Міністерство освіти й науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Кафедра цифрових технологій в енергетиці

Візуалізація графічної та геометричної інформації

Розрахунково графічна робота

Варіант - 1

**Виконав:**

Студент 1-го курсу

групи ТР-21мп НН ІАТЕ

Барчук Роман

**Перевірив:**

Демчишин А.А.

Київ-2022

**Завдання**

1. Нанести текстуру на поверхню з практичного завдання №2.
2. Реалізація масштабування текстури (координати текстури) масштабування/обертання навколо визначеної користувачем точки – непарні варіанти реалізують масштабування, парні варіанти реалізують обертання
3. Повинна бути можливість переміщати точку 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.

**Теоретичні відомості**

WebGL — це стандарт на базі [OpenGL ES](https://uk.wikipedia.org/wiki/OpenGL_ES) 2.0, що дозволяє розробникам вбудовувати в [веб-браузери](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1%D0%BE%D0%B3%D0%BB%D1%8F%D0%B4%D0%B0%D1%87), які підтримують [HTML5](https://uk.wikipedia.org/wiki/HTML5), повноцінну 3D-графіку, не вдаючись до посередництва [плагінів](https://uk.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B3%D1%96%D0%BD). Технологія дозволяє упроваджувати апаратно-прискорену 3D графіку у веб сторінки без необхідності використовувати спеціальні [плагіни](https://uk.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%B3%D1%96%D0%BD) веб-браузера на будь-якій платформі, що підтримує [OpenGL](https://uk.wikipedia.org/wiki/OpenGL). Технічно це буде прив'язкою скриптів [JavaScript](https://uk.wikipedia.org/wiki/JavaScript) до функцій, визначених в бібліотеках OpenGL ES 2.0, реалізованих на рівні браузера.

У WebGL текстури зазвичай завантажуються в графічний процесор і доступ до них здійснюється за допомогою UV (текстурних) координат. UV-координати використовуються для визначення положення точки на двовимірному зображенні текстури по відношенню до тривимірного об’єкта, до якого вона застосована. Таким чином, WebGL надає досить потужний набір інструментів для створення різноманітної інтерактивної графіки та анімації у сучасних веб-застосунках.  
 Для того щоб додати текстуру потрібно зробити деякі маніпуляції з вершиним шейдером, щоб він надалі використовував текстуру, а не кольори. Крім цього, як правило саме у вершиному шейдері, змінюється, якщо потрібно, координати текстури за допомогою масштабування, переносу та інших перетворень за допомогою матриць. Матричні перетворення та відображення текстури є двома важливими техніками, які можна використовувати для додання реалізму та деталізації 3D-моделей. Після невеликих змін у вершинному шейдері необхідно завантажити текстуру використовуючи JavaScript з Інтернету та зв'язати її з буфером координат текстури, що обчислюються під час обчислення координат поверхні.

**Виконання роботи**

1. **Система контролю версій**Перед початком роботи було створено гілку в GIT з назвою CGW, для зручного зберігання та контролю версій програми.
2. **Текстури**  
   Будь-яке цифрове зображення можна використовувати як текстура, але якщо ширина та висота зображення не є ступенем 2, існують обмеження щодо використання зображення. Найкраще, щоб розміри зображення дорівнювали степеню 2, що робить відображення текстур більш ефективним і знімає будь-які обмеження на його використання. Таким чином, розмір зображення має бути 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 або 2048 пікселів.  
   Було обрано нейтральну, неодноманітну, не чорну текстуру (щоб не спутати з градієнтом, фоном чи заливкою кольором). При виборі текстури потрібно пам’ятати про деякі нюанси. Власні фото, та деякі зображення не можуть бути відображені в сучасних браузерах через те, що не можуть пройти відповідно перевірку безпеки (що може стати сюрпризом для розробника). Також потрібно оминати заблоковані або недоступні веб-ресурси. Текстури з таких ресурсів не будуть правильно відображатися. В даному випадку було обрано текстуру , що зображена на рисунку 1.

A picture containing text, outdoor object

Description automatically generated

Рис.1 Обрана текстура

1. **Координати текстури**  
   Координати призначені кожній вершині моделі, керують відображенням кольорів зображення на гранях полігону. Ідея текстурних координат проста для розуміння, але призначення текстурних координат окремим вершинам може стати нетривіальною задачею.

Координата текстури — це два числа, які представляють місце(координату) на зображенні. Перше число, який називається s, є відсотком від лівої сторони зображення. Другий дріб, який називається t, є відсотком від нижньої частини зображення. (Система координат для зображення в WebGL має початок у нижньому лівому куті з віссю +S праворуч і віссю +T вгору.) На рисунку 2 є кілька прикладів координат текстури.

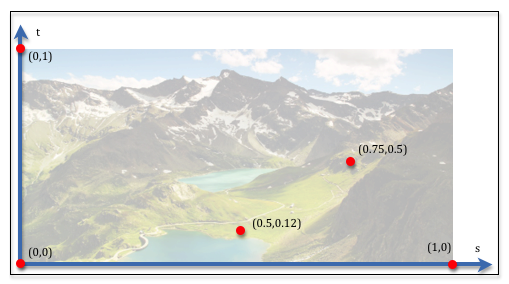


Рис. 2 приклад координат текстур

1. **Створення об’єкту текстури**

Коли ми візуалізуємо модель, ми хочемо, щоб дані моделі зберігалися в пам’яті графічного процесора, щоб вони були безпосередньо доступні для шейдерної програми. Об’єкт текстури зберігає зображення та всі пов’язані змінні стану, необхідні для створення відображення текстури. Ви можете створити стільки текстурних об’єктів, скільки має пам’ять графічний процесор.

Створення об’єкта текстури складається з трьох основних кроків:

* Створіть новий об'єкт текстури
* Встановіть параметри, які керують використанням об’єкта текстури.
* Скопіюйте зображення в об’єкт текстури

Функція на рисунку 3 створює об’єкт текстури. В коментарях були записані деякі замітки та пояснення до коду.

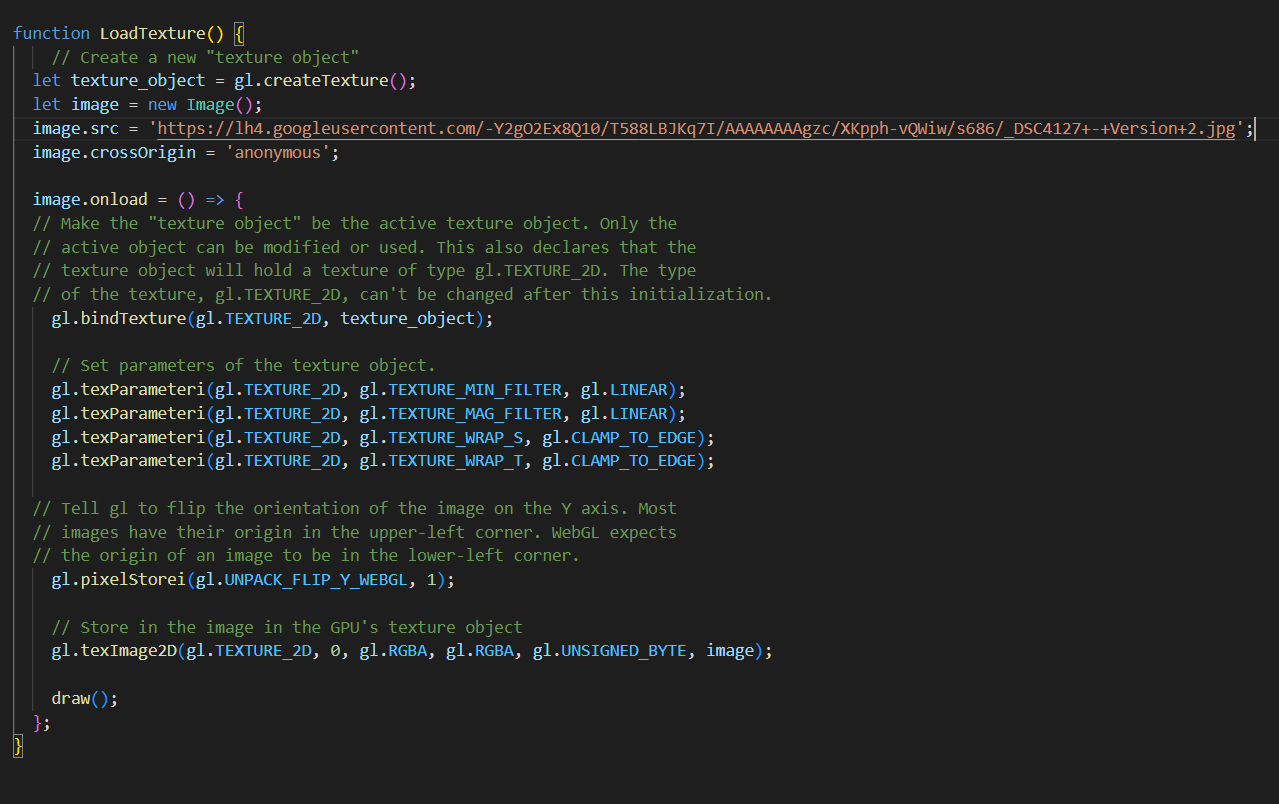
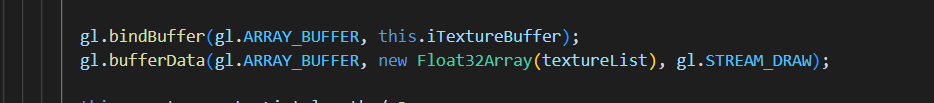


Рис. 3 Створення об’єкту текстури.

1. **Буфер координат**

Наступним кроком потрібно створити буфер для координат текстури, що будуть обраховані під час створення координат поверхні (рисунок 4),

Рис. 4 Буфер координат

Та забіндити (прив’язати) буфер до атрибуту (рисунок 5)

Text

Description automatically generatedРис. 5 Прив’язка буферу до атрибуту

1. **Переміщення точки вздовж простору поверхні (u,v)**

Тепер потрібно створити функцію для переміщення точки яку задає користувач, у vertex шейдері та функцію для масштабування текстури навколо точки що задає користувач (рисунок 6).

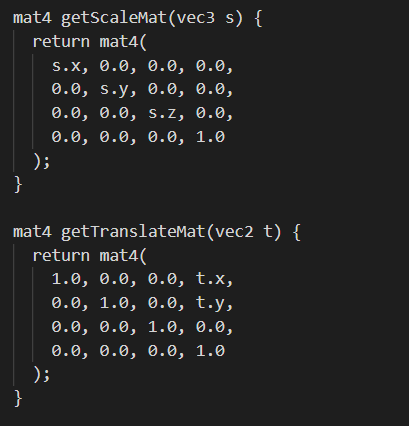


Рис. 6 Функції переміщення точки

**Результат роботи**

На рисунку 7 зображено фігуру, результат виконання розрахунково графічної роботи з нанесення текстури на поверхню.

Graphical user interface

Description automatically generated with medium confidence

Рис. 7 Поверхня з накладеною текстурою

На рисунку 8 зображено поверхню з нанесеною та зміненою текстурою. Були змінені параметри точки користувача навколо якої відбувається масштабування

A picture containing schematic

Description automatically generated

Рис. 8 Поверхня зі зміненими параметрами

**Код**

'use strict';

let gl;                         // The webgl context.

let surface;                    // A surface model

let shProgram;                  // A shader program

let spaceball;                  // A SimpleRotator object that lets the user rotate the view by mouse.

let point = { u: 0, v: 0 };

let b = 3

let c = 2

let d = 4

let X = (u,v) => (0.05 \* (f(a, b, v) \* (1 + Math.cos(u) + (d \*\* 2 - c \*\* 2) \* ((1 - Math.cos(u)) / f(a, b, v)))) \* Math.cos(v));

let Y = (u,v) => (0.05 \* (f(a, b, v) \* (1 + Math.cos(u) + (d \*\* 2 - c \*\* 2) \* ((1 - Math.cos(u)) / f(a, b, v)))) \* Math.sin(v));

let Z = (u,v) => (0.05 \* (f(a, b, v) - (d \*\* 2 - c \*\* 2) / f(a, b, v)) \* Math.sin(u));

function f(a, b, j) {

  return ((a \* b) / (Math.sqrt(a \*\* 2 \* Math.sin(j) \*\* 2 + b \*\* 2 \* Math.cos(j) \*\* 2)))

}

// let X = (u,v) => (R + a \* Math.cos(u/2)) \* Math.cos(u/3) + a \* Math.cos(u/3) \* Math.cos(v - Math.PI);

// let Y = (u,v) => (R + a \* Math.cos(u/2)) \* Math.sin(u/3) + a \* Math.sin(u/3) \* Math.cos(v - Math.PI);

// let Z = (u,v) => a + Math.sin(u/2) + a \* Math.sin(v - Math.PI);

function deg2rad(angle) {

    return angle \* Math.PI / 180;

}

let pValue = 0;

const getParabolaCords = () => {

  const p = Math.sin(pValue) \* 2.5;

  return [p, 10, (-10 + (p \* p))];

}

// Constructor

function Model(name) {

    this.name = name;

    this.iVertexBuffer = gl.createBuffer();

    this.iNormalsBuffer = gl.createBuffer();

    this.iTextureBuffer = gl.createBuffer();

    this.count = 0;

    this.BufferData = function({ vertexList, normalsList, textureList }) {

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

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

        gl.bindBuffer(gl.ARRAY\_BUFFER, this.iNormalsBuffer)

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

        gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTextureBuffer);

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

        this.count = vertexList.length / 3;

    }

    this.Draw = function() {

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

        gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0, 0);

        gl.enableVertexAttribArray(shProgram.iAttribVertex);

        gl.bindBuffer(gl.ARRAY\_BUFFER, this.iNormalsBuffer);

        gl.vertexAttribPointer(shProgram.iNormalsVertex, 3, gl.FLOAT, true, 0, 0);

        gl.enableVertexAttribArray(shProgram.iNormalsVertex);

        gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTextureBuffer);

        gl.enableVertexAttribArray(shProgram.iTextureCoords);

        gl.vertexAttribPointer(shProgram.iTextureCoords, 2, gl.FLOAT, false, 0, 0);

        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.iNormalsVertex = -1;

    // Location of the uniform specifying a color for the primitive.

    this.iColor = -1;

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

    this.iModelViewProjectionMatrix = -1;

    this.iViewWorldPosition = -1;

    this.iWMatrix = -1;

    this.iWInverseTranspose = -1;

    this.iLightWorldPosition = -1;

    this.iLightDir = -1;

    this.iTextureCoords = -1;

    this.iTMU = -1;

    this.iFScale = -1;

    this.iFPoint = -1;

    this.Use = function() {

        gl.useProgram(this.prog);

    }

}

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

 \* (Note that the use of the above drawPrimitive function is not an efficient

 \* way to draw with WebGL.  Here, the geometry is so simple that it doesn't matter.)

 \*/

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 \*/

    const projection = m4.perspective(Math.PI/8, 1, 8, 12);

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

    const modelView = spaceball.getViewMatrix();

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

    const matAccum1 = m4.multiply(translateToPointZero, modelView );

    /\* Multiply the projection matrix times the modelview matrix to give the

       combined transformation matrix, and send that to the shader program. \*/

    const modelViewProjection = m4.multiply(projection, matAccum1);

    const worldInverseMatrix = m4.inverse(matAccum1);

    const worldInverseTransposeMatrix = m4.transpose(worldInverseMatrix);

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

    gl.uniform3fv(shProgram.iViewWorldPosition, [0, 0, 0]);

    gl.uniform3fv(shProgram.iLightWorldPosition, getParabolaCords());

    gl.uniform3fv(shProgram.iLightDir, [0, -1, 0]);

    gl.uniformMatrix4fv(shProgram.iWInverseTranspose, false, worldInverseTransposeMatrix);

    gl.uniformMatrix4fv(shProgram.iWMatrix, false, matAccum1);

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

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

    const scaleU = document.getElementById('textureScaleU').value;

    const scaleV = document.getElementById('textureScaleV').value;

    console.log(scaleU, scaleV, point)

    gl.uniform2fv(shProgram.iFScale, [scaleU, scaleV]);

    gl.uniform2fv(shProgram.iFPoint, [X(point.u, point.v), Y(point.u, point.v)]);

    gl.uniform1i(shProgram.iTMU, 0);

    surface.Draw();

}

function CreateSurfaceData() {

    const vertexList = [];

    const normalsList = [];

    const textureList = [];

    const dU = 0.001;

    const dV = 0.001;

    for (let u = diapazonUFrom; u <= diapazonUTo; u += step) {

        for (let v = diapazonVFrom; v <= diapazonVTo; v += step) {

            const u0 = u;

            const v0 = v;

            const u1 = u + step;

            const v1 = v + step;

            let x0 = X(u0, v0);

            let y0 = Y(u0, v0);

            let z0 = Z(u0, v0);

            let xR = X(u1, v0);

            let yR = Y(u1, v0);

            let zR = Z(u1, v0);

            vertexList.push(x0, z0, y0);

            vertexList.push(xR, zR, yR);

            normalsList.push(...m4.cross(calcDerU(u0, v0, dU), calcDerV(u0, v0, dV)))

            normalsList.push(...m4.cross(calcDerU(u1, v0, dU), calcDerV(u1, v0, dV)));

            textureList.push(...calcTextureUV(u0, v0));

            textureList.push(...calcTextureUV(u1, v1));

        }

    }

    return { vertexList, normalsList, textureList };

}

const calcDerU = (u, v, dU) => ([

  (X(u + dU, v) - X(u, v)) / deg2rad(dU),

  (Y(u + dU, v) - Y(u, v)) / deg2rad(dU),

  (Z(u + dU, v) - Z(u, v)) / deg2rad(dU),

])

const calcDerV = (u, v, dV) => ([

  (X(u, v + dV) - X(u, v)) / deg2rad(dV),

  (Y(u, v + dV) - Y(u, v)) / deg2rad(dV),

  (Z(u, v + dV) - Z(u, v)) / deg2rad(dV),

])

const calcTextureUV = (u, v) => ([u / diapazonUTo, v / diapazonVTo]);

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

function initGL() {

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

    shProgram = new ShaderProgram('Basic', prog);

    shProgram.Use();

    shProgram.iModelViewProjectionMatrix = gl.getUniformLocation(prog, "ModelViewProjectionMatrix");

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

    shProgram.iNormalsVertex             = gl.getAttribLocation(prog, "normal");

    shProgram.iColor                     = gl.getUniformLocation(prog, "color");

    shProgram.iWInverseTranspose         = gl.getUniformLocation(prog, "wInverseTranspose");

    shProgram.iWMatrix                   = gl.getUniformLocation(prog, "wMatrix");

    shProgram.iViewWorldPosition         = gl.getUniformLocation(prog, "ViewWorldPosition");

    shProgram.iLightWorldPosition        = gl.getUniformLocation(prog, "LightWorldPosition");

    shProgram.iLightDir                  = gl.getUniformLocation(prog, "lightDir");

    shProgram.iTextureCoords             = gl.getAttribLocation(prog, 'textureCoords');

    shProgram.iTMU                       = gl.getUniformLocation(prog, 'tmu');

    shProgram.iFScale                    = gl.getUniformLocation(prog, 'fScale');

    shProgram.iFPoint                    = gl.getUniformLocation(prog, 'fPoint');

    surface = new Model('Surface');

    surface.BufferData(CreateSurfaceData());

    LoadTexture();

    gl.enable(gl.DEPTH\_TEST);

}

const reDraw = () => {

  surface.BufferData(CreateSurfaceData());

  draw();

}

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

 \* identifier for that program.  If an error occurs while compiling or

 \* linking the program, an exception of type Error is thrown.  The error

 \* string contains the compilation or linking error.  If no error occurs,

 \* the program identifier is the return value of the function.

 \* The second and third parameters are strings that contain the

 \* source code for the vertex shader and for the fragment shader.

 \*/

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() {

    let canvas;

    try {

        canvas = document.getElementById("webglcanvas");

        gl = canvas.getContext("webgl");

        if ( ! gl ) {

            throw "Browser does not support WebGL";

        }

    }

    catch (e) {

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

            "<p>Sorry, could not get a WebGL graphics context.</p>";

        return;

    }

    try {

        initGL();  // initialize the WebGL graphics context

    }

    catch (e) {

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

            "<p>Sorry, could not initialize the WebGL graphics context: " + e + "</p>";

        return;

    }

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

    draw();

}

window.addEventListener("keydown", (event) => {

  switch (event.key) {

    case 'ArrowLeft':

      pValue -= 0.1;

      draw();

      break;

    case 'ArrowRight':

      pValue += 0.1;

      draw();

      break;

    case 'w':

      point.v = point.v + step;

      draw();

      break;

    case 's':

      point.v = point.v - step;

      draw();

      break;

    case 'd':

      point.u = point.u + step;

      draw();

      break;

    case 'a':

      point.u = point.u - step;

      draw();

      break;

    default:

      break;

  }

});

function LoadTexture() {

    // Create a new "texture object"

  let texture\_object = gl.createTexture();

  let image = new Image();

  image.src = 'https://lh4.googleusercontent.com/-Y2gO2Ex8Q10/T588LBJKq7I/AAAAAAAAgzc/XKpph-vQWiw/s686/\_DSC4127+-+Version+2.jpg';

  image.crossOrigin = 'anonymous';

  image.onload = () => {

  // Make the "texture object" be the active texture object. Only the

  // active object can be modified or used. This also declares that the

  // texture object will hold a texture of type gl.TEXTURE\_2D. The type

  // of the texture, gl.TEXTURE\_2D, can't be changed after this initialization.

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

    // Set parameters of the texture object.

    gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MIN\_FILTER, gl.LINEAR);

    gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_MAG\_FILTER, gl.LINEAR);

    gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.CLAMP\_TO\_EDGE);

    gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.CLAMP\_TO\_EDGE);

  // Tell gl to flip the orientation of the image on the Y axis. Most

  // images have their origin in the upper-left corner. WebGL expects

  // the origin of an image to be in the lower-left corner.

    gl.pixelStorei(gl.UNPACK\_FLIP\_Y\_WEBGL, 1);

    // Store in the image in the GPU's texture object

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

    draw();

  };

}

// Vertex shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec3 normal;

uniform mat4 ModelViewProjectionMatrix;

uniform mat4 wInverseTranspose;

uniform mat4 wMatrix;

uniform vec3 LightWorldPosition;

uniform vec3 ViewWorldPosition;

varying vec3 v\_normal;

varying vec3 v\_surfaceToLight;

varying vec3 v\_surfaceToView;

attribute vec2 textureCoords;

uniform vec2 fPoint;

uniform vec2 fScale;

varying vec2 v\_textureCoords;

mat4 getScaleMat(vec3 s) {

return mat4(

s.x, 0.0, 0.0, 0.0,

0.0, s.y, 0.0, 0.0,

0.0, 0.0, s.z, 0.0,

0.0, 0.0, 0.0, 1.0

);

}

mat4 getTranslateMat(vec2 t) {

return mat4(

1.0, 0.0, 0.0, t.x,

0.0, 1.0, 0.0, t.y,

0.0, 0.0, 1.0, 0.0,

0.0, 0.0, 0.0, 1.0

);

}

void main() {

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

v\_normal = mat3(wInverseTranspose) \* normal;

vec3 surfaceWorldPosition = (wMatrix \* vec4(vertex, 1.0)).xyz;

v\_surfaceToLight = LightWorldPosition - surfaceWorldPosition;

v\_surfaceToView = ViewWorldPosition - surfaceWorldPosition;

mat4 scaleMat = getScaleMat(vec3(fScale, .0));

mat4 translateMat = getTranslateMat(-fPoint);

mat4 translateBackMat = getTranslateMat(fPoint);

vec4 textureCoordTr = translateMat \* vec4(textureCoords, 0, 0);

vec4 textureCoordRotate = textureCoordTr \* scaleMat;

vec4 textureCoordTrBack = textureCoordRotate \* translateBackMat;

v\_textureCoords = vec2(textureCoordTrBack);

}`;

// Fragment shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

precision highp float;

#else

precision mediump float;

#endif

uniform vec4 color;

uniform vec3 lightDir;

varying vec3 v\_normal;

varying vec3 v\_surfaceToView;

varying vec3 v\_surfaceToLight;

uniform sampler2D tmu;

varying vec2 v\_textureCoords;

void main() {

vec3 normal = normalize(v\_normal);

float shininess = 100.0;

vec3 surfaceToLightDirection = normalize(v\_surfaceToLight);

vec3 surfaceToViewDirection = normalize(v\_surfaceToView);

vec3 halfVector = normalize(surfaceToLightDirection + surfaceToViewDirection);

float dotDir = dot(surfaceToLightDirection, -lightDir);

float light = dot(normal, surfaceToLightDirection);

float specular = 0.0;

if (light > 0.0) {

specular = pow(dot(normal, halfVector), shininess);

}

vec4 texture = texture2D(tmu, v\_textureCoords);

gl\_FragColor = texture \* color;

gl\_FragColor.rgb \*= light;

gl\_FragColor.rgb += specular;

}`;