**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**

Topic: «Operations on texture coordinates»

Variant 1

**Performed:**  
5th year student,

the TR-43mp group

Bahnii A.I.

**Inspected:**

PhD Kaleniuk O. S.

Kyiv-2024

**TASK**

The objective of this project is to enhance texture mapping by implementing the scaling of texture coordinates around a user-specified pivot point on a 3D surface. Building upon the foundational texture mapping functionality developed in a previous control assignment, this implementation introduces dynamic scaling of texture coordinates in the "u" and "v" parameter spaces. The scaling operation is centered around a pivot point that users can interactively reposition using keyboard controls. Specifically, the A and D keys facilitate horizontal movement along the "u" axis, while the W and S keys enable vertical movement along the "v" axis. Additionally, a visual marker is incorporated to indicate the current position of the pivot point, providing immediate visual feedback to users.

**THEORETICAL INFORMATION**

WebGL (Web Graphics Library) is a powerful JavaScript API designed for rendering interactive 2D and 3D graphics within web browsers without the need for external plugins. It is built upon the OpenGL ES standard, which is tailored for embedded systems, thereby ensuring efficient performance across a wide range of devices. WebGL leverages the capabilities of the GPU (Graphics Processing Unit) to deliver high-performance graphics, making it integral to modern web applications, including games, simulations, and data visualizations.

At the core of WebGL’s functionality are shaders, which are programmable components written in GLSL (OpenGL Shading Language). Shaders allow developers to define how vertices and pixels are processed and rendered, providing fine-grained control over the graphics pipeline. There are two primary types of shaders in WebGL: vertex shaders and fragment shaders. Vertex shaders handle the processing of individual vertices, transforming their positions from object space to clip space and passing essential data such as normals, texture coordinates, and other vertex attributes to fragment shaders. Fragment shaders, on the other hand, determine the color and other attributes of each pixel, facilitating the application of textures, lighting, and shading effects.

Texture mapping is a fundamental technique in computer graphics that involves applying detailed 2D images (textures) onto 3D surfaces. By associating texture coordinates (u, v) with each vertex of a 3D model, textures can be accurately projected, adding realism and complexity without increasing geometric detail. The precision of texture mapping depends on several factors, including the resolution and format of the texture images, the accuracy of the UV coordinates assigned to each vertex, and the algorithms used to interpolate these coordinates across the surface.

The process of scaling texture coordinates involves modifying the (u, v) values to enlarge or shrink the texture on the surface. When scaling is centered around a specific pivot point, it ensures that the transformation maintains visual coherence relative to that point. Mathematically, this process entails a series of transformations: translating the pivot point to the origin, applying the scaling transformation, and then translating the pivot point back to its original position. This sequence ensures that scaling occurs uniformly around the desired pivot point, preserving the texture's alignment and preventing distortion.

Understanding UV mapping is crucial for effective texture scaling. UV mapping is the process of projecting a 2D texture onto a 3D model by mapping each vertex of the model to corresponding points on the texture. The "u" and "v" coordinates represent the horizontal and vertical axes of the texture, respectively. Proper UV mapping ensures that textures wrap around 3D objects without stretching or compressing unnaturally. Scaling the texture coordinates alters how the texture is displayed on the surface, allowing for effects such as zooming in on specific areas or repeating patterns.

In the context of this project, implementing texture scaling around a user-specified pivot point enhances interactivity and control over how textures are displayed on 3D surfaces. This capability is particularly useful in applications where users need to focus on specific areas of a textured model, such as in detailed inspections, gaming environments, or artistic renderings. By providing keyboard controls to adjust the pivot point, users can dynamically manipulate the texture scaling, observing real-time changes that affect the visual representation of the 3D surface.

Moreover, the introduction of a visual marker for the pivot point serves as an essential tool for user orientation. It provides immediate feedback on the current center of scaling, allowing users to intuitively understand how their inputs affect the texture mapping. This visual aid enhances the user experience by making the interaction more transparent and predictable.

Advanced texture scaling techniques may also involve considerations such as anisotropic filtering, mipmapping, and normal mapping. Anisotropic filtering improves the quality of textures on surfaces viewed at oblique angles, mipmapping enhances performance and visual quality by using precomputed texture levels, and normal mapping adds detailed surface information without increasing geometric complexity.

**IMPLEMENTATION DETAILS**

The implementation of texture scaling around a user-specified pivot point in WebGL was carried out through a series of targeted enhancements in both the shader programs and the JavaScript logic. These modifications were essential to enable dynamic interactivity, allowing users to adjust the texture scaling centered on a movable pivot point while providing immediate visual feedback.

Initially, three pivotal variables were introduced in the JavaScript code: pivotU, pivotV, and texScale. These variables represent the normalized texture coordinates of the pivot point along the "u" and "v" axes, and the scaling factor for the texture, respectively. Setting pivotU and pivotV to 0.5 positioned the pivot point at the center of the texture, and a texScale of 1.0 ensured that the texture scaling began without any initial scaling applied.

To facilitate communication between the JavaScript code and the shaders, uniform locations for the scaling factor and pivot point were obtained. This step was crucial as it allowed the JavaScript to dynamically update the shader variables during runtime, enabling real-time adjustments based on user input. By retrieving these uniform locations, the application could effectively pass the current values of texScale, pivotU, and pivotV to the vertex shader, ensuring that texture scaling and pivot adjustments were accurately reflected in the rendered scene.

A fundamental function, getSurfacePoint, was implemented to translate the normalized texture coordinates into precise 3D coordinates on the conical surface. This function played a critical role in accurately positioning the pivot marker within the 3D space. By converting the texture coordinates (pivotU, pivotV) into spatial coordinates, the application could determine the exact location on the surface where the pivot marker should be rendered, ensuring that it accurately represented the center of the texture scaling transformation.

User interaction was integrated through the enhancement of the initKeyboard function. Event listeners were added to respond to specific key presses, enabling users to move the pivot point horizontally and vertically along the "u" and "v" axes using the **A**, **D**, **W**, and **S** keys. Additionally, the **Q** and **E** keys were mapped to decrease and increase the texture scaling factor, respectively. These interactions allowed users to dynamically adjust both the position of the pivot point and the extent of texture scaling in real time. The function ensured that the updated values remained within predefined bounds to prevent excessive distortion or misalignment of the texture, thereby maintaining visual integrity.

To provide clear and immediate visual feedback, a red pivot marker was introduced into the scene. This marker served as a tangible indicator of the current pivot point's location, enhancing user awareness and control over the texture scaling process. The implementation involved creating a separate shader program specifically for rendering the pivot marker. By utilizing distinct shader programs, the application ensured that the marker remained prominently visible against the textured surface, maintaining its red coloration for easy identification.

The initialization of the pivot shader program involved compiling and linking the pivot vertex and fragment shaders, and obtaining the necessary attribute and uniform locations. A small sphere was created to represent the pivot marker, and its geometry was defined to ensure it rendered accurately within the scene. During each rendering cycle, the application calculated the pivot marker's 3D position based on the current pivotU and pivotV values. This position was then transformed appropriately using translation matrices to align the marker with the calculated surface point. The pivot marker was subsequently drawn using the dedicated shader program, ensuring it accurately reflected the pivot point's current location.

Within the vertex shader, the texture coordinates were adjusted to facilitate scaling around the pivot point. This adjustment involved shifting the original texture coordinates relative to the pivot point, applying the scaling factor, and then shifting them back to their original position. This sequence of transformations ensured that the scaling occurred uniformly around the pivot point, preserving the texture's alignment and preventing any distortion. By manipulating the texture coordinates in this manner, the shader effectively controlled how the texture was mapped onto the 3D surface, enabling dynamic and user-controlled scaling effects.

**USER`S INSTRUCTION**

The application provides an interactive environment to explore texture scaling on a 3D surface. Upon launching, you will see the central canvas where the conical surface is rendered. Alongside, there are control elements including sliders to adjust parameters like surface resolution and texture scaling, and instructions for navigating the scene.

