National Technical University of Ukraine

"Kyiv Polytechnic Institute named after Igor Sikorsky"

Educational and Scientific Institute of Nuclear and Thermal Energy

Department of Digital Technologies in Energy

**Visualization of graphical and geometric information**

Calculation and graphics work

**Operations on texture coordinates**

Variant 15

Performed:

1st year master’s student,

the TR-42mp group

Anna Tkachuk

Inspected:

Oleksandr Kaleniuk

Kyiv-2024

**TASK**

Develop a program that performs operations on texture coordinates to enable scaling or rotation of a texture around a user-specified point, based on the variant requirements.

**Requirements**

* Reuse texture mapping from Control task.
* Implement texture *scaling* (texture coordinates) around user specified point.
* Allow the user to move the specified point dynamically on the texture surface in the *(u, v)* space using keyboard controls:
* Keys *A* and *D* to move the point along the *u* parameter.
* Keys *W* and *S* to move the point along the *v* parameter.

**THEORY**

*WebGL (Web Graphics Library)* is a JavaScript API for rendering high-performance interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins. WebGL does so by introducing an API that closely conforms to OpenGL ES 2.0 that can be used in HTML [<canvas>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas) elements. WebGL allows developers to use the GPU for rendering graphics, making it highly performant for complex visualizations and interactive applications.

WebGL operates as a low-level API, meaning that developers must have a good understanding of computer graphics concepts, including shaders, buffers, and coordinate systems. The API relies on GLSL (OpenGL Shading Language) for writing vertex and fragment shaders that process rendering data on the GPU.

WebGL is widely used in various applications, including:

* Interactive 3D visualizations.
* Gaming engines.
* Data-driven visualizations.
* Augmented and virtual reality applications.

In WebGL, textures are images or patterns mapped onto 3D or 2D surfaces to provide detail and realism. The mapping of these textures is controlled using texture coordinates, also known as UV coordinates. These coordinates define how a texture is projected onto a geometry's surface.

The texture coordinate system is normalized, meaning:

* The U axis runs horizontally from 0.0 to 1.0.
* The V axis runs vertically from 0.0 to 1.0.

Texture coordinates are typically specified in the vertex data and passed to the fragment shader to determine how the texture is displayed on each pixel of a geometry.

Operations on texture coordinates allow developers to manipulate the appearance of textures on surfaces. Common operations include scaling, rotating, translating, and distorting the texture.

1. *Scaling.* Scaling modifies the size of the texture on the surface. It involves multiplying the texture coordinates by a scale factor. This operation can zoom in or out of the texture pattern, affecting its resolution on the surface.
2. *Rotation.* Rotation changes the orientation of the texture around a pivot point.
3. *Translation.* Translation moves the texture on the surface by adding an offset to the texture coordinates. This operation is useful for scrolling textures or dynamically positioning them.
4. *Custom manipulations.* Texture coordinates can also be altered to achieve advanced effects, such as wrapping, mirroring, or distortion. These techniques involve conditional modifications or non-linear transformations of the UV coordinates.

Texture coordinate manipulations are typically performed in *shaders*, which run directly on the GPU for high performance. These operations can be implemented in either:

* *Vertex Shaders*: Transform texture coordinates at the vertex level, allowing smoother interpolation across the surface.
* *Fragment Shaders*: Apply transformations per pixel for fine-grained control and dynamic effects.

The adjusted coordinates are passed to the texture2D (or texture in modern GLSL) function, which fetches the corresponding texel (texture pixel) from the texture map. This function uses the transformed UV coordinates to sample the texture and determine the color or data value for rendering.

**IMPLEMENTATION DETAILS**

The application is composed of several interconnected modules developed using WebGL and JavaScript. The main components include:

*HTML interface*: Serves as the rendering canvas for the 3D surface and provides user controls such as sliders and key bindings for interaction.

*JavaScript logic*: Handles the rendering pipeline, surface modeling, texture mapping, and processing of user inputs.

*Shaders*: Custom vertex and fragment shaders that execute core calculations for vertex transformations and texture scaling.

**Texture scaling**

Scaling is achieved by altering the UV coordinates within the vertex shader.

