Ministry of Science of Ukraine

NATIONAL TECHNICAL UNIVERSITY OF UKRAINE
«IGOR SIKORSKY KYIV POLYTECHNIC INSTITUTE»
Cathedra of Design Automation of Energy Processes and Systems

Subject Operations on texture coordinates. Astroidal Torus

Calculation and graphics work

Academic discipline «Visualization of graphical and geometric information»

Student

Pereverziev Luka Ruslanovych

Kyiv – 2023

1. TASK DESCRIPTION

Requirements:

- Map the texture over the surface from practical assignment #2.
- Implement texture scaling (texture coordinates) scaling / rotation around user specified point- odd variants implement scaling, even variants implement rotation
- It has to be possible to move the point along the surface *(u,v)* space using a keyboard. E.g. keys **A** and **D** move the point along *u* parameter and keys **W** and **S** move the point along *v* parameter.
- The practical assignment has to reside in a branch named CGW.

2. THEORETICAL BACKGROUND

2.1. Astroidal Torus

A surface of the rotation of an astroid is formed by an astroid $x^{2/3} + z^{2/3} = a^{2/3}$ rotating about any of two its axes Ox or Oz. If an astroid $x = x(u) = a\cos^3 u, \quad z = z(u) = a\sin^3 u$ is placed at the r distant from the axis of rotation, then we will have an astroidal torus.

An inner area bounded by an astroid is $A = \frac{3}{8}\pi a^2$.

An astroidal torus can be defined by parametrical equations:

$$X = X(u,v) = [r + x(u)\cos\theta - z(u)\sin\theta]\cos v;$$
$$Y = Y(u,v) = [r + x(u)\cos\theta - z(u)\sin\theta]\sin v;$$
$$Z = Z(u) = x(u)\sin\theta + z(u)\cos\theta,$$

where θ is the angle of rotation of local axes x, z of the generatrix astroid in the vertical plane containing the axis. The local coordinate system is rotated counter-clockwise if the θ angle has positive value.

The astroidal torus with $\theta = 0,\ 0 \le v \le 2\pi,\ -\pi \le u \le \pi,$ a = r = 1 is given in the following figure:

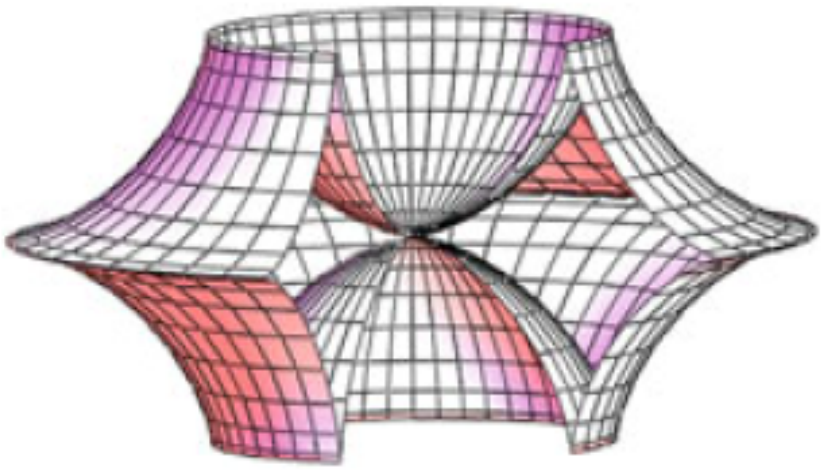

Fig. 2.1. Astroidal torus.