To begin, open the application in a browser that supports WebGL. The rendered surface will appear in the canvas area, surrounded by control elements. You can manipulate the resolution of the surface using the uStepsand vStepssliders, which adjust the detail level of the rendered geometry. Moving these sliders alters the smoothness of the surface in real time, providing immediate feedback. There is also a slider that controls the radius of the light sphere. In addition, you can use sliders to control the position of the light. Figure 1 demonstrates the application interface with default settings.

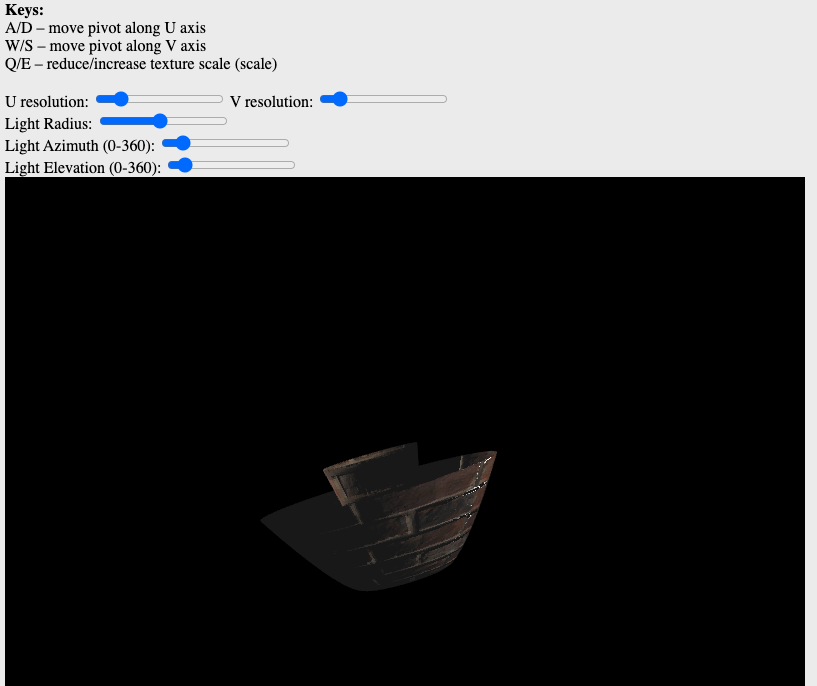


Figure 1. Application interface

Adjusting the texture scaling is straightforward. The texture scale slider allows you to increase or decrease the size of the texture applied to the surface. Scaling is centered around a pivot point, which can be dynamically repositioned. Figure 2 illustrates how modifying the texture scale changes the appearance of the surface, showcasing the effect of real-time scaling.