The operation is centered around a user-specified pivot point, which can be dynamically adjusted using keyboard controls (keys A and D to move the point along u parameter and keys W and S move the point along v parameter).

**Interactive user input**

The application supports interaction through keyboard and mouse inputs.

Users can rotate the 3D surface via mouse drag, and reposition the scaling pivot point using designated keys.

**Shader logic**

Vertex shaders compute the transformed UV coordinates, scaling them relative to the pivot point.

Fragment shaders handle texture sampling based on the modified UV coordinates, along with processing lighting, shading, and surface highlighting effects.

The surface is constructed as a conical structure using Cartesian Folium equations. The Model class is responsible for generating the vertices, normals, tangents, bitangents, and UV coordinates. The UV coordinates are normalized within the range of 0 to 1 to ensure accurate texture mapping.

The scaling operation is implemented using translation and scaling matrices. The UV coordinates are first translated to position the pivot point at the origin, scaled, and then translated back to their original location.

To improve user interaction, the pivot point is visually highlighted in yellow when it is close to the UV coordinates of a rendered pixel. The fragment shader calculates the distance between the current UV coordinates and the highlight position, blending the highlight color if the distance falls within a predefined threshold.

The implementation uses three textures—diffuse, specular, and normal. These textures are loaded asynchronously, and the rendering process begins only after all textures are fully loaded.

The application offers interactive features such as sliders to adjust the U and V steps of the surface and the texture scaling factor. Keyboard controls (A, D, W, S) allow users to dynamically move the scaling pivot point.

The rendering process is powered by WebGL and utilizes a perspective projection matrix and a model-view matrix to position the camera and the surface. A rendering loop ensures the surface is continuously updated, incorporating user interactions and lighting calculations.

**USER`S INSTRUCTION**

Upon launch, you will see a central canvas displaying a conical surface. In addition, there are controls including sliders for adjusting equation parameters, surface resolution, and texture scaling, as well as instructions for navigating the scene. Figure 1 shows the program interface with default settings.

