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:

TR-41mp group

Moroz Oleksandr

Kyiv-2024

**Task Description**

This chapter introduces the problem, objectives, and scope of the task.

**Objective**

The primary goal of this task is to implement dynamic texture mapping transformations on a 3D surface. Users can interactively control the texture’s position, scale, and rotation via keyboard inputs.

**Key Features**

* Transform textures through scaling, rotation, and translation.
* Allow user interaction to adjust the texture's transformation in real-time.
* Render the transformed textures on a dynamically generated 3D surface using WebGL.

**Scope**

This project showcases interactive texture manipulation using WebGL, focusing on user interaction, rendering performance, and visual effects. The implementation is modular to allow future extensions.

2. **Theory**

This chapter explains the fundamental concepts underlying the implementation of texture transformations on a 3D surface.

2.1 Texture Mapping

Texture mapping is a graphical technique where a 2D image, called a texture, is applied to a 3D surface. The goal is to enhance the realism of 3D models by giving them detailed appearances without adding complexity to their geometry. Each vertex of the 3D surface is assigned texture coordinates (u, v) that determine how the texture is mapped onto the surface.

* Texture Coordinates (u, v): These are normalized values between [0, 1] representing positions within the texture image.
  + u corresponds to the horizontal axis (left to right).
  + v corresponds to the vertical axis (bottom to top).

The mapping process involves interpolating these coordinates across the surface's faces (typically triangles) and sampling the texture image to determine the color of each pixel.

2.2 Transformations in Texture Mapping

In computer graphics, transformations such as translation, scaling, and rotation are essential for altering the appearance of textures. These transformations are applied to the texture coordinates before sampling the texture.

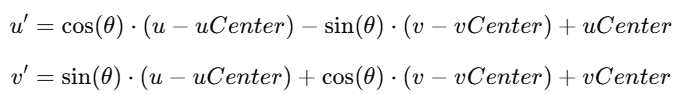
1. Translation  
   Translation shifts the texture across the surface by adding an offset (∆u, ∆v) to the texture coordinates:



1. Scaling  
   Scaling adjusts the size of the texture on the surface. A scaling factor s is applied to both coordinates:



1. Rotation  
   Rotation pivots the texture around a specified point (uCenter, vCenter). A rotation angle θ transforms the coordinates as follows:

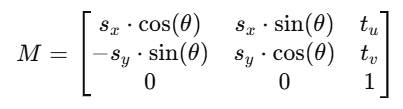


These transformations are combined into a transformation matrix, which simplifies their application and allows efficient computation in shaders.

2.3 Matrix Representation of Transformations

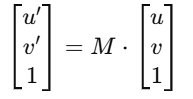
A 3x3 transformation matrix is used to represent texture transformations in 2D space. This matrix enables combined translation, scaling, and rotation. The general form is:

Where:



* sx sy: Scaling factors in the horizontal and vertical directions.
* θ: Rotation angle.
* tu,tv: Translation offsets.

The transformed texture coordinates [u′,v′,1] are obtained by multiplying the original coordinates [u,v,1] with the matrix M:



2.4 Shader Programming

Shaders are small programs executed on the GPU to process graphics data. They are written in GLSL (OpenGL Shading Language). For texture mapping, two main shaders are used:

1. Vertex Shader  
   The vertex shader is responsible for transforming vertex positions and texture coordinates. It applies the transformation matrix to the texture coordinates and passes them to the fragment shader.

Example of texture coordinate transformation in a vertex shader:

vec3 transformedTex = TextureMatrix \* vec3(tex, 1.0);

vTexCoord = transformedTex.xy;

1. Fragment Shader  
   The fragment shader computes the final color of each pixel. It samples the texture using the transformed coordinates.

Example of texture sampling in a fragment shader:

vec4 color = texture2D(textureSampler, vTexCoord);

**3. Implementation Details**

The project implements dynamic texture transformations on a 3D surface using WebGL. It is structured into several components to manage the generation of the 3D surface, texture transformations, rendering, and user interaction. The implementation focuses on applying transformations like scaling, rotation, and translation to texture coordinates in real-time based on user inputs.