2.2. Texture mapping

Texture mapping is a method for mapping a texture on a computer-generated graphic. Texture can be high frequency detail, surface texture, or color. A texture map is an image applied (mapped) to the surface of a shape or polygon. This may be a bitmap image or a procedural texture. They may be stored in common image file formats, referenced by 3d model formats or material definitions, and assembled into resource bundles. To apply the texture, every vertex in a polygon is assigned a texture coordinate (which is also known as UV coordinates). The next step is projecting a 3D model's surface to a 2D image, which is called UV mapping. The UV mapping process at its simplest requires three steps: unwrapping the mesh, creating the texture, and applying the texture to a respective face of the polygon. The UV mapping process involves assigning pixels in the image to surface mappings on the polygon, usually done by

"programmatically" copying a triangular piece of the image map and pasting it onto a triangle on the object. A UV map can either be generated automatically by the software application, made manually by the artist, or some combination of both.

3. IMPLEMENTATION DETAILS

In this work, the WebGL API is used to model a given complex shape, namely, the asteroidal torus. Several polygon rendering options were considered, among which the triangle method was chosen as the most effective. Vertex and pixel shaders that calculate intensity of illumination for the surface coming from a single light source were implemented. The following figure depicts the asteroidal torus illuminated by a spotlight with the use of Phong shading.

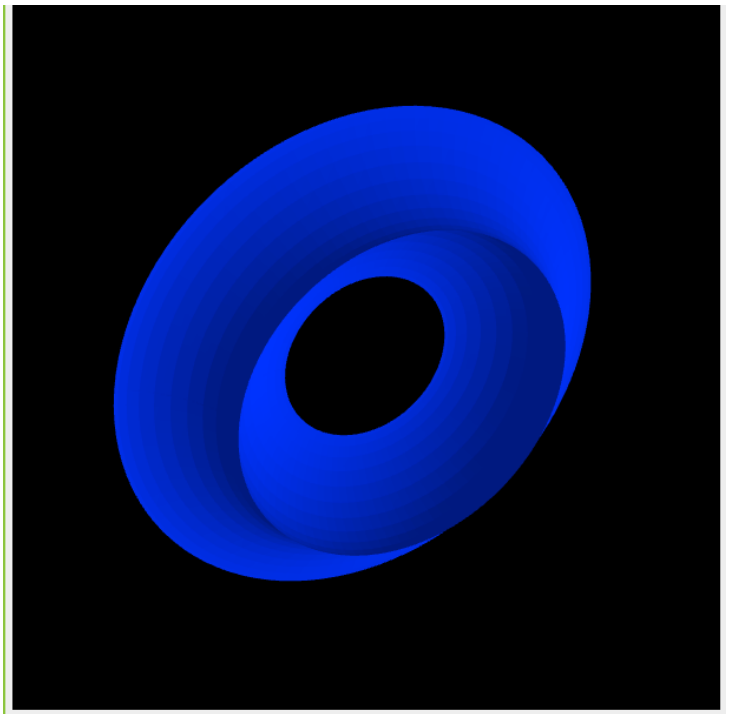

Fig. 3.1. Astroidal torus.

The source code of both the shaders and the program were modified to support texture mapping and texture scaling. This includes variables and functions used for matrix translating and matrix scaling; creating, buffering and applying textures; rotating the torus and moving the point of texture scaling with user input.

The following free 512x512 texture was chosen for painting the target shape and subsequently uploaded to GitHub for use in the project.

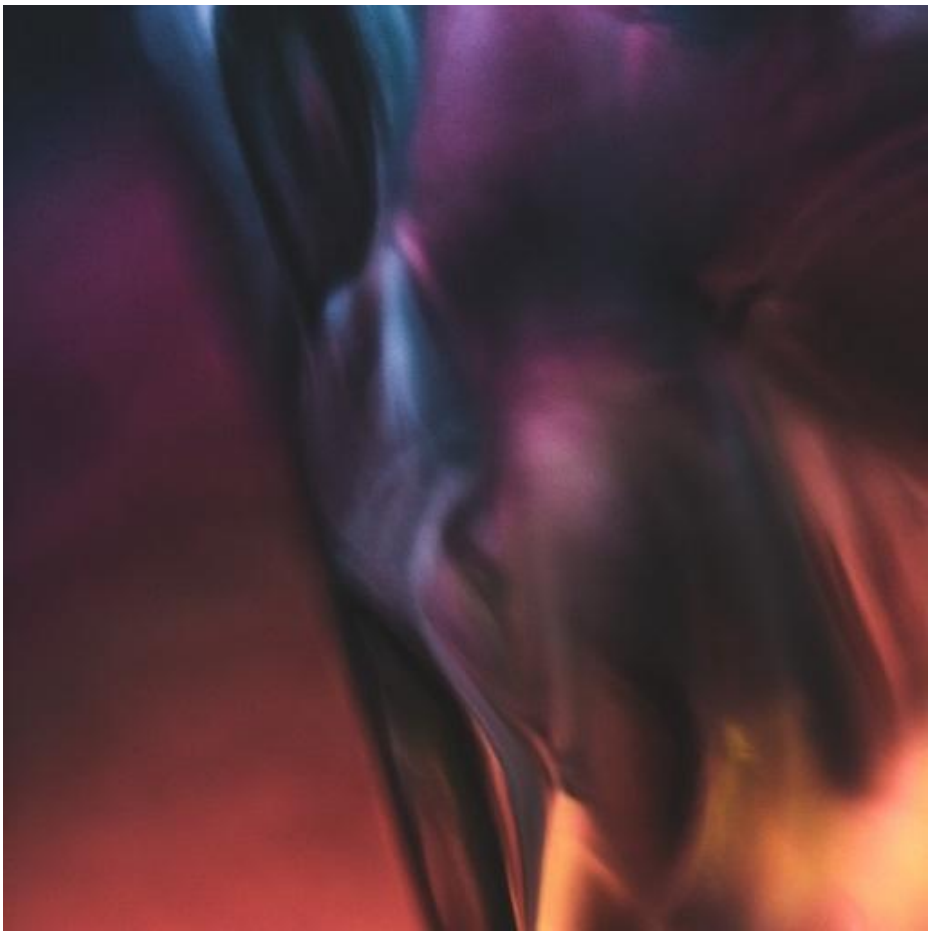

Fig. 3.2. Free texture image.

The following figure depicts the asteroidal torus painted with the texture above.

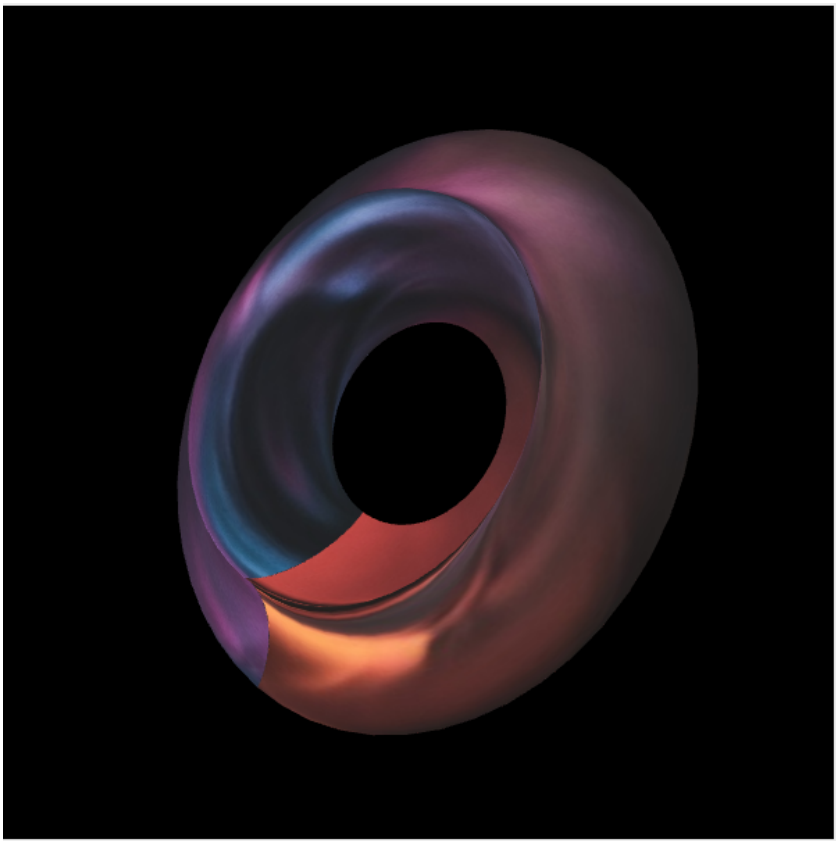

Fig. 3.3. Texture mapped over the asteroidal torus.

Texture scaling around the user specified point is achieved through specifying the target point using the WASD keys. The said point is highlighted in red, as shown in the figure below.

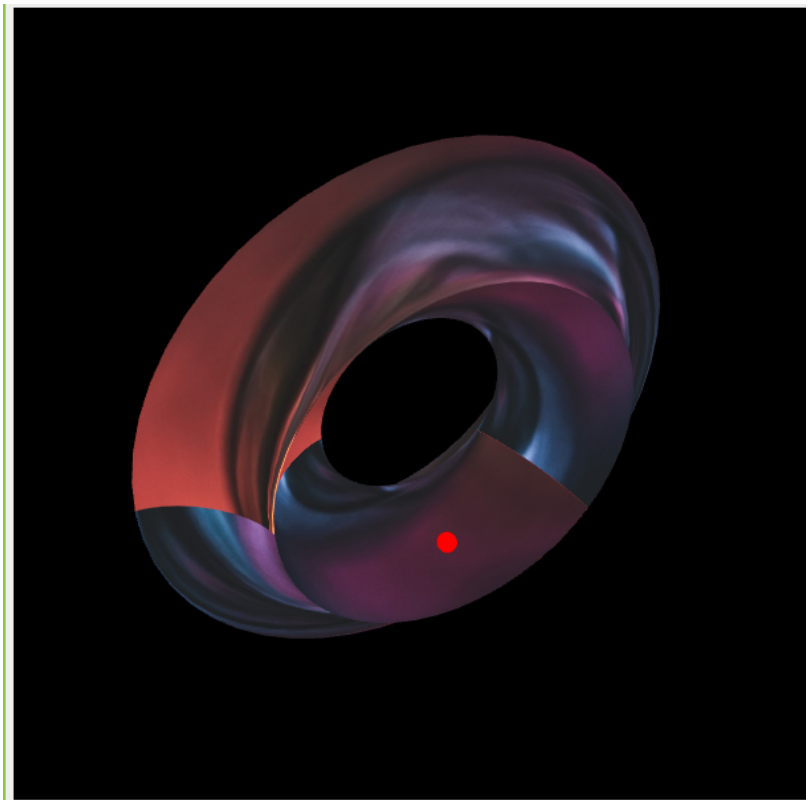

Fig. 3.4. Texture scaled over the target point.

4. USER INSTRUCTIONS

To view the final result, the user has to clone the repository https://github.com/RewindRewind/WebGL, switch to the “CGW” branch and run index.html, upon which the following webpage will open:

