текстура.

Вони можуть бути одно, двох, або тривимірними. Для використання з сучасним апаратним забезпеченням дані карти текстури можуть зберігатися в розрізненому або мозаїчному порядку для покращення когерентності кешу. API візуалізації зазвичай керують ресурсами карти текстури (які можуть бути розташовані в пам’яті пристрою) як буферами або поверхнями.

Зазвичай вони містять дані кольору **RGB** (збережені як прямі кольори, стислі формати або індексовані кольори), а іноді й додатковий канал для альфа-змішування (**RGBA**). Можна використовувати альфа-канал (який може бути зручним для зберігання у форматах, аналізованих апаратним забезпеченням) для інших цілей, наприклад для відображення.

Кілька текстурних карт (або каналів) можна комбінувати для контролю дзеркальності, нормалей, зміщення або підповерхневого розсіювання, наприклад для візуалізації шкіри.

**Виконання завдання**

Перш за все було створено нову гілку з назвою CGW у репозиторії GitHub. Після виконання завдання програмний код, gif демонстрація та звіт було завантажено туди.

Наступним кроком виконання був вибір текстури. Для цього потрібно розуміти, що потрібна саме POT (power of two) текстура, тобто її розмір повинен бути у степені двійки. Було обрано наступну текстуру з розмірами 1024 на 1024 (Рисунок 1).



Рисунок 1 – Обрана текстура (лід)

Іншим важливим етапом для додавання текстури на сторінку є вказання атрибуту crossorigin рівним anonymous (Рисунок 2) для забезпечення підтримки CORS і можливості завантаження її зі стороннього ресурсу.

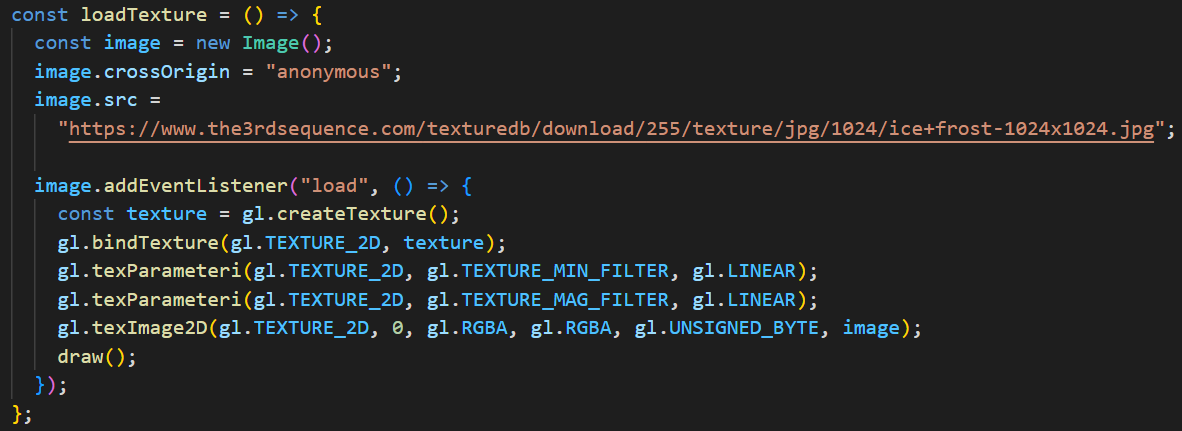


Рисунок 2 – Створення об’єкту текстури та її завантаження.

Далі потрібно передати текстуру до відеокарти. Для цього необхідно: передати картинку до буферу (Рисунок 3), використати її у шейдері як uniform sampler2D (Рисунок 4), а потім передати координати точки (u, v) до вершинного шейдеру у якості атрибуту.

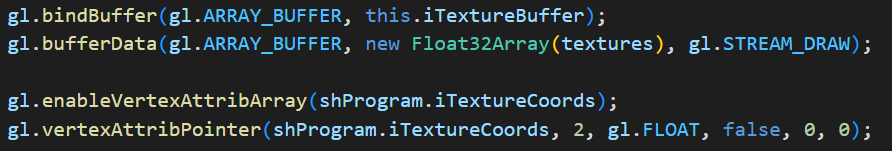


Рисунок 3 – Створення буферу та його прив’язка до атрибуту.

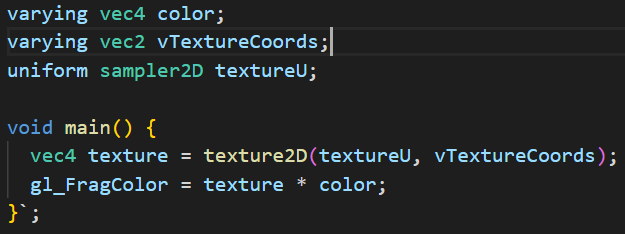


Рисунок 4 – Використання текстури у шейдері.

Також, було створено дві функції у шейдері. Перша з них відповідає за обертання фігури навколо точки (Рисунок 5).

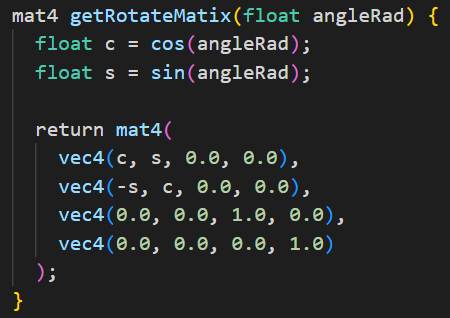


Рисунок 5 – Функція для обертання.

Наступна функція відповідає за переміщення точки, заданої користувачем (Рисунок 6).

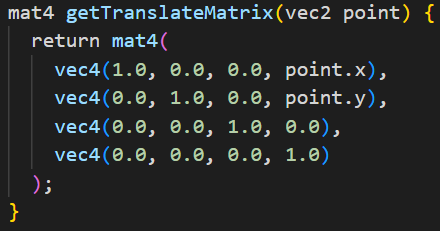


Рисунок 6 – Функція для переміщення точки.

**Інструкція користувача**

Розроблений додаток має наступний вигляд (Рисунок 7). Зверху знаходиться опис, потім повзунок для зміни кута обертання. Нижче знаходиться canvas з поверхнею, поверх якої нанесена текстура.

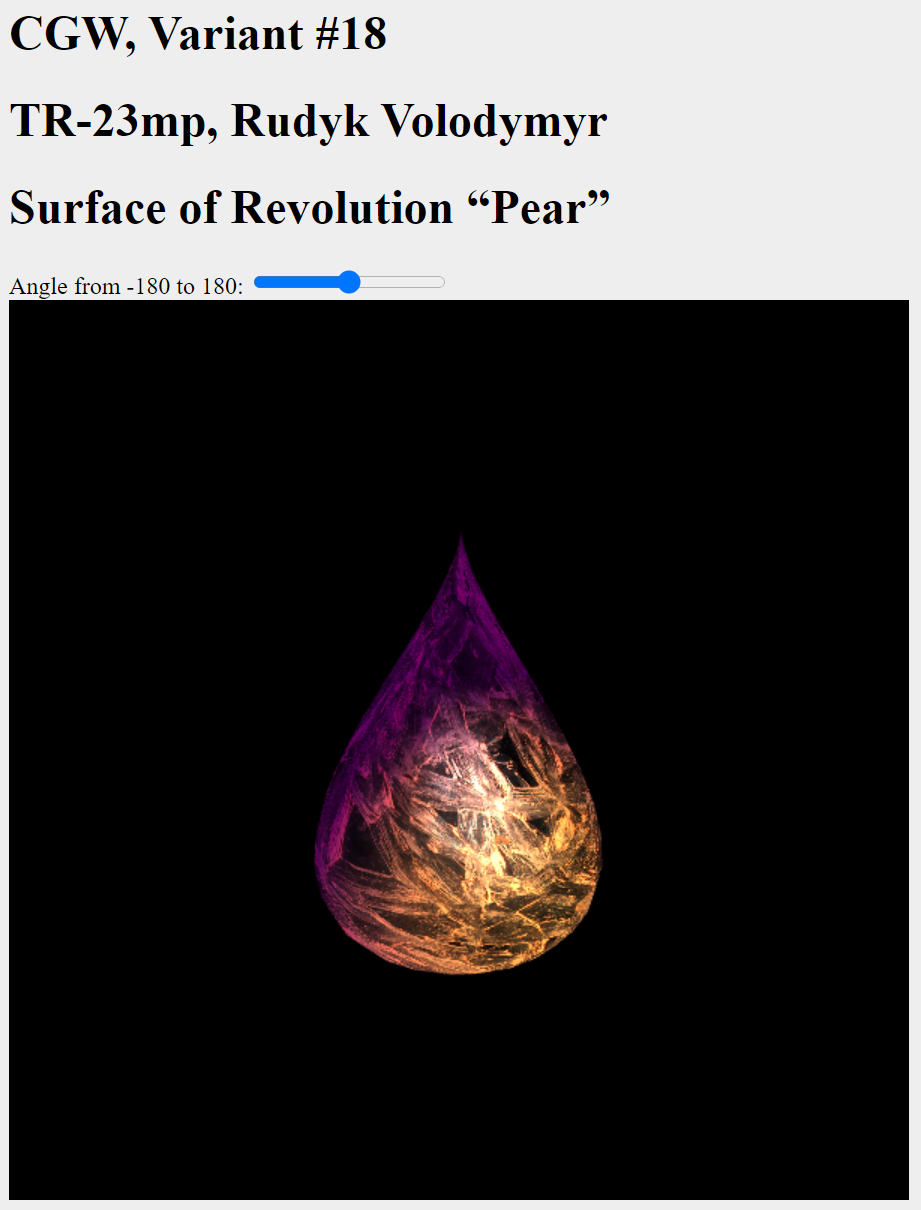


Рисунок 7 – Інтерфейс додатку.

Після зміни значення кута, фігура матиме наступний вигляд (Рисунок 8).

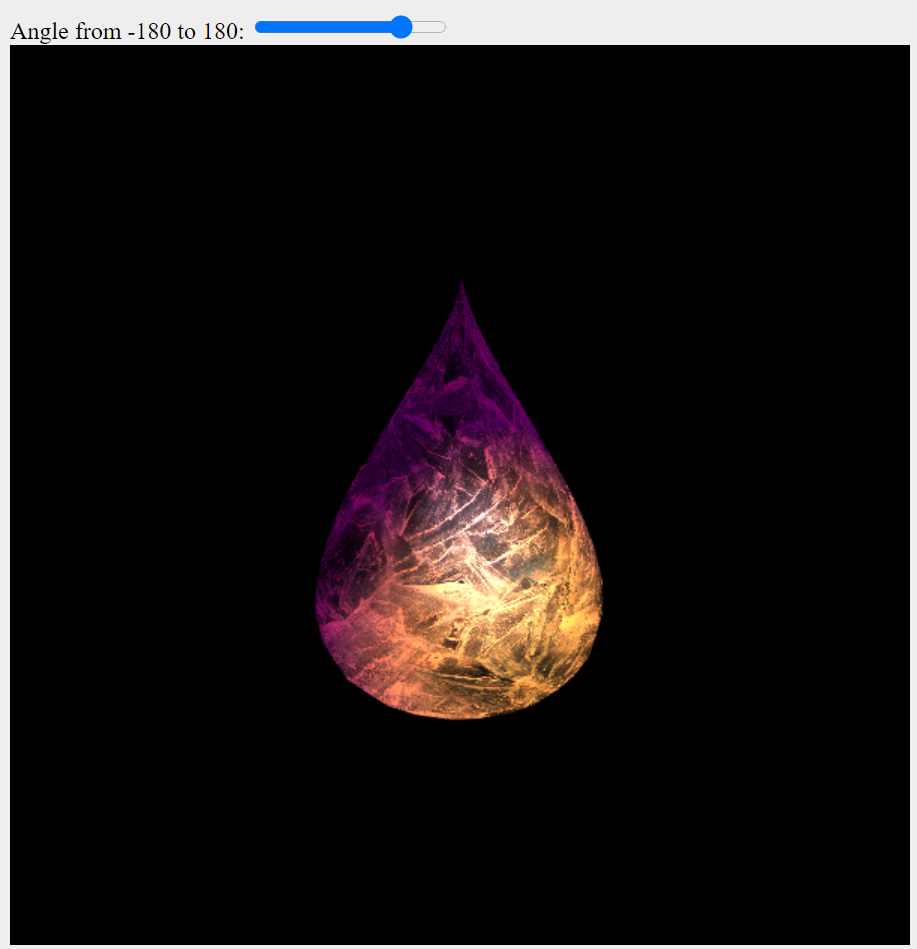


Рисунок 8 – Поверхня після зміни кута.

Для зміни положення точки, заданої користувачем використовуються клавіші WASD, як і було зазначено у завданні (Рисунок 9).

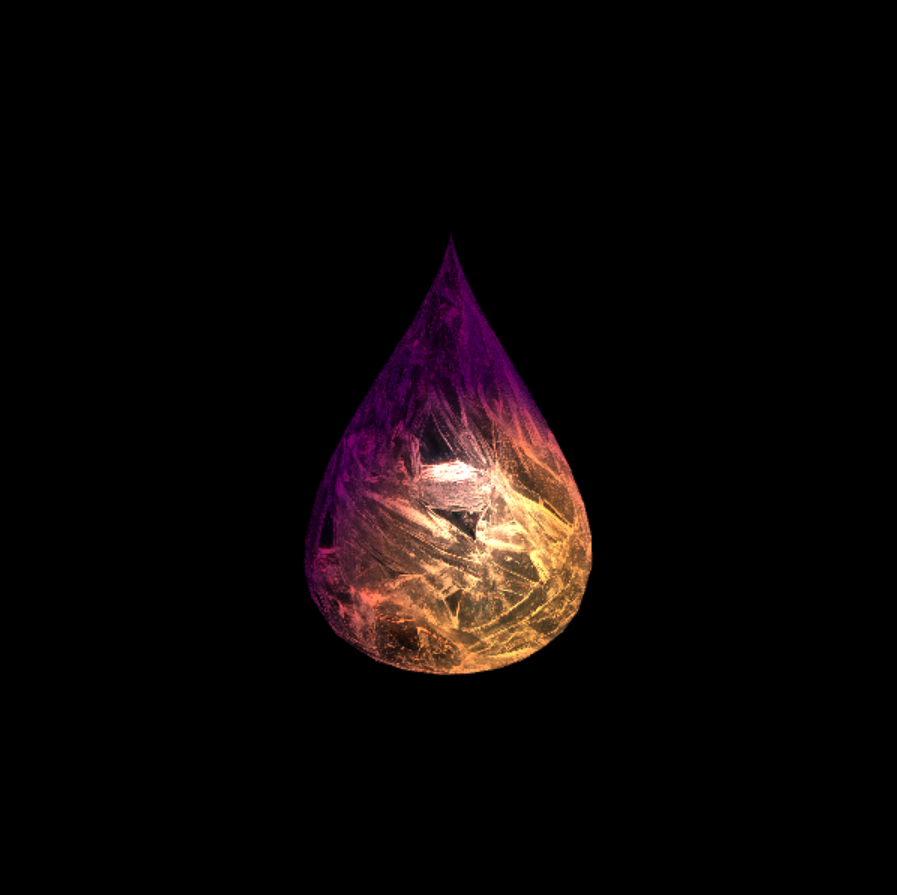


Рисунок 9 – Вигляд фігури після зміни координат точки.

**Лістинг коду**

**vertexShader.glsl**

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec3 normal;

attribute vec2 textureCoords;

uniform mat4 normalMatrix;

uniform mat4 ModelViewProjectionMatrix;

uniform float shininess;

uniform vec3 ambientColor;

uniform vec3 diffuseColor;

uniform vec3 specularColor;

uniform vec3 lightPosition;

uniform float textureAngle;

uniform vec2 texturePoint;

varying vec4 color;

varying vec2 vTextureCoords;

mat4 getRotateMatix(float angleRad) {

  float c = cos(angleRad);

  float s = sin(angleRad);

  return mat4(

    vec4(c, s, 0.0, 0.0),

    vec4(-s, c, 0.0, 0.0),

    vec4(0.0, 0.0, 1.0, 0.0),

    vec4(0.0, 0.0, 0.0, 1.0)

  );

}

mat4 getTranslateMatrix(vec2 point) {

  return mat4(

    vec4(1.0, 0.0, 0.0, point.x),

    vec4(0.0, 1.0, 0.0, point.y),

    vec4(0.0, 0.0, 1.0, 0.0),

    vec4(0.0, 0.0, 0.0, 1.0)

  );

}

void main() {

    vec4 vertexPosition4 = ModelViewProjectionMatrix \* vec4(vertex, 1.0);

    vec3 vertexPosition = vec3(vertexPosition4) / vertexPosition4.w;

    vec3 normalInterpolation = vec3(normalMatrix \* vec4(normal, 0.0));

    gl\_Position = vertexPosition4;

    vec3 normal = normalize(normalInterpolation);

    vec3 lightDirection = normalize(lightPosition - vertexPosition);

    float nDotLight = max(dot(normal, lightDirection), 0.0);

    float specularLight = 0.0;

    if (nDotLight > 0.0) {

        vec3 viewDirection = normalize(-vertexPosition);

        vec3 halfDirection = normalize(lightDirection + viewDirection);

        float specularAngle = max(dot(halfDirection, normal), 0.0);

        specularLight = pow(specularAngle, shininess);

    }

    vec3 diffuse = nDotLight \* diffuseColor;

    vec3 ambient = ambientColor;

    vec3 specular = specularLight \* specularColor;

    mat4 rotatedMatrix = getRotateMatix(textureAngle);

    mat4 translatedMatrix = getTranslateMatrix(-texturePoint);

    mat4 translatedBackMatrix = getTranslateMatrix(texturePoint);

    vec4 vTranslatedMatrix = translatedMatrix \* vec4(textureCoords, 0, 0);

    vec4 vRotatedMatrix = vTranslatedMatrix \* rotatedMatrix;

    vec4 vTranslatedBackMatrix = vRotatedMatrix \* translatedBackMatrix;

    vTextureCoords = vec2(vTranslatedBackMatrix);

    color = vec4(diffuse + ambient + specular, 1.0);

}`;

**fragmentShader.glsl**

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

    precision highp float;

#else

    precision mediump float;

#endif

varying vec4 color;

varying vec2 vTextureCoords;

uniform sampler2D textureU;

void main() {

  vec4 texture = texture2D(textureU, vTextureCoords);

  gl\_FragColor = texture \* color;

}`;