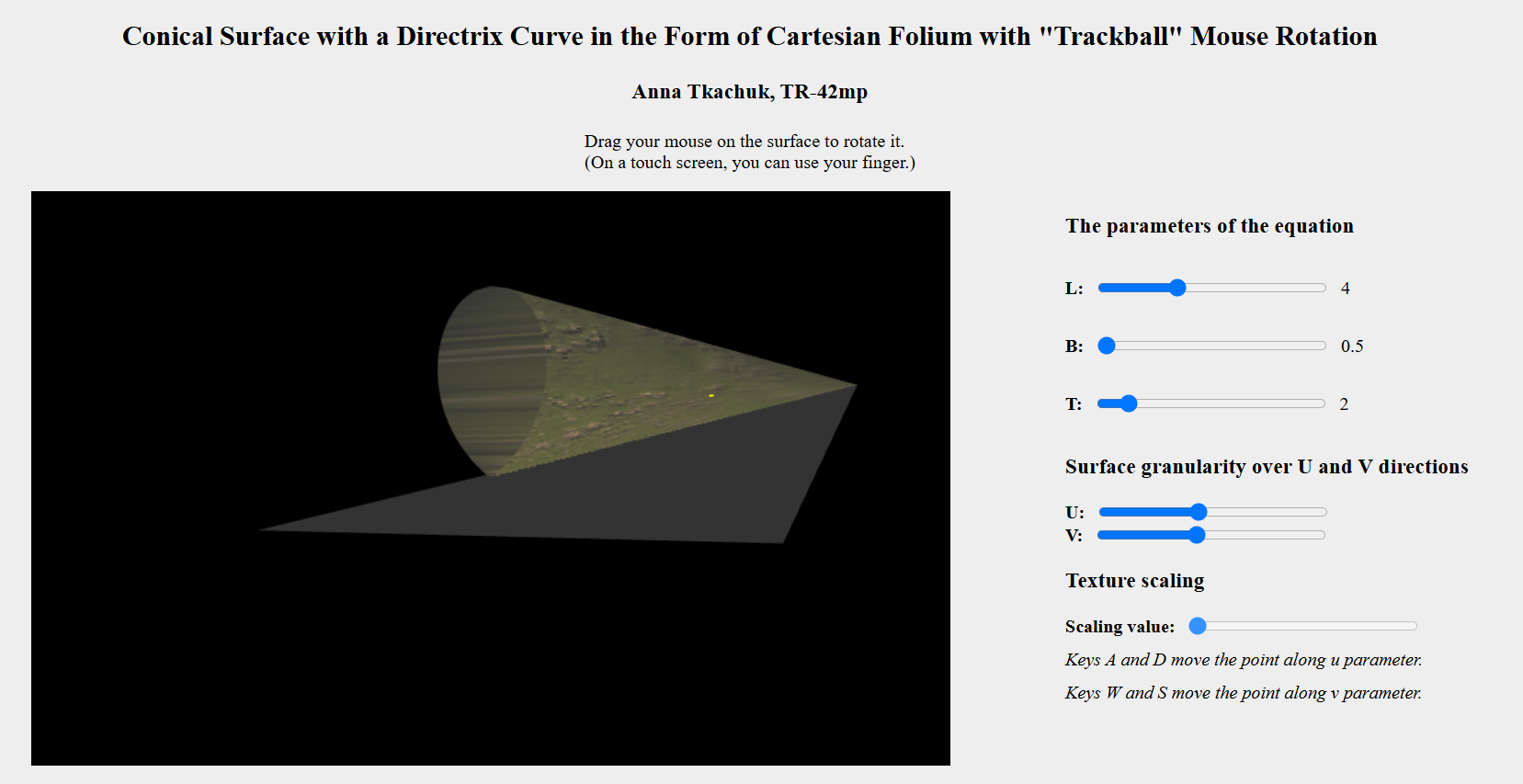


Figure 1. Program interface

The texture scale slider enables users to increase or decrease the size of the texture applied to the surface. The scaling operation is centered around a pivot point that can be repositioned dynamically. Figure 2 demonstrates how altering the texture scale affects the surface’s appearance, highlighting the impact of real-time scaling.