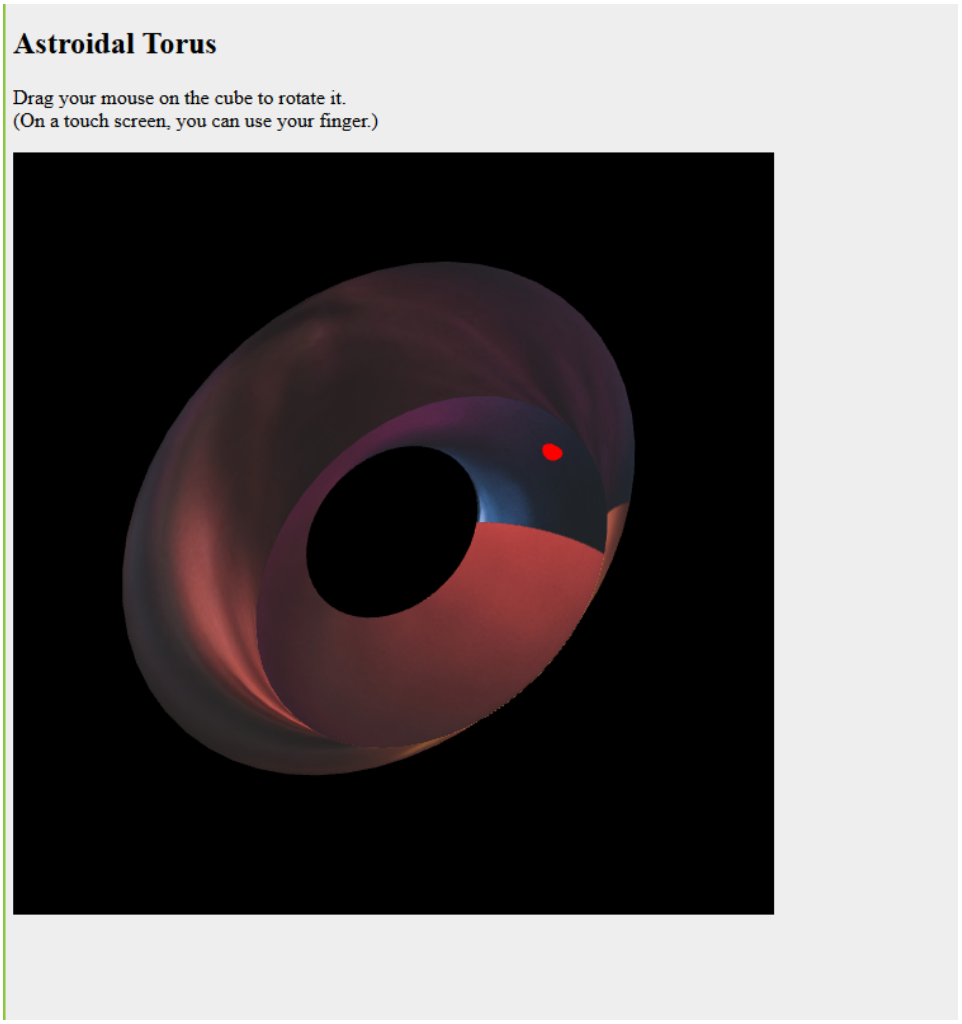

Fig. 4.1. index.html.

The final program’s use cases include:

- rotating the astroidal torus by dragging the mouse on the shape;
- moving the point of texture scaling by using WASD keys;
- scaling the texture by moving the cursor across the shape;

Each action is independent and can be performed simultaneously.

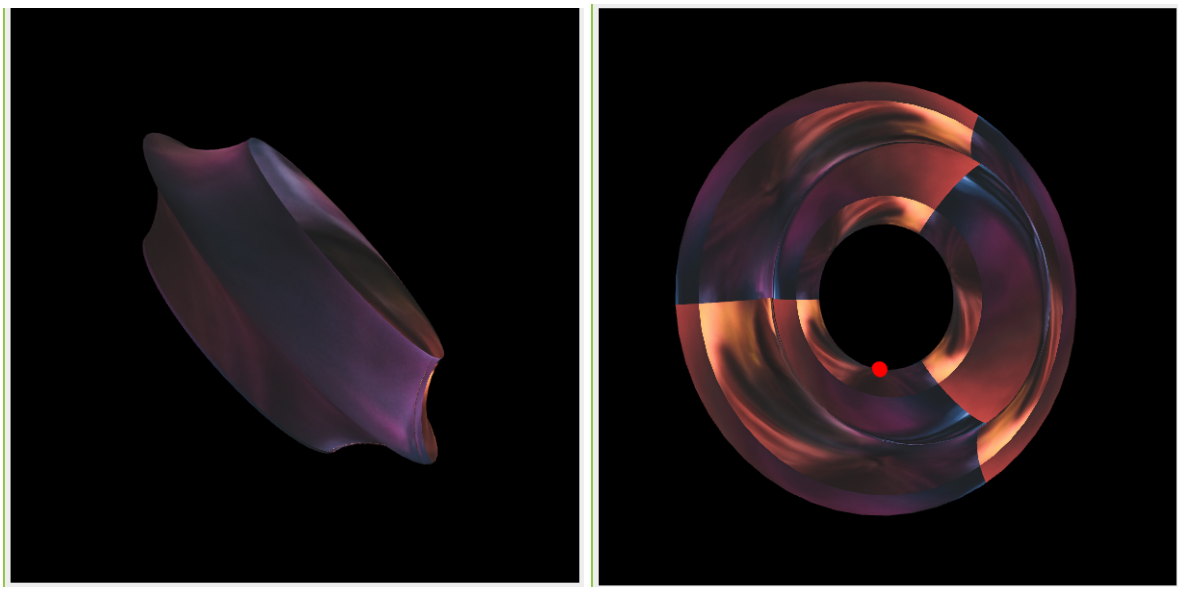

Fig. 4.1, 4.2. Visual demonstration of implemented functionality.