The project uses three main files: Model.js for creating the 3D surface, shader.gpu for GPU-based vertex and fragment shader logic, and main.js for WebGL setup, rendering, and user interaction.

The Model.js file generates a parametric 3D surface. The surface is defined by its granularity in the u and v directions, and shape parameters like length (L), thickness (T), and breadth (B). The vertex positions are calculated using a parametric equation. Normals are computed for each vertex to support proper lighting. Texture coordinates are assigned to each vertex in the (u, v) space to map textures onto the surface. All data, including vertex positions, normals, and texture coordinates, is stored in GPU buffers to enable efficient rendering.

The shader.gpu file contains the vertex and fragment shaders written in GLSL. The vertex shader applies transformations to vertex positions and texture coordinates. A transformation matrix is used to scale, rotate, and translate the texture coordinates. The fragment shader samples textures using the transformed coordinates and blends them. This allows multiple textures to be combined on the 3D surface. The shaders are designed to take advantage of the GPU's parallel processing capabilities for real-time performance.

The main.js file initializes WebGL and manages the rendering process. It sets up the WebGL context, loads and compiles the shaders, and links them into a shader program. The shader program manages the uniforms and attributes required for the rendering process, including the transformation matrix, model-view-projection matrix, and texture data. Textures are loaded asynchronously, and their data is transferred to GPU memory. A default placeholder texture is displayed until the image is fully loaded.

User interaction is implemented through keyboard inputs. Keys A and D move the texture horizontally in the u direction, while W and S move it vertically in the v direction. Keys Q and E increase and decrease the texture's scale, and keys Z and C rotate the texture. The transformation matrix is recalculated dynamically in response to these inputs. The matrix accounts for scaling, rotation, and translation around a user-specified center. It is passed to the vertex shader as a uniform in every frame.

The rendering loop clears the canvas and depth buffer at the beginning of each frame. It calculates the model-view-projection matrix for the current camera view and updates the texture transformation matrix. These matrices and other required data are passed to the shaders. The 3D surface is then drawn using gl.drawElements, which renders the geometry defined by the vertex and index buffers.

This implementation effectively combines WebGL and GLSL to create an interactive system for texture mapping. The modular structure makes the system scalable and allows for future extensions, such as adding more complex surface equations or additional texture blending techniques. By leveraging GPU-based processing, the system achieves real-time performance, even with dynamic user interactions and high surface granularity.

**4. User's Instruction with Screenshots**

This chapter provides a step-by-step guide for using the application.

**Step 1: Open the Application**

* Open the index.html file in a browser supporting WebGL.

