National Technical University of Ukraine

“Igor Sikorsky Kyiv Polytechnic Institute”

Educational and Scientific Institute of Atomic and Thermal Energy

# **Visualization of graphical and geometric information**

Calculation and graphics work

(Operations on texture coordinates)

**Prepared by:**

group TR-23mp

Maksym Zhuk

Kyiv - 2023

# Description of the task

## Operations on texture coordinates

Scale and rotate a texture around user specified point.

## 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.

## Theory

### Astroidal Helicoid

A helical surface with a generatrix curve in the form of an astroid (Fig. 1) is given by the following parametrical equations:

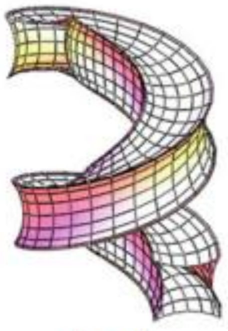


Fig. 1

$$x = x(u,v) = [a + x_0(v)\cos\vartheta + y_0(v)\sin\vartheta]\cos u;$$
$$y = y(u,v) = [a + x_0(v)\cos\vartheta + y_0(v)\sin\vartheta]\sin u;$$
$$z = z(u,v) = bu - x_0(v)\sin\vartheta + y_0(v)\cos\vartheta,$$

where $x_0(v) = c\cos^3 v,\ y_0(v) = c\sin^3 v,\ c$ is a radius of a

rolling circle with a point forming the astroid.

### Forms of definition of the Astroidal Helicoid

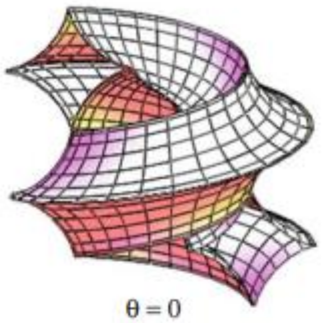
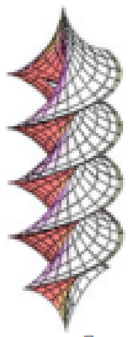


Fig. 2

A local axis yo intersects a helical axis at an angle θ. The

origin of the local system of Cartesian coordinates xo, yo is

located on the helical directrix within the a distance of the helical axis and moves along this axis in proportion to the angular velocity. If θ = 0 and b = c/π, then the coordinate lines v = 0 and v = π touch each other (Fig. 2). An interesting surface shown in Fig. 3 is formed when θ = 0 and a = 0.



$a = 0$

**Fig. 3**

## Implementation details

The Astroidal Helicoid is a type of mathematical surface that has minimal surface area for a given boundary. It is a continuous function of two variables, and it can be represented by a parametric equation in three-dimensional space.

To implement The Klein Surface minimal surface in WebGL, I did the following:

(1) Wrote a function to generate surface vertices: first wrote a function that generates surface vertices using the parametric equation for the Richmond minimum surface. This function takes two parameters, u and v, which represent the coordinates in the parametric domain of the surface. It returns an array of three-dimensional vertices, each represented as an array of three coordinates.

(2) Wrote a function to generate surface normals: Wrote a function that generates surface normals for each vertex. They are used to calculate the illumination on the surface.

(3) Wrote a function to generate surface texture coordinates: to display a texture on a surface, also wrote a function that generates texture coordinates for each vertex. The texture coordinates determine the display of the texture image on the surface.

