Ministry of Education and Science of Ukraine

National Technical University of Ukraine

"Kyiv Polytechnic Institute named after Igor Sikorsky"

Educational and Scientific Institute of Atomic and Thermal Energy

Department of Digital Technologies in Energy

**Graphics work**

**Topic:** «Operations on texture coordinates»

from the discipline: «Visualization of graphical and geometric information»

Variant 20

**Performed by:**

student of the TP-43mp group

Yatsenko M.I.

**Checked by:**

Kaleniuk O. C.

Kyiv – 2024

**1. Task**

* Reuse texture mapping from Control task.
* 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.

**2. Theoretical information**

**2.1. Working with Multiple Buffers in WebGL**

In WebGL applications, using multiple buffers is an efficient way to manage different types of data, such as vertices, normals, tangents, UV coordinates, and indices. Each buffer focuses on a specific aspect of the object's rendering.

Benefits of Using Multiple Buffers:

1. Improved Organization: Storing data in separate buffers simplifies management and allows for targeted updates. For instance, UV coordinates can be modified independently without affecting other data.

2. Enhanced Performance: WebGL's `gl.bufferSubData` enables efficient updates by modifying only the data that has changed, instead of reloading entire buffers.

3. Greater Flexibility: Using multiple buffers allows objects to be rendered with different datasets, like applying alternative textures or normals, without requiring new models.

Buffers are initialized and loaded with the required data, ensuring compatibility with the shader program. The shader interprets buffer data through specific functions to ensure seamless rendering.

**2.2. Rendering Flat Shading**

Flat shading is a technique where each triangle in a model is rendered with a uniform color, emphasizing the geometry of the object and creating a stylized, angular appearance. This shading approach is frequently used for low-polygon models or specific artistic styles.

Key Features of Flat Shading:

1. Each triangle uses a single normal vector, calculated for the entire surface.

2. Lighting intensity is computed once per triangle, contrasting with smoother techniques like Gouraud or Phong shading, which calculate lighting at each pixel.

Flat shading simplifies rendering by avoiding interpolation between vertices, producing sharp edges between triangles. It reduces computational demands while highlighting the object's geometric structure.

**2.3. Rotating UV Coordinates**

Rotating UV coordinates modifies the orientation of a texture applied to an object's surface, enabling dynamic transformations and visual effects. This is achieved by applying a rotation matrix to the UV coordinates.

How the Process Works:

1. A rotation matrix is generated based on the specified angle of rotation.

2. The matrix transforms each UV coordinate, rotating the texture on the surface of the object.

3. The modified coordinates are updated in the relevant buffer.

This approach allows for real-time texture adjustments, such as rotating patterns on an object's surface, without altering the texture's actual content. It is particularly useful for creating dynamic effects and refining the texture's alignment.

### **3. Implementation Details: Texture Rotation Around a User-Specified Point**

This functionality enables dynamic rotation of textures around a pivot point specified by the user, enhancing user interaction and providing precise control over texture adjustments within a WebGL application.

#### **Key Components and Logic**

1. **User-Defined Pivot Point**:
   * The pivot point for the rotation is set through the point property in the Model object:
   * this.point = [0.5, 0.5];
2. **Rotation Angle**:
   * The rotation angle is controlled via an HTML input field where users enter the angle in degrees:
   * gl.uniform1f(shProgram.iAngle, parseFloat(document.getElementById('Angle').value) \* (Math.PI / 180.0));
3. **Shader Integration**:
   * The vertex and fragment shaders implement the rotation transformation for the UV coordinates.
   * A 2D rotation matrix is utilized to compute the rotation.
   * The UV coordinates are adjusted by subtracting the pivot point before rotation and adding it back afterward to ensure rotation occurs around the defined pivot.
4. **Updating UV Coordinates**:
   * In the vertex shader, the UV coordinates of the texture are updated using the rotation matrix.
   * This dynamic adjustment ensures the rotated texture is displayed correctly on the rendered surface.
   * The rotation transformation is handled entirely within the shader, leaving the original UV data unchanged.
5. **Drawing the Model**:
   * The Draw method binds the texture and sends the rotation parameters (iPoint and iAngle) to the shader before rendering:
   * gl.uniform2fv(shProgram.iPoint, this.point);
   * gl.uniform1f(shProgram.iAngle, rotationAngle);
   * gl.drawElements(gl.TRIANGLES, this.count, gl.UNSIGNED\_INT, 0);

#### **UI Integration:**

* HTML elements provide an interface for user interaction:
  + Input fields for specifying the rotation angle (#Angle).
  + Event listeners to capture user input and trigger a redraw of the scene when the angle is updated.

#### **Supporting Functions:**

* normalizeUV: Ensures the UV coordinates are normalized within the [0, 1] range.
* LoadTexture: Handles texture loading and configuration (e.g., wrapping, filtering).

**4. User's instruction**

### **4.1. Application Interface Overview**

Once the application is loaded, you will see a 3D surface rendered on the canvas with interactive controls. Below the canvas, there are input fields and sliders to modify settings (fig. 1) :

* U Min, U Max: Input field for setting the limitations for U (usual -Pi/2;Pi/2).
* V Min, V Max: Input field for setting the limