МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Кафедра цифрових технологій в енергетиці

Звіт з графічно-розрахункової роботи

з дисципліни «Візуалізація графічної та геометричної інформації»

Виконав:

Студент 1-го курсу магістратури

Інституту атомної та теплової енергетики

Групи ТР-22мп  
Дулєбов Ілля Ігорович

Прийняв:

Демчишин А. А.

Київ – 2022

**Постановка задачі**

**Варіант:** 27 **-** непарний варіант, реалізувати масштабування.

**Мета роботи:** засвоїти навички роботи з текстурами в WebGL, відобразити текстуру на побудованій поверхні та реалізувати операцію масштабування текстурних координат за варіантом.

**Вимоги:**

* Нанести текстуру на поверхню з практичного завдання №2.
* Реалізувати масштабування текстури (координати текстури) відповідно до визначеної користувачем точки.
* Реалізувати переміщення точку вздовж простору поверхні (u,v) за допомогою клавіатури. Клавіші A і D переміщують точку вздовж параметра u, а клавіші W і S переміщують точку вздовж параметра v.
* Реалізувати зміну значення масштабування використовуючи + для збільшення та — для зменшення.

**Теоретичні відомості**

Текстурування — накладання растрових зображень (текстур) на 3D-модель для надання рельєфності, фактури та кольору. Воно широко застосовується у відеоігровій індустрії, дизайні та кіновиробництві при імітації реалістичних поверхонь різних об'єктів, ефектів старіння, впливу погодних умов, механічного впливу тощо.

Основні види текстурування

Найпростіший спосіб надати тривимірній моделі фактури - накласти на неї карту кольору. Це плоске растрове зображення, в якому ілюзорна геометрія текстури створюється завдяки комбінації темних та світлих ділянок, що відображають поглиблення та опуклість. Таке текстурування спрощує та прискорює моделювання об'єкта та його малювання. Якщо уважно розглянути готовий об'єкт (особливо під кутом), то буде помітно, що він абсолютно плоский.

Щоб надати 3D-моделі або віртуальній поверхні ілюзію рельєфності, використовуються такі види текстурування:

bump mapping - до колірної текстури застосовується одноканальна (чорно-біла) карта висот та джерело світла, які формують ефект западин та опуклостей за рахунок різного освітлення окремих ділянок;

normal mapping - у технології рельєфність текстури досягається за рахунок кольорової карти нормалей, RGB-канали якої перетворюються на три координати вектора, за яким розраховується освітленість поверхні;

parallax mapping — цей спосіб полягає у зміні координат окремих пікселів текстури, а також освітленні з урахуванням рельєфу картою нормалей, що дозволяє досягти максимально повної рельєфності поверхні;

MIP-текстурування – це накладення кількох копій однієї й тієї ж текстури з різною роздільною здатністю залежно від потужності відеокарти, близькості віртуального об'єкта до спостерігача та інших факторів.

Більше ресурсів споживає технологія displacement map (карта зміщень). До полігональної моделі додаються додаткові вершини, які зміщуються вище або нижче за площину полігону відповідно до карти висот. На відміну від попередніх методів, що створюють лише ілюзію рельєфності, карта усунення формує реальний рельєф текстури. Хоча загальна геометрія об'єкта не змінюється (кількість полігонів залишається тим самим), вона стає складнішою. Через високі витрати ресурсів на промальовування даний метод по можливості замінюється більш простими способами.

Так як у природі не існує ідеально рівних і гладких поверхонь, для текстурування 3D-моделей можна використовувати візуальний шум, що процедурно генерується. З його допомогою досягаються ефекти потертості, шорсткості, фактури різних матеріалів та поверхонь: грубої чи дрібної штукатурки, бетону, асфальту, шліфованого металу, апельсинової шкірки тощо.

**Опис деталей реалізації**

Для відображення текстури на поверхні було завантажено зображення за посиланням з інтернету використовуючи базовий інтерфейс JavaScript. Налаштування параметрів тектсури було встановлено на такі значення: TEXTURE\_WRAP\_S — REPEAT, TEXTURE\_WRAP\_T — REPEAT, TEXTURE\_MIN\_FILTER — LINEAR, TEXTURE\_MAG\_FILTER — LINEAR.

Обрахунок текстурних координат відповідно кожного вертекса реалізовано в основному циклі і здійснюється за допомогою нормалізації параметрів uv, за допомогою яких будується поверхня.

Також, були створені додаткові глобльані параметри які зберігають координати точки масштабування та значення масштабування, які можна змінювати за допомогою вводу з клавіатури. Дані значення надсилаються в шейдер кожного фрейму.

Крім того, в вертексному шейдері було створено функцію яка реалізує масштабування текстурних координат відповідно довільної точки, для мінімізації затрат циклів процесора на обрахевання та відправлення цих даних на графічний процесор.

**Інструкція користувача**

Для взаємодії з програмою реалізовано інтерфейс який включає в себе такі клавіші:

* WS – переміщення точки масштабування вздовж параметра v;
* AD – переміщення точки масштабування вздовж параметра u;
* +- – зміна значення масштабування.

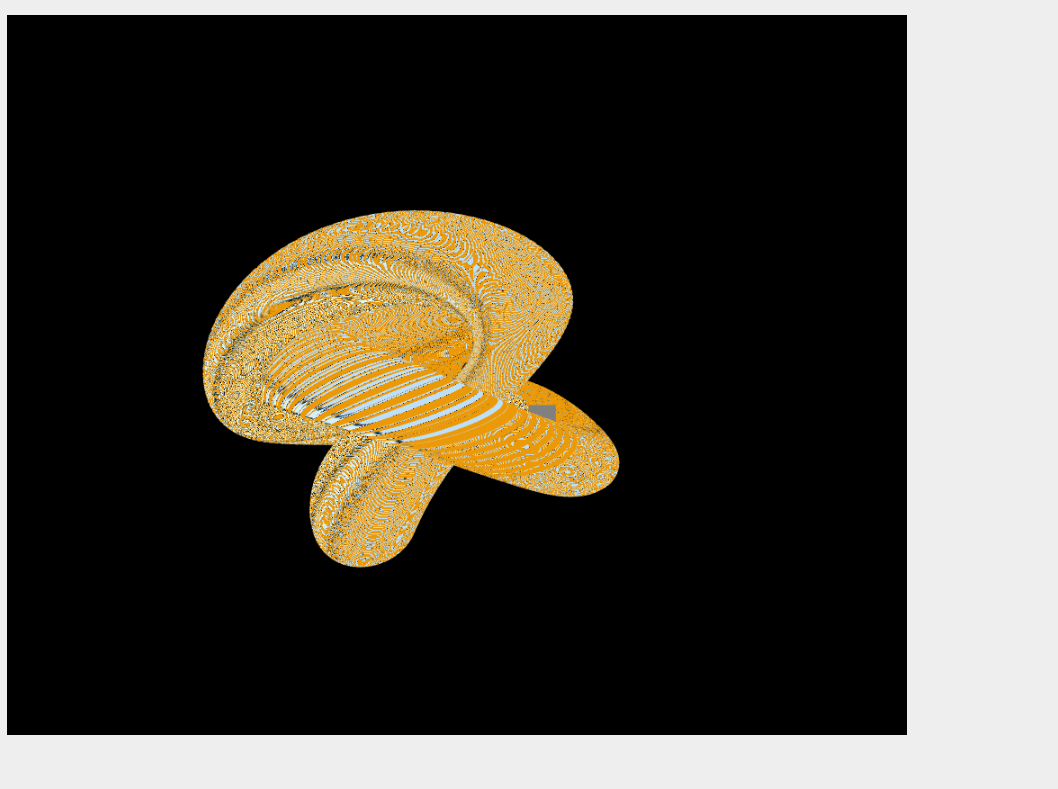


Рисунок 3 — Текстура нанесена на поверхню з значеннями за замовчуванням.

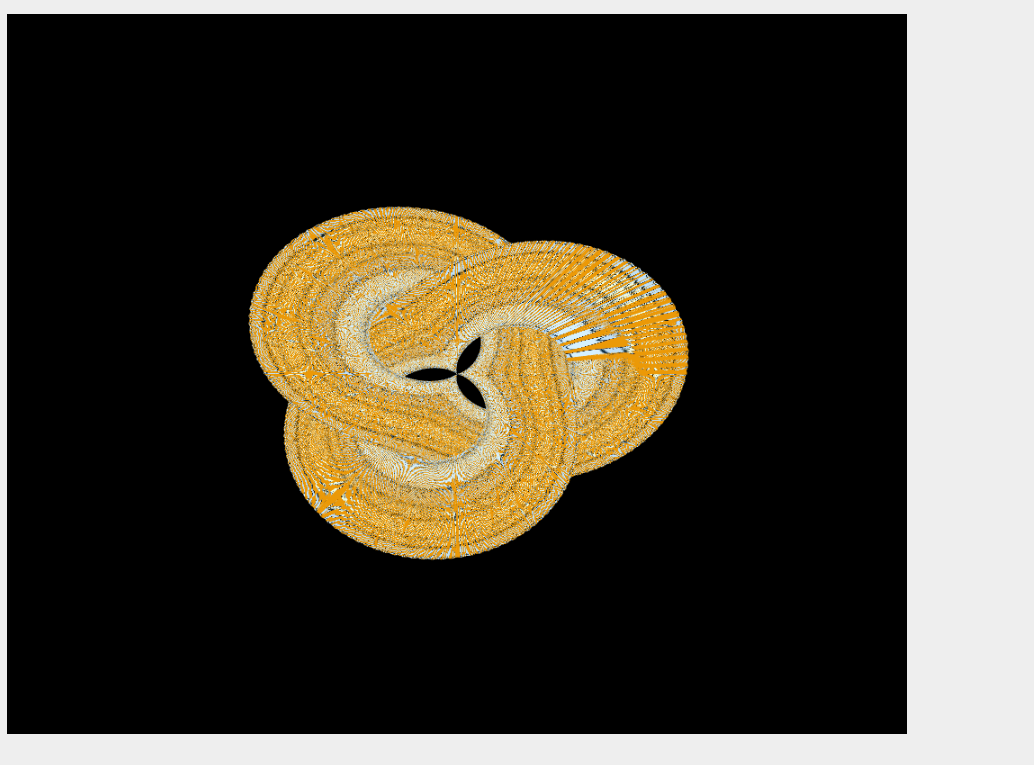


Рисунок 4 — Текстура нанесена на поверхню з довільними значеннями.

**Опис вихідного коду**

'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 InputCounter = 0.0;

let ScalePointLocationU = 0.0;

let ScalePointLocationV = 0.0;

let ControllerScaleValue = 1;

function deg2rad(angle) {

return angle \* Math.PI / 180;

}

// Constructor

function Model(name) {

this.name = name;

this.iVertexBuffer = gl.createBuffer();

this.iNormalBuffer = gl.createBuffer();

this.iTextureBuffer = gl.createBuffer();

this.iPointVertexBuffer = gl.createBuffer();

this.count = 0;

this.BufferData = function(vertices, normals, texCoords) {

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

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

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

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

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

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

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

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array([0, 0, 0]), gl.DYNAMIC\_DRAW);

this.count = vertices.length/3;

}

this.Draw = function() {

gl.uniform1i(shProgram.iDrawPoint, false);

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.iNormalBuffer);

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

gl.enableVertexAttribArray(shProgram.iNormalVertex);

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

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

gl.enableVertexAttribArray(shProgram.iTextureCoords);

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

// Draw point

gl.uniform1i(shProgram.iDrawPoint, true);

gl.uniform3fv(shProgram.iScalePointWorldLocation, [CalcX(ScalePointLocationU, ScalePointLocationV), CalcY(ScalePointLocationU, ScalePointLocationV), CalcZ(ScalePointLocationU, ScalePointLocationV)]);

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

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

gl.enableVertexAttribArray(shProgram.iAttribVertex);

gl.drawArrays(gl.POINTS, 0, 1);

}

}

// Constructor

function ShaderProgram(name, program) {

this.name = name;

this.prog = program;

this.iAttribVertex = -1;

this.iNormalVertex = -1;

this.iTextureCoords = -1;

this.iColor = -1;

this.iModelViewProjectionMatrix = -1;

this.iWorldMatrix = -1;

this.iWorldInverseTranspose = -1;

this.iLightWorldPosition = -1;

this.iLightDirection = -1;

this.iViewWorldPosition = -1;

this.iTexture = -1;

this.iScalePointLocation = -1;

this.iScaleValue = -1;

this.iDrawPoint = -1;

this.iScalePointWorldLocation = -1;

this.Use = function() {

gl.useProgram(this.prog);