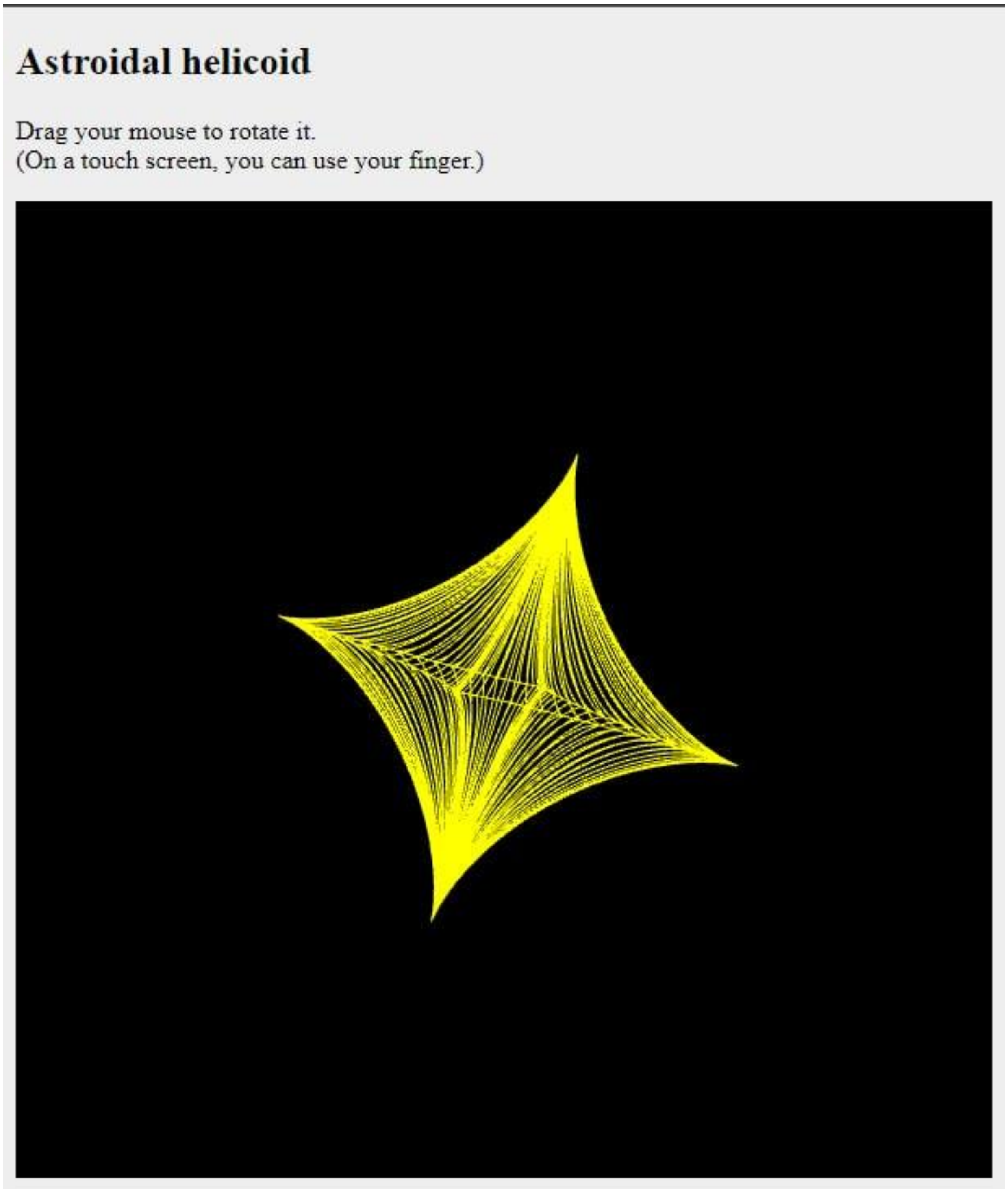
(4) Loaded the texture image: inserted a reference to the texture image using the image element, then created a texture object from the image using gl.createTexture().I also bound the texture object to the gl.TEXTURE_2D target using gl.bindTexture() and set the texture parameters using gl.texParameteri().

(5) Adjusted texture and coordinate vertex buffers: Created texture and coordinate vertex buffers and filled them with data using gl.bufferData(). I also bound the buffersand set the vertex attribute pointers using gl.vertexAttribPointer() to pass the data to the shader program.

(6) Test in the shader program: enabled texturing in the vertex and fragment shaders by declaring the sampler2D uniform and the vTexCoord attribute and using thetexture2D() function to sample the texture value in texture coordinates. Bind the textureobject to the texture block using gl.activeTexture() and gl.uniform1i() and passed the texture block to the shader program as a uniform.

(7) Drawn a surface with texturing enabled: In the Draw() method, enabled texturing in the WebGL backend by enabling the gl.TEXTURE_2D capability an

# User's instruction with screenshots



## Astroidal helicoid

Drag your mouse to rotate it.
(On a touch screen, you can use your finger.)