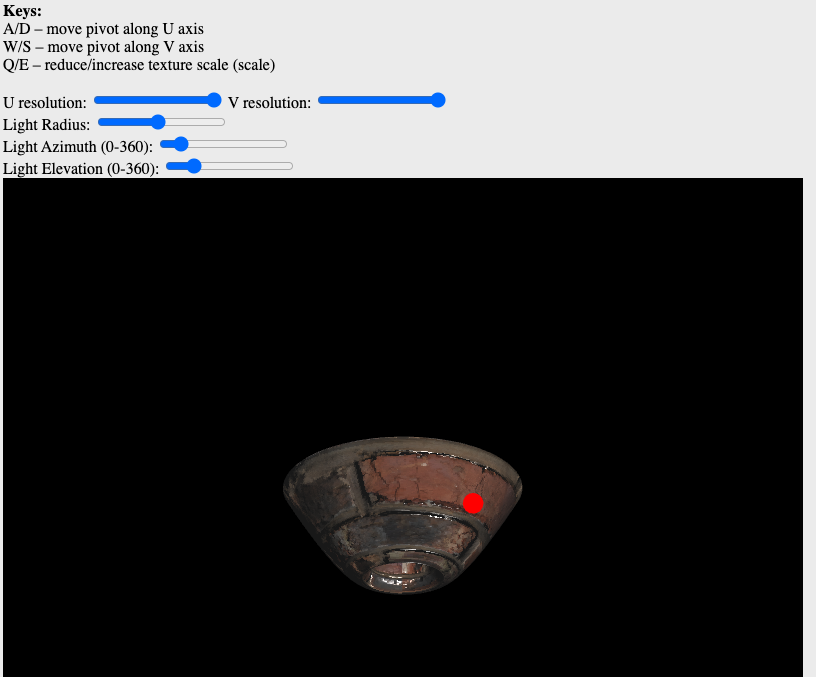


Figure 2. Adjusting texture scale

Repositioning the pivot point is achieved using the keyboard. Pressing the A or D keys shifts the pivot along the horizontal (U) axis, while W or S moves it vertically along the V axis. Figure 3 shows figure with the pivot moved to the right using A key. This interactive feature allows precise control over the scaling transformation and makes it easy to experiment with various effects.

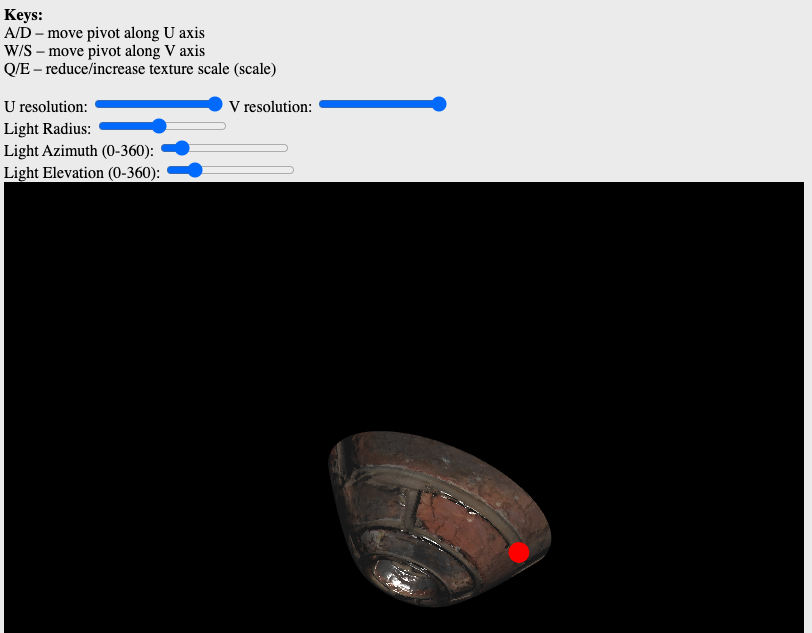


Figure 3. Pivot point highlighting

Through this interface, you can fully explore the dynamic behavior of texture scaling and gain a deeper understanding of how UV coordinates interact with the surface geometry.

**SOURCE CODE**

The getSurfacePoint function converts normalized texture coordinates (uParam, vParam) into 3D spatial coordinates on the conical surface. This ensures accurate placement of the pivot marker based on the current pivot settings.

function getSurfacePoint(uParam, vParam) {  
 let U = uParam \* 2.0 \* *Math*.PI;  
 let tMin = -2.0;  
 let tMax = 2.0;  
 let T = tMin + vParam \* (tMax - tMin);  
  
 let a = 2.0;  
 let c = 1.5;  
 let theta = *Math*.PI / 8;  
 let scaleVal = 0.2;  
  
 let x = scaleVal \* (a + T\**Math*.cos(theta) + c\*T\*T\**Math*.sin(theta)) \* *Math*.cos(U);  
 let y = scaleVal \* (a + T\**Math*.cos(theta) + c\*T\*T\**Math*.sin(theta)) \* *Math*.sin(U);  
 let z = scaleVal \* (-T\**Math*.sin(theta) + c\*T\*T\**Math*.cos(theta));  
 return [x,y,z];  
}

These lines pass the current texture scaling factor and pivot coordinates to the vertex shader. This dynamic updating allows the shader to adjust the texture mapping in real time based on user input.