5. SOURCE CODE

main.js:

```
'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;
let texturePoint;
let scalingKoef;

function deg2rad(angle) {
    return angle * Math.PI / 180;
}


// Constructor
function Model(name) {
    this.name = name;
    this.iVertexBuffer = gl.createBuffer();
    this.iTextureBuffer = gl.createBuffer();
    this.count = 0;
    this.countTexture = 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 (normals) {

        gl.bindBuffer(gl.ARRAY_BUFFER, this.iTextureBuffer);
```

```
        gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normals),
gl.STREAM_DRAW);

        this.countTexture = normals.length/2;
    }

    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.iTextureBuffer);
        gl.vertexAttribPointer(shProgram.iAttribTexture, 2, gl.FLOAT, false,
0, 0);
        gl.enableVertexAttribArray(shProgram.iAttribTexture);

        gl.drawArrays(gl.TRIANGLE_STRIP, 0, this.count);
    }

    this.DisplayPoint = function () {
        gl.bindBuffer(gl.ARRAY_BUFFER, this.iVertexBuffer);
        gl.vertexAttribPointer(shProgram.iAttribVertex, 3, gl.FLOAT, false, 0,
0);
        gl.enableVertexAttribArray(shProgram.iAttribVertex);
        gl.drawArrays(gl.LINE_STRIP, 0, this.count);
    }
}

function CreateSphereSurface(r = 0.05) {
    let vertexList = [];
    let lon = -Math.PI;
    let lat = -Math.PI * 0.5;
    while (lon < Math.PI) {
        while (lat < Math.PI * 0.5) {
            let v1 = sphereSurfaceData(r, lon, lat);
            vertexList.push(v1.x, v1.y, v1.z);
            lat += 0.05;
        }
        lat = -Math.PI * 0.5;
```

```
        lon += 0.05;
    }
    return vertexList;
}

function sphereSurfaceData(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 };
}


// Constructor
function ShaderProgram(name, program) {

    this.name = name;
    this.prog = program;

    // Location of the attribute variable in the shader program.
    this.iAttribVertex = -1;
    this.iAttribTexture = -1;
    // Location of the uniform matrix representing the combined
transformation.
    this.iModelViewProjectionMatrix = -1;

    this.iTranslatePoint = -1;
    this.iTexturePoint = -1;
    this.iScalingKoef = -1;
    this.iTMU = -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 */
    let projection = m4.perspective(Math.PI/8,1,8,12);

    /* Get the view matrix from the SimpleRotator object.*/
    let modelView = spaceball.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(shProgram.iModelViewProjectionMatrix, false,
modelViewProjection);

    gl.uniform1i(shProgram.iTMU, 0);
    gl.enable(gl.TEXTURE_2D);
    gl.uniform2fv(shProgram.iTexturePoint, [texturePoint.x, texturePoint.y]);
    gl.uniform1f(shProgram.iScalingKoef, scalingKoef);
    surface.Draw();
    let tr = AstroidalTorus(map(texturePoint.x, 0, 1, 0,
Math.PI*2),map(texturePoint.y, 0, 1, 0, Math.PI*2))
    gl.uniform3fv(shProgram.iTranslatePoint, [tr.x, tr.y, tr.z]);
    gl.uniform1f(shProgram.iScalingKoef, -scalingKoef);
    point.DisplayPoint();
}

function CreateSurfaceData() {
    let vertexList = [];
    let uMax = Math.PI * 2;
    let vMax = Math.PI * 2;
    let uStep = uMax / 50;
    let vStep = vMax / 50;
```

```
    for (let u = 0; u <= uMax; u += uStep) {
        for (let v = 0; v <= vMax; v += vStep) {
            let vert = AstroidalTorus(u, v);
            let avert = AstroidalTorus(u + uStep, v);
            let bvert = AstroidalTorus(u, v + vStep);
            let cvert = AstroidalTorus(u + uStep, v + vStep);

            vertexList.push(vert.x, vert.y, vert.z);
            vertexList.push(avert.x, avert.y, avert.z);
            vertexList.push(bvert.x, bvert.y, bvert.z);

            vertexList.push(avert.x, avert.y, avert.z);
            vertexList.push(cvert.x, cvert.y, cvert.z);
            vertexList.push(bvert.x, bvert.y, bvert.z);
        }
    }

    return vertexList;
}



function CreateTexture() {
    let texture = [];
    let uMax = Math.PI * 2;
    let vMax = Math.PI * 2;
    let uStep = uMax / 50;
    let vStep = vMax / 50;

    for (let u = 0; u <= uMax; u += uStep) {
        for (let v = 0; v <= vMax; v += vStep) {
            let u1 = map(u, 0, uMax, 0, 1);
            let v1 = map(v, 0, vMax, 0, 1);
            texture.push(u1, v1);
            u1 = map(u + uStep, 0, uMax, 0, 1);
            texture.push(u1, v1);
            u1 = map(u, 0, uMax, 0, 1);
            v1 = map(v + vStep, 0, vMax, 0, 1);
            texture.push(u1, v1);
            u1 = map(u + uStep, 0, uMax, 0, 1);
            v1 = map(v, 0, vMax, 0, 1);
```

```
            texture.push(u1, v1);
            v1 = map(v + vStep, 0, vMax, 0, 1);
            texture.push(u1, v1);
            u1 = map(u, 0, uMax, 0, 1);
            v1 = map(v + vStep, 0, vMax, 0, 1);
            texture.push(u1, v1);
        }
    }

    return texture;
}

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 AstroidalTorus(u, v) {
    let r = 1;
    let a = 0.5;
    let t = 0.5 * Math.PI;
    let x = (r + a * Math.cos(u) ** 3 * Math.cos(t) - a * Math.sin(u) ** 3 *
Math.sin(t)) * Math.cos(v);
    let y = (r + a * Math.cos(u) ** 3 * Math.cos(t) - a * Math.sin(u) ** 3 *
Math.sin(t)) * Math.sin(v);
    let z = a * Math.cos(u) ** 3 * Math.sin(t) + Math.sin(u) ** 3 *
Math.cos(t);
    return { x: x, y: y, z: z }
}



/* 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.iAttribTexture = gl.getAttribLocation(prog, "texture");
```

```
    shProgram.iModelViewProjectionMatrix = gl.getUniformLocation(prog,
"ModelViewProjectionMatrix");
    shProgram.iTranslatePoint = gl.getUniformLocation(prog, 'translatePoint');
    shProgram.iTexturePoint = gl.getUniformLocation(prog, 'texturePoint');
    shProgram.iScalingKoef = gl.getUniformLocation(prog, 'scalingKoef');
    shProgram.iTMU = gl.getUniformLocation(prog, 'tmu');

    surface = new Model('AstroidalTorus');
    surface.BufferData(CreateSurfaceData());
    LoadTexture();
    surface.TextureBufferData(CreateTexture());
    point = new Model('Point');
    point.BufferData(CreateSphereSurface())

    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() {
    texturePoint = { x: 0.1, y: 0.1 }
    scalingKoef = 1.0;
    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();
}
```

```
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/RewindRewind/WebGL/CGW/texture.jpg";
    image.onload = () => {
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.texImage2D(
            gl.TEXTURE_2D,
            0,
            gl.RGBA,
            gl.RGBA,
            gl.UNSIGNED_BYTE,
            image
        );
        draw()
    }
}
window.onkeydown = (e) => {
    switch (e.keyCode) {
        case 87:
            texturePoint.x -= 0.02;
            break;
        case 83:
            texturePoint.x += 0.02;
            break;
        case 65:
            texturePoint.y += 0.02;
            break;
        case 68:
            texturePoint.y -= 0.02;
            break;
    }
    texturePoint.x = Math.max(0.001, Math.min(texturePoint.x, 0.999))
```

```
    texturePoint.y = Math.max(0.001, Math.min(texturePoint.y, 0.999))
    draw();
}

onmousemove = (e) => {
    scalingKoef = map(e.clientX, 0, window.outerWidth, 0, Math.PI)
    draw()
};
```