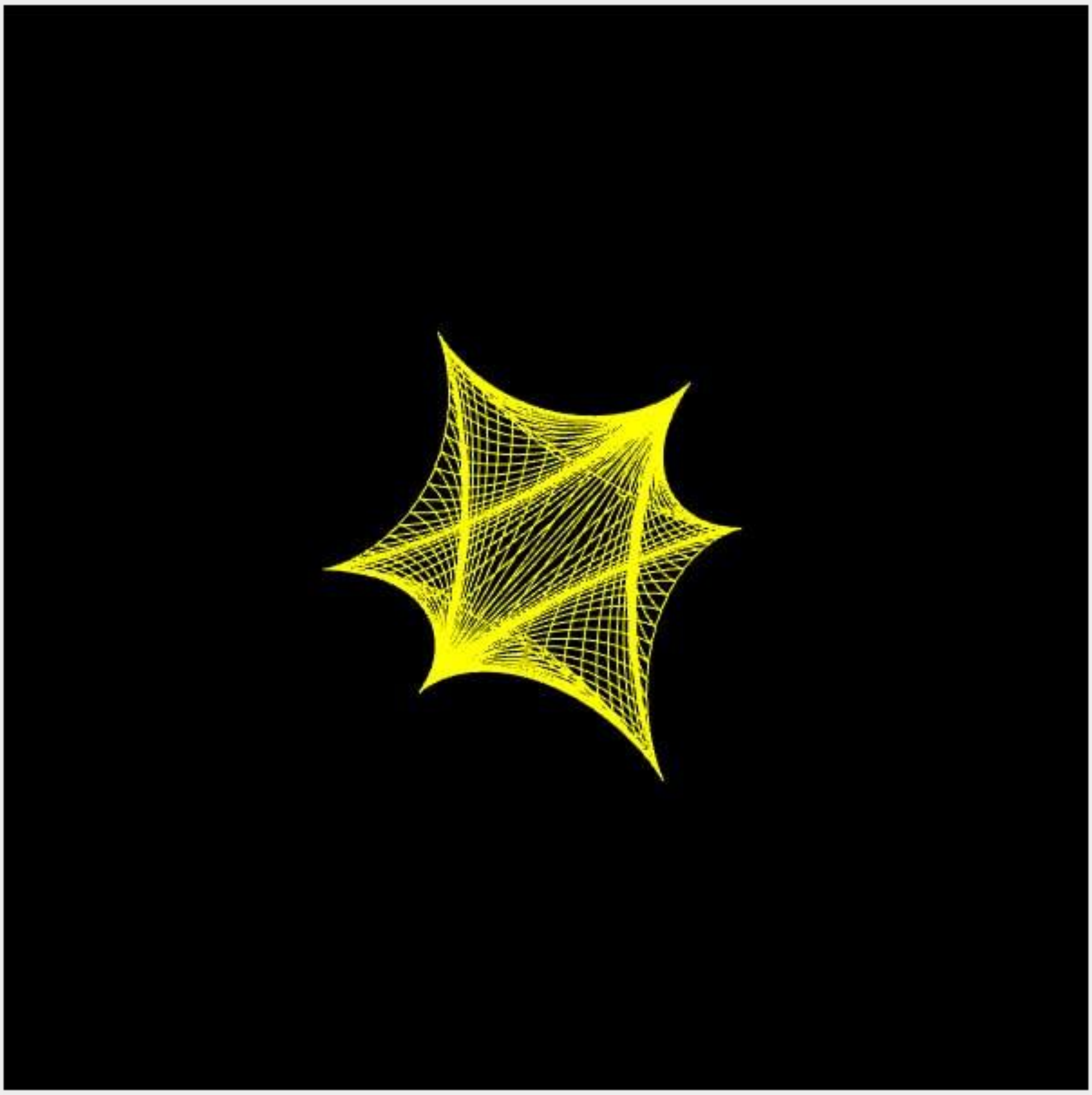
Keys **A** and **D** - move the point along u parameter

Keys **W** and **S** - move the point along v parameter

**Left/right** keyboard keys - move the light

## Astroidal helicoid

Drag your mouse to rotate it.
(On a touch screen, you can use your finger.)



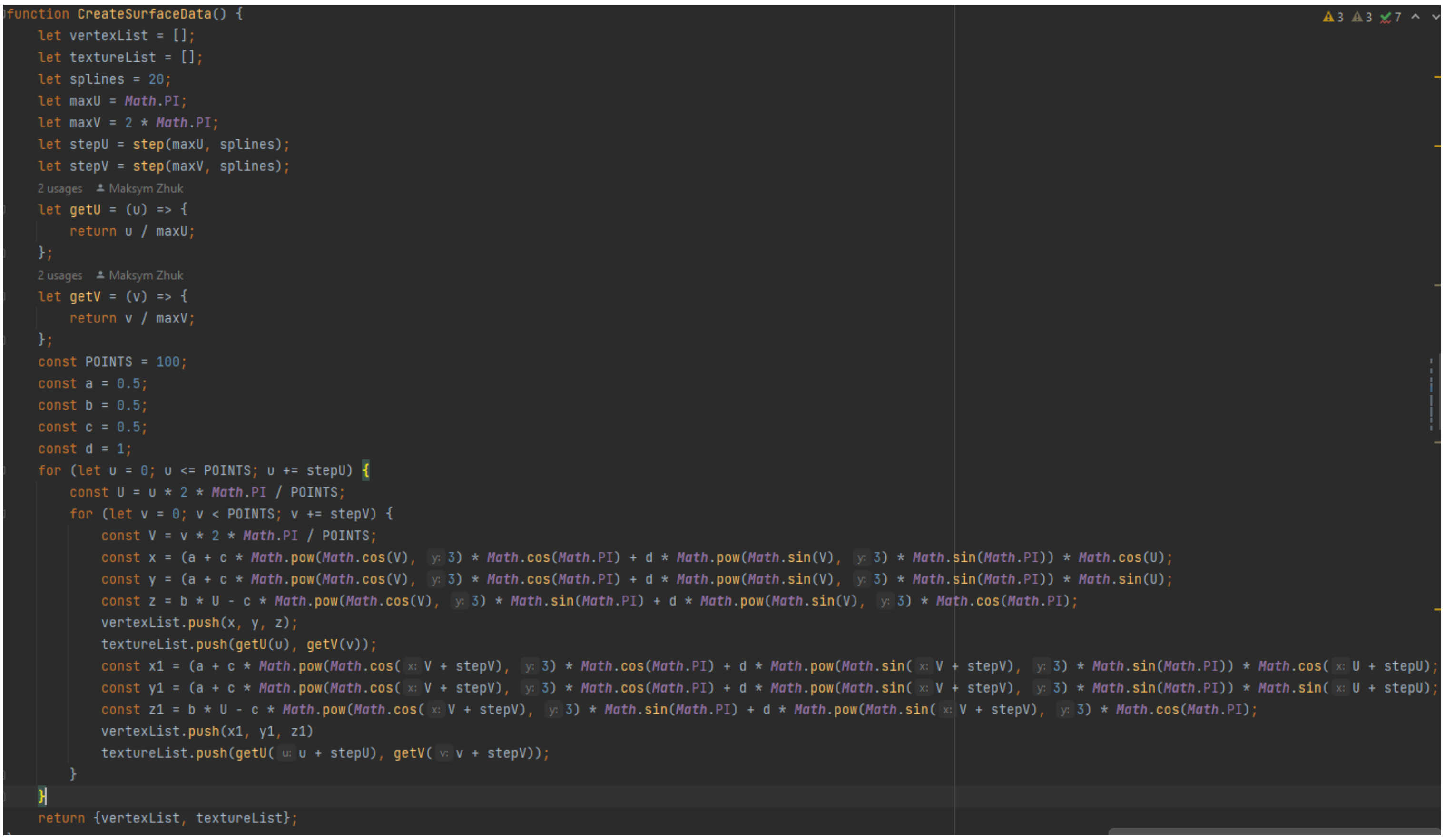
# Source code

```
// Vertex shader
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);
}`;
```

## Fragment shader

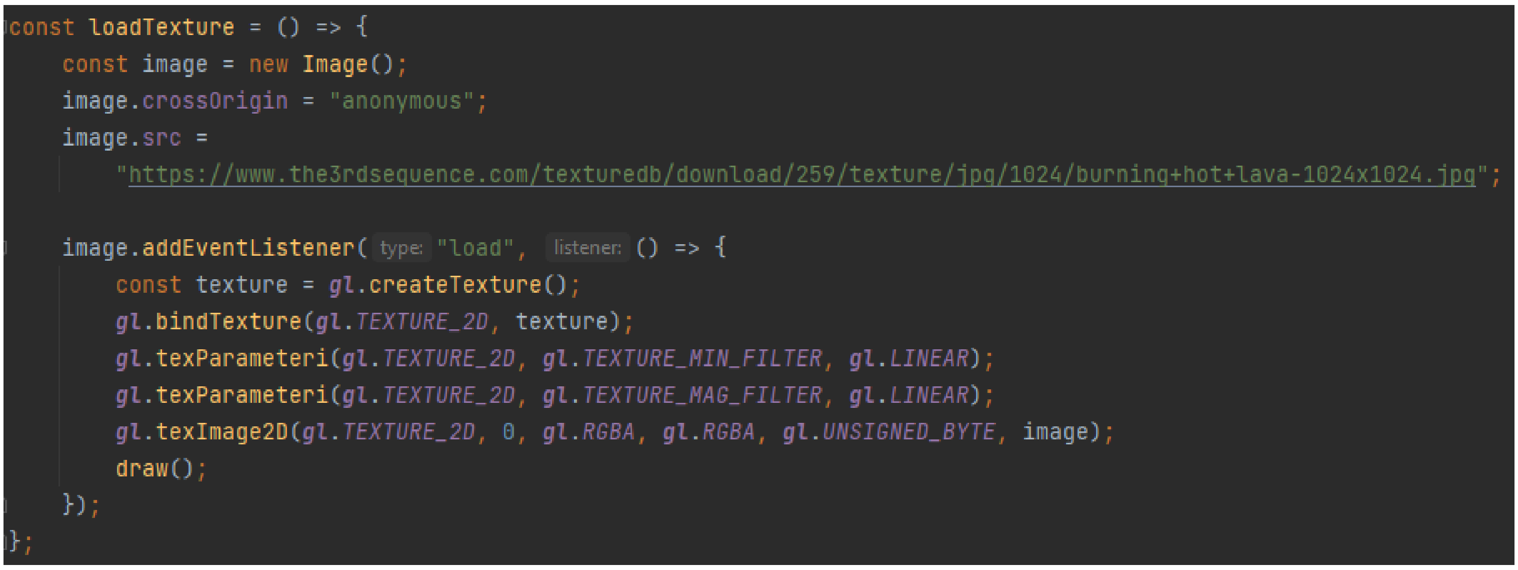
```
const fragmentShaderSource = `
#ifdef GL_FRAGMENT_PRECISION_HIGH
    precision highp float;
#else
    precision mediump float;
#endif

varying vec4 color;
varying vec2 vTextureCoords;
uniform sampler2D textureU;

void main() {
  vec4 texture = texture2D(textureU, vTextureCoords);
  gl_FragColor = texture * color;
```