**Step 2: Control Texture Transformations**

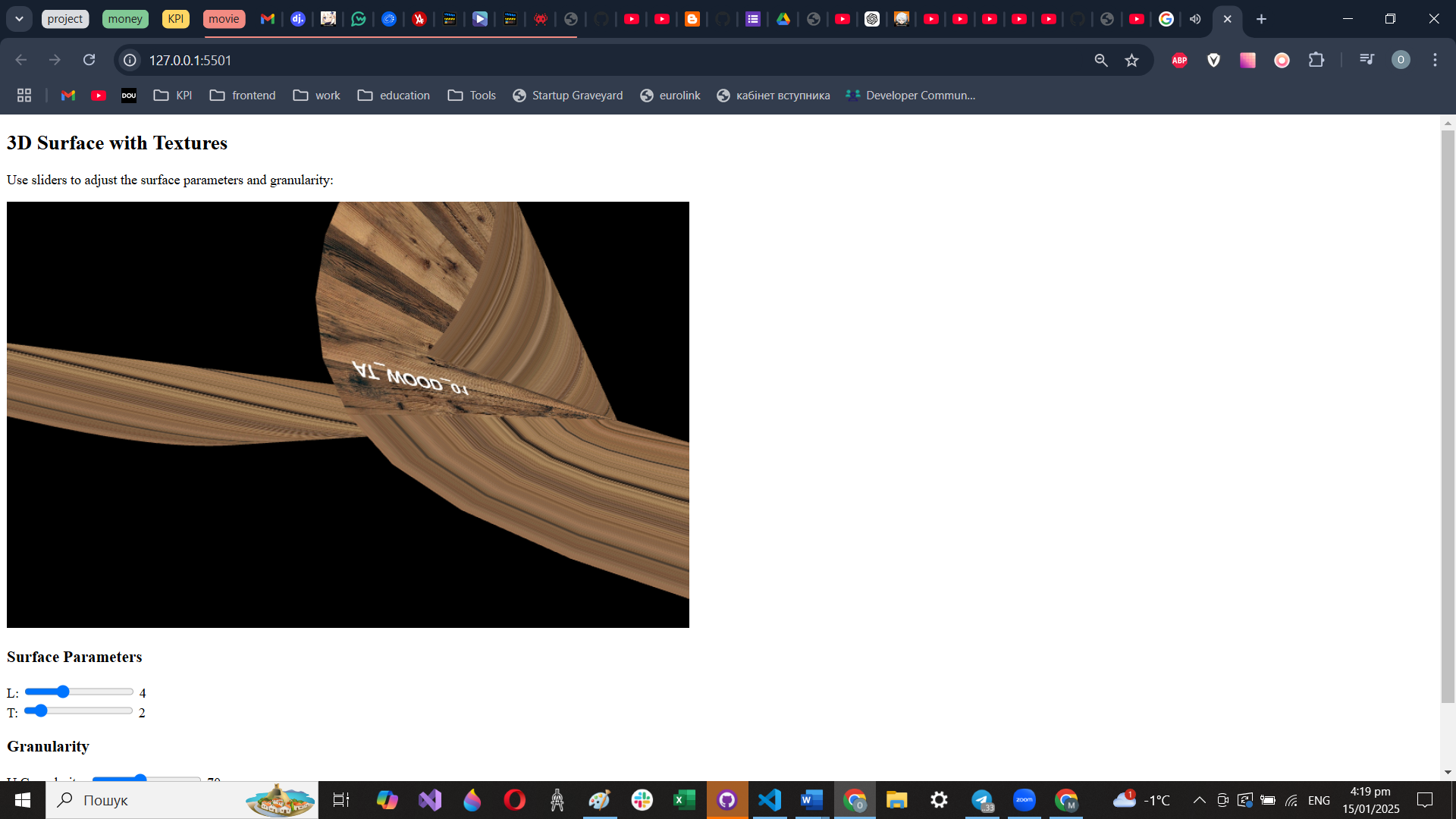
* Use the following keyboard controls to interact with the texture:
  + **Move Texture**:
    - A/D: Move left/right.
    - W/S: Move up/down.
  + **Scale Texture**:
    - Q: Increase scale.
    - E: Decrease scale.
  + **Rotate Texture**:
    - Z: Rotate clockwise.
    - C: Rotate counterclockwise.