shader.gpu:

```
// Vertex shader
const vertexShaderSource = `
attribute vec3 vertex;
attribute vec2 texture;
uniform mat4 ModelViewProjectionMatrix;

varying vec2 v_texcoord;

uniform vec3 translatePoint;
uniform vec2 texturePoint;
uniform float scalingKoef;

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() {
    mat4 sMatrix = scaling(scalingKoef);
    mat4 t1Matrix =
translation(texturePoint.x,texturePoint.y,0.0);
    mat4 t2Matrix =
translation(-texturePoint.x,-texturePoint.y,0.0);
    vec4 texture1 = vec4(texture,0.0,0.0)*t1Matrix;
    vec4 textureRotated = texture1*sMatrix;
    vec4 texture2 = textureRotated*t2Matrix;
    v_texcoord = vec2(texture2.x,texture2.y);
    vec4 vertPos4 = ModelViewProjectionMatrix *
vec4(vertex, 1.0);
    gl_Position = vertPos4;
    if(scalingKoef<0.0){
    vec4 tPoint =
translation(translatePoint.x,translatePoint.y,transl
atePoint.z)*vec4(vertex,1.0);
    gl_Position = ModelViewProjectionMatrix*tPoint;
```

```
    }
}`;


// Fragment shader
const fragmentShaderSource = `
#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif

varying vec2 v_texcoord;
uniform sampler2D tmu;
uniform float scalingKoef;

void main() {
    //Applying texture
    vec4 texColor = texture2D(tmu, v_texcoord);
    gl_FragColor = texColor;
    if(scalingKoef<0.0){
    gl_FragColor = vec4(0.,0.,0.,0.);
    }
}`;
```

index.html:

```
<!DOCTYPE html>
<html><head><meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">



<title>CGW</title>
<style>
   body {
       background-color: #EEEEEE;
   }
</style>

<script src="./Utils/trackball-rotator.js"></script>
<script src="./Utils/m4.js"></script>
<script src="./shader.gpu"></script>
<script src="./main.js"></script>
```

```
</head>
<body onload="init()">

<h2>Astroidal Torus</h2>

<p id="message">Drag your mouse on the cube to rotate it.<br>
(On a touch screen, you can use your finger.)</p>


<div id="canvas-holder">
   <canvas width="600" height="600" id="webglcanvas"
style="background-color:red"></canvas>
</div>


</body></html>
```