## Parametric function of the surface



```
function CreateSurfaceData() {
    let vertexList = [];
    let textureList = [];
    let splines = 20;
    let maxU = Math.PI;
    let maxV = 2 * Math.PI;
    let stepU = step(maxU, splines);
    let stepV = step(maxV, splines);
    2 usages  Maksym Zhuk
    let getU = (u) => {
        return u / maxU;
    };
    2 usages  Maksym Zhuk
    let getV = (v) => {
        return v / maxV;
    };
    const POINTS = 100;
    const a = 0.5;
    const b = 0.5;
    const c = 0.5;
    const d = 1;
    for (let u = 0; u <= POINTS; u += stepU) {
        const U = u * 2 * Math.PI / POINTS;
        for (let v = 0; v < POINTS; v += stepV) {
            const V = v * 2 * Math.PI / POINTS;
            const x = (a + c * Math.pow(Math.cos(V),  y: 3) * Math.cos(Math.PI) + d * Math.pow(Math.sin(V),  y: 3) * Math.sin(Math.PI)) * Math.cos(U);
            const y = (a + c * Math.pow(Math.cos(V),  y: 3) * Math.cos(Math.PI) + d * Math.pow(Math.sin(V),  y: 3) * Math.sin(Math.PI)) * Math.sin(U);
            const z = b * U - c * Math.pow(Math.cos(V),  y: 3) * Math.sin(Math.PI) + d * Math.pow(Math.sin(V),  y: 3) * Math.cos(Math.PI);
            vertexList.push(x, y, z);
            textureList.push(getU(u), getV(v));
            const x1 = (a + c * Math.pow(Math.cos( x: V + stepV),  y: 3) * Math.cos(Math.PI) + d * Math.pow(Math.sin( x: V + stepV),  y: 3) * Math.sin(Math.PI)) * Math.cos( x: U + stepU);
            const y1 = (a + c * Math.pow(Math.cos( x: V + stepV),  y: 3) * Math.cos(Math.PI) + d * Math.pow(Math.sin( x: V + stepV),  y: 3) * Math.sin(Math.PI)) * Math.sin( x: U + stepU);
            const z1 = b * U - c * Math.pow(Math.cos( x: V + stepV),  y: 3) * Math.sin(Math.PI) + d * Math.pow(Math.sin( x: V + stepV),  y: 3) * Math.cos(Math.PI);
            vertexList.push(x1, y1, z1)
            textureList.push(getU( u: u + stepU), getV( v: v + stepV));
        }
    }
    return {vertexList, textureList};
```

## Parametric function of the texture



```
const loadTexture = () => {
    const image = new Image();
    image.crossOrigin = "anonymous";
    image.src =
        "https://www.the3rdsequence.com/texturedb/download/259/texture/jpg/1024/burning+hot+lava-1024x1024.jpg";

    image.addEventListener( type: "load",  listener: () => {
        const 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);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
        draw();
    });
};
```