**Step 3: Observe Real-Time Updates**

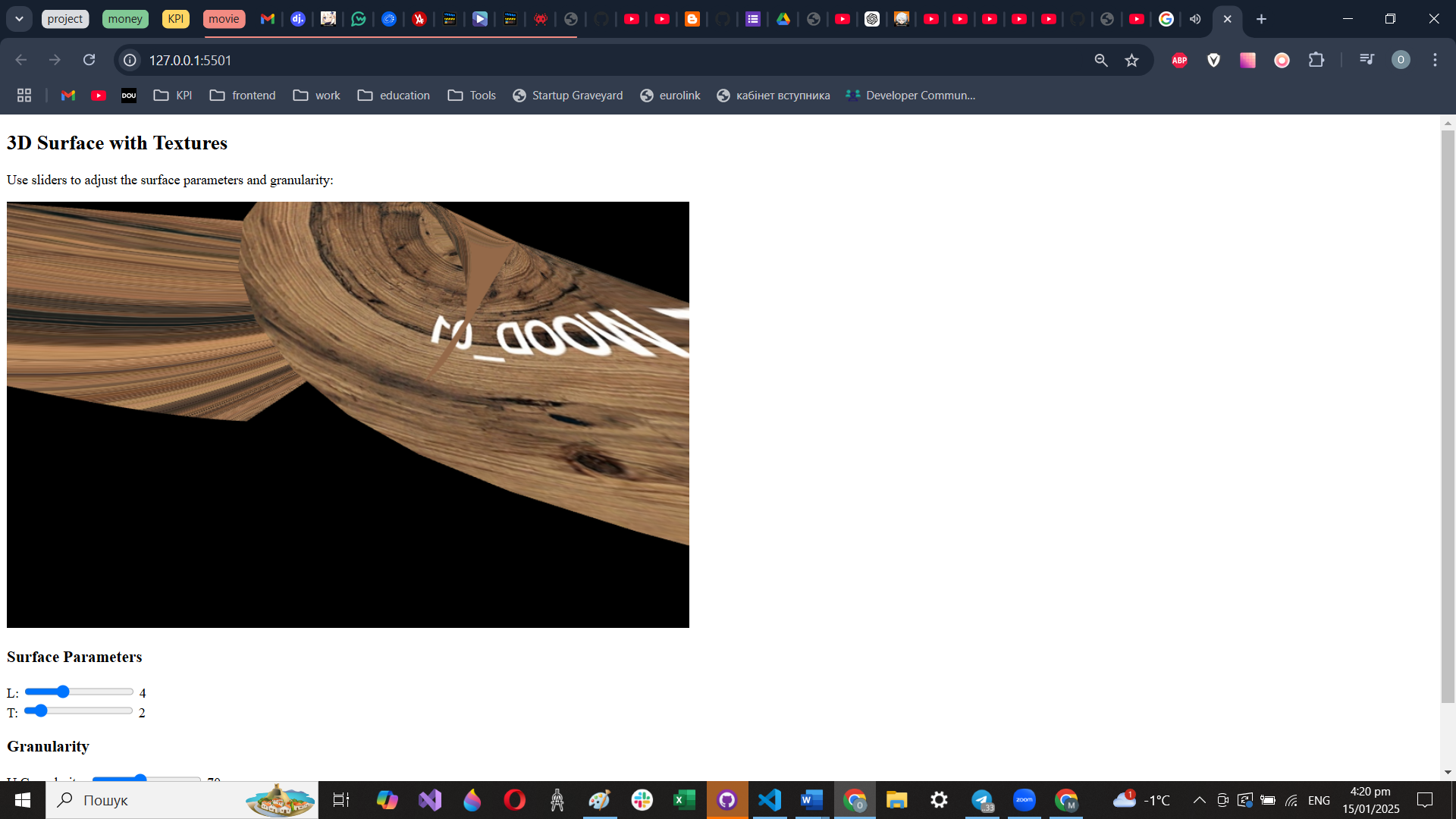
* Observe how the texture on the surface responds to your inputs.

**Screenshots**

1. Initial Surface



1. Transformed Texture



**5. Sample Source Code**

// Vertex Shader

const vertexShaderSource = `

attribute vec3 vertex;

attribute vec2 tex;

uniform mat4 ModelViewProjectionMatrix;

uniform mat3 TextureMatrix;

varying vec2 vTexCoord;

void main() {

// Transform texture coordinates

vec3 transformedTex = TextureMatrix \* vec3(tex, 1.0);

vTexCoord = transformedTex.xy;

// Calculate the vertex position

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

}`;

// Fragment Shader

const fragmentShaderSource = `

#ifdef GL\_FRAGMENT\_PRECISION\_HIGH

precision highp float;

#else

precision mediump float;

#endif

uniform sampler2D iTMU0; // Texture 0

uniform sampler2D iTMU1; // Texture 1

varying vec2 vTexCoord;

void main() {

// Sample both textures and blend them equally

vec4 texColor0 = texture2D(iTMU0, vTexCoord);

vec4 texColor1 = texture2D(iTMU1, vTexCoord);

// Blend the colors from both textures

gl\_FragColor = 0.5 \* texColor0 + 0.5 \* texColor1;

}`;

JavaScript: Texture Transformation

function updateTextureMatrix() {

const cosR = Math.cos(rotation);

const sinR = Math.sin(rotation);

return [

scale \* cosR, scale \* sinR, 0.0,

-scale \* sinR, scale \* cosR, 0.0,

uCenter \* (1 - scale \* cosR) - vCenter \* scale \* sinR,

vCenter \* (1 - scale \* cosR) + uCenter \* scale \* sinR, 1.0

];

}