gl.uniform1f(shProgram.iTexScale, texScale);  
gl.uniform2f(shProgram.iTexPivot, pivotU, pivotV);

This segment activates the pivot marker's shader program, sets the necessary transformation matrices, calculates the pivot's 3D position, and renders the pivot marker sphere at the calculated location, providing visual feedback to the user.

pivotProgram.Use();  
  
gl.uniformMatrix4fv(pivotProgram.iProjectionMatrix, false, projection);  
gl.uniformMatrix4fv(pivotProgram.iViewMatrix, false, viewMatrix);  
  
let pivotXYZ = getSurfacePoint(pivotU, pivotV);  
let pivotMatrix = m4.translation(pivotXYZ[0], pivotXYZ[1], pivotXYZ[2]);  
gl.uniformMatrix4fv(pivotProgram.iModelMatrix, false, pivotMatrix);  
  
pivotSphere.Draw();

The initPivotProgram function sets up the shader program dedicated to rendering the pivot marker. It compiles the pivot vertex and fragment shaders, retrieves attribute and uniform locations, and initializes a small sphere to visually represent the pivot point on the surface.

function initPivotProgram() {  
 let prog = createProgram(gl, pivotVertexShaderSource, pivotFragmentShaderSource);  
 pivotProgram = new ShaderProgram('Pivot', prog);  
 pivotProgram.Use();  
  
 pivotProgram.iAttribVertex = gl.getAttribLocation(prog, "aVertexPosition");  
  
 pivotProgram.iModelMatrix = gl.getUniformLocation(prog, "uModelMatrix");  
 pivotProgram.iViewMatrix = gl.getUniformLocation(prog, "uViewMatrix");  
 pivotProgram.iProjectionMatrix = gl.getUniformLocation(prog, "uProjectionMatrix");  
  
 pivotSphere = new LightModel(gl, pivotProgram);  
 pivotSphere.CreateSphereData(0.1, 16, 16);  
}

The initKeyboard function adds event listeners for key presses, enabling users to move the pivot point along the "u" and "v" axes and adjust the texture scaling factor using specific keys. It ensures that the values remain within defined limits to maintain visual consistency and prevent excessive distortion.

function initKeyboard() {  
 *document*.addEventListener('keydown', (e) => {  
 switch(e.code) {  
 case 'KeyA':  
 pivotU -= 0.01;  
 break;  
 case 'KeyD':  
 pivotU += 0.01;  
 break;  
 case 'KeyW':  
 pivotV += 0.01;  
 break;  
 case 'KeyS':  
 pivotV -= 0.01;  
 break;  
 case 'KeyQ':  
 texScale -= 0.05;  
 break;  
 case 'KeyE':  
 texScale += 0.05;  
 break;  
 }  
 if (pivotU < 0) pivotU = 0; if (pivotU>1) pivotU = 1;  
 if (pivotV < 0) pivotV = 0; if (pivotV>1) pivotV = 1;  
 if (texScale < 0.1) texScale = 0.1;  
 if (texScale > 5.0) texScale = 5.0;  
  
 draw();  
 });  
}

These shader sources define the rendering of the pivot marker. The vertex shader transforms the marker's vertices based on model, view, and projection matrices, while the fragment shader assigns a solid red color to ensure the marker is clearly visible against the textured surface.

var pivotVertexShaderSource =`  
attribute vec3 aVertexPosition;  
  
uniform mat4 uModelMatrix;  
uniform mat4 uViewMatrix;  
uniform mat4 uProjectionMatrix;  
  
void main(void) {  
 gl\_Position = uProjectionMatrix \* uViewMatrix \* uModelMatrix \* vec4(aVertexPosition,1.0);  
}  
`;  
  
var pivotFragmentShaderSource =  
`#ifdef GL\_FRAGMENT\_PRECISION\_HIGH  
 precision highp float;  
#else  
 precision mediump float;  
#endif  
  
void main(void) {  
 gl\_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}  
`;