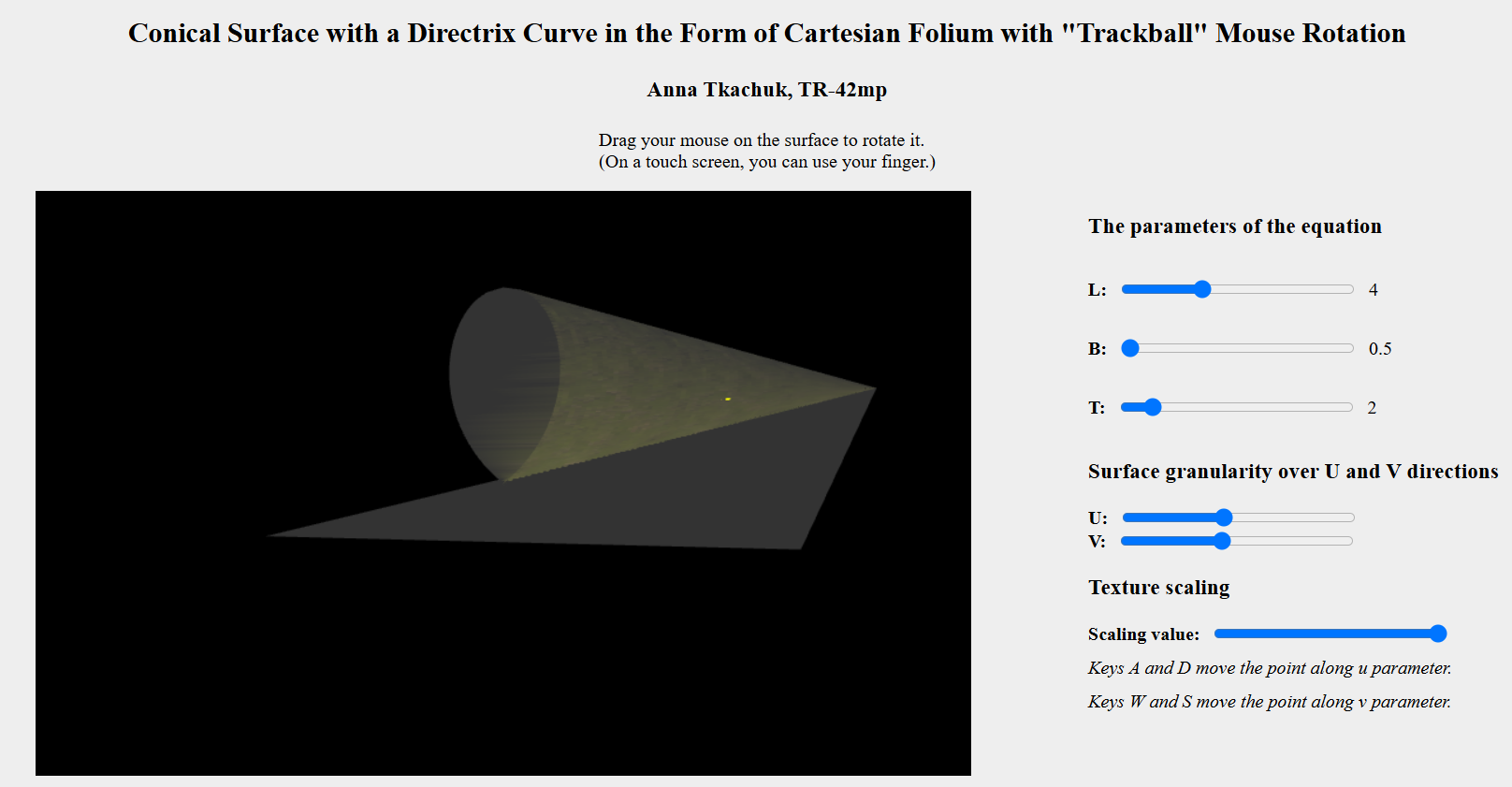


Figure 2. Adjusting Texture Scale

Pressing the A or D keys shifts the pivot along the U axis, while W or S moves it vertically along the V axis. Figure 3 shows figure with the pivot moved.

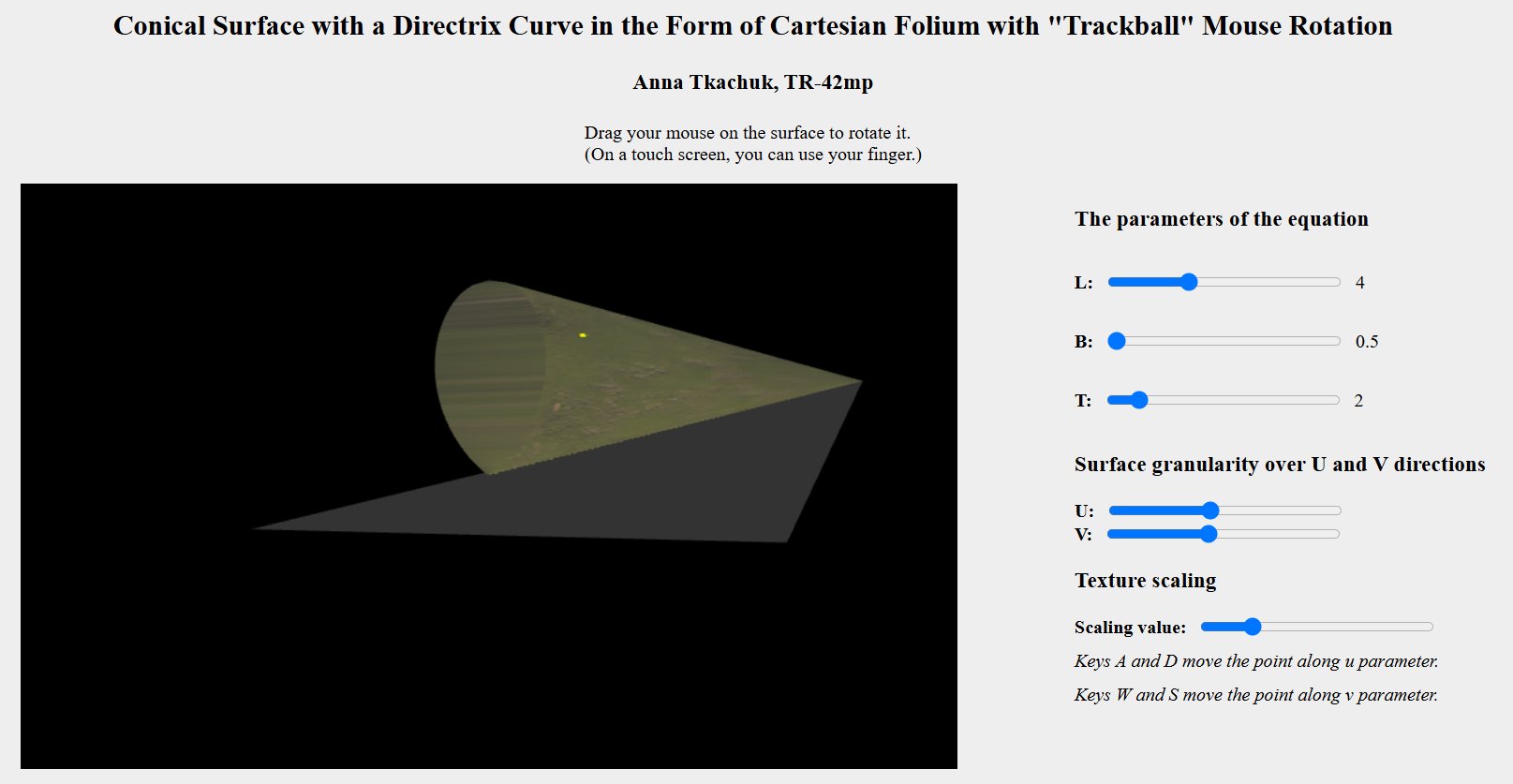


Figure 3. Pivot Point Highlighting

**CODE**

Function responsible for loading a texture:

function loadTexture(url) {

    const texture = gl.createTexture();

    texture.image = new Image();

    texture.image.onload = function() {

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

        gl.texImage2D(

            gl.TEXTURE\_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED\_BYTE,

            texture.image

        );

        gl.generateMipmap(gl.TEXTURE\_2D);

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

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

        texturesLoaded++;

    }

    texture.image.src = url;

    return texture;

}

Handling texture scaling around a pivot point in the vertex shader:

void main(void) {

vec4 posEye = ModelViewMatrix \* vec4(vertex, 1.0);

vPosition = posEye.xyz;

vec3 N = normalize(NormalMatrix \* normal); // Prioritize normal

vec3 T = tangent - dot(tangent, N) \* N;

T = normalize(T);

T = NormalMatrix \* T;

vec3 B = NormalMatrix \* bitangent;

B = normalize(cross(N, T));

vN = N;

vT = T;

vB = B;

vOrigUV = uv;

// Texture scaling

vec2 shifted = uv - texPivot;

shifted \*= texScale;

vTexCoord = shifted + texPivot;

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

}

Highlighting the pivot point in the fragment shader:

void main(void) {

mat3 TBN = mat3(normalize(vT), normalize(vB), normalize(vN));

vec3 normalSample = texture2D(normalTex, vTexCoord).rgb;

normalSample = 2.0 \* normalSample - 1.0;

vec3 N = normalize(TBN \* normalSample);

vec3 diffColor = texture2D(diffuseTex, vTexCoord).rgb;

float specFactor = texture2D(specularTex, vTexCoord).r;

vec3 L = normalize(lightPos - vPosition);

vec3 V = vec3(0.0, 0.0, 1.0);

vec3 R = reflect(-L, N);

float diffuse = max(dot(N, L), 0.0);

float spec = 0.0;

if(diffuse > 0.0) {

spec = pow(max(dot(R, V), 0.0), shininess) \* specFactor;

}

vec3 ambient = Ka;

vec3 diff = Kd \* diffColor \* diffuse;

vec3 specular= Ks \* spec;

vec3 finalRGB = ambient + diff + specular;

vec4 outColor = vec4(finalRGB, 1.0) \* color;

float distUV = distance(vOrigUV, pointUV);

if(distUV < 0.01) {

float pointStrength = 0.9;

vec4 pointColor = vec4(1.0, 1.0, 0.0, 1.0);

outColor = mix(outColor, pointColor, pointStrength);

}

gl\_FragColor = outColor;

}