}

}

function draw() {

gl.clearColor(0,0,0,1);

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

gl.enable(gl.CULL\_FACE);

// Enable the depth buffer

gl.enable(gl.DEPTH\_TEST);

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

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

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

let modelView = spaceball.getViewMatrix();

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

let matAccum1 = m4.multiply(WorldMatrix, modelView );

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

var worldInverseMatrix = m4.inverse(matAccum1);

var worldInverseTransposeMatrix = m4.transpose(worldInverseMatrix);

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

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

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

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

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

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

gl.uniform4fv(shProgram.iColor, [0.5,0.5,0.5,1] );

gl.uniform2fv(shProgram.iScalePointLocation, [ScalePointLocationU / 360.0, ScalePointLocationV / 90.0] );

gl.uniform1f(shProgram.iScaleValue, ControllerScaleValue);

gl.uniform1i(shProgram.iTexture, 0);

surface.Draw();

}

function CreateSurfaceData()

{

let step = 1.0;

let uend = 360 + step;

let vend = 90 + step;

let DeltaU = 0.0001;

let DeltaV = 0.0001;

let vertexList = [];

let normalsList = [];

let textCoords = [];

for (let u = 0; u < uend; u += step) {

for (let v = 0; v < vend; v += step) {

let unext = u + step;

/\*

\*-------\*

| |

| |

0-------\*

\*/

let x = CalcX(u, v);

let y = CalcY(u, v);

let z = CalcZ(u, v);

vertexList.push( x, y, z );

/\*

0-------\*

| |

| |

\*-------\*

\*/

x = CalcX(unext, v);

y = CalcY(unext, v);

z = CalcZ(unext, v);

vertexList.push( x, y, z );

// Normals

let DerivativeU = CalcDerivativeU(u, v, DeltaU);

let DerivativeV = CalcDerivativeV(u, v, DeltaV);

let result = m4.cross(DerivativeV, DerivativeU);

normalsList.push(result[0], result[1], result[2]);

DerivativeU = CalcDerivativeU(unext, v, DeltaU);

DerivativeV = CalcDerivativeV(unext, v, DeltaV);

result = m4.cross(DerivativeV, DerivativeU);

normalsList.push(result[0], result[1], result[2]);

textCoords.push(u / uend, v / vend);

textCoords.push(unext / uend, v / vend);

}

}

return [vertexList, normalsList, textCoords];

}

function CreatePointData()

{

let vertexList = [

];

let normalsList = [];

let textCoords = [];

}

function CalcX(u, v)

{

let uRad = deg2rad(u);

let vRad = deg2rad(v);

return vRad \* Math.cos(uRad);

}

function CalcY(u, v)

{

let uRad = deg2rad(u);

let vRad = deg2rad(v);

return vRad \* Math.sin(uRad);

}

function CalcZ(u, v)

{

let a = 1;

let b = 1;

let c = 1;

let uRad = deg2rad(u);

let vRad = deg2rad(v);

return c \* Math.sqrt(a \* a - (b \* b \* Math.cos(uRad) \* Math.cos(uRad)));

}

function CalcDerivativeU(u, v, uDelta)

{

let x = CalcX(u, v);

let y = CalcY(u, v);

let z = CalcZ(u, v);

let Dx = CalcX(u + uDelta, v);

let Dy = CalcY(u + uDelta, v);

let Dz = CalcZ(u + uDelta, v);

let Dxdu = (Dx - x) / deg2rad(uDelta);

let Dydu = (Dy - y) / deg2rad(uDelta);

let Dzdu = (Dz - z) / deg2rad(uDelta);

return [Dxdu, Dydu, Dzdu];

}

function CalcDerivativeV(u, v, vDelta)

{

let x = CalcX(u, v);

let y = CalcY(u, v);

let z = CalcZ(u, v);

let Dx = CalcX(u, v + vDelta);

let Dy = CalcY(u, v + vDelta);

let Dz = CalcZ(u, v + vDelta);

let Dxdv = (Dx - x) / deg2rad(vDelta);

let Dydv = (Dy - y) / deg2rad(vDelta);

let Dzdv = (Dz - z) / deg2rad(vDelta);

return [Dxdv, Dydv, Dzdv];

}

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

function initGL() {

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

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

shProgram.Use();

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

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

shProgram.iTextureCoords = gl.getAttribLocation(prog, "texcoord");

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

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

shProgram.iWorldInverseTranspose = gl.getUniformLocation(prog, "WorldInverseTranspose");

shProgram.iWorldMatrix = gl.getUniformLocation(prog, "WorldMatrix");

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

shProgram.iLightDirection = gl.getUniformLocation(prog, "LightDirection");

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

shProgram.iTexture = gl.getUniformLocation(prog, "u\_texture");

shProgram.iScalePointLocation = gl.getUniformLocation(prog, "ScalePointLocation");

shProgram.iScaleValue = gl.getUniformLocation(prog, "ScaleValue");

shProgram.iDrawPoint = gl.getUniformLocation(prog, "bDrawpoint");

shProgram.iScalePointWorldLocation = gl.getUniformLocation(prog, "ScalePointWorldLocation");

surface = new Model('Surface');

let SurfaceData = CreateSurfaceData();

surface.BufferData(SurfaceData[0], SurfaceData[1], SurfaceData[2]);

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

}

/\* 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() {

// Canvas

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;

}

// GL

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);

LoadTexture();

}

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

switch (event.key) {

case "ArrowLeft":

ProcessArrowLeftDown();

break;

case "ArrowRight":

ProcessArrowRightDown();

break;

case "W":

ProcessWDown();

break;

case "w":

ProcessWDown();

break;

case "S":

ProcessSDown();

break;

case "s":

ProcessSDown();

break;

case "A":

ProcessADown();

break;

case "a":

ProcessADown();

break;

case "D":

ProcessDDown();

break;

case "d":

ProcessDDown();

break;

case "+":

ProcessPlusDown();

break;

case "-":

ProcessSubtractDown();

break;

default:

break;

}

draw();

});

function ProcessArrowLeftDown()

{

InputCounter -= 0.05;

}

function ProcessArrowRightDown()

{

InputCounter += 0.05;

}

function CalcParabola()

{

let TParam = Math.sin(InputCounter) \* 1.2;

return [TParam, 6, -10 + (TParam \* TParam)];

}

function LoadTexture()

{

var texture = gl.createTexture();

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

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_S, gl.REPEAT);

gl.texParameteri(gl.TEXTURE\_2D, gl.TEXTURE\_WRAP\_T, gl.REPEAT);

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

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

gl.texImage2D(gl.TEXTURE\_2D, 0, gl.RGBA, 1, 1, 0, gl.RGBA, gl.UNSIGNED\_BYTE,

new Uint8Array([0, 0, 255, 255]));

var image = new Image();

image.crossOrigin = "anonymous"

image.src = "https://i1.photo.2gis.com/images/profile/30258560049997155\_fe3f.jpg";

image.addEventListener('load', function() {

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

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

console.log("Texture is loaded!");

draw();

});

}

function ProcessWDown()

{

ScalePointLocationV -= 5.0;

ScalePointLocationV = clamp(ScalePointLocationV, 0.0, 90);

}

function ProcessSDown()

{

ScalePointLocationV += 5.0;

ScalePointLocationV = clamp(ScalePointLocationV, 0.0, 90);

}

function ProcessADown()

{

ScalePointLocationU -= 5.0;

ScalePointLocationU = clamp(ScalePointLocationU, 0.0, 360);

}

function ProcessDDown()

{

ScalePointLocationU += 5.0;

ScalePointLocationU = clamp(ScalePointLocationU, 0.0, 360);

}

function ProcessPlusDown()

{

ControllerScaleValue += 0.05;

ControllerScaleValue = clamp(ControllerScaleValue, 0.5, 2.0);

}

function ProcessSubtractDown()

{

ControllerScaleValue -= 0.05;

ControllerScaleValue = clamp(ControllerScaleValue, 0.5, 2.0);

}

function clamp(value, min, max)

{

if(value < min)

{

value = min

}

else if(value > max)

{

value = max;

}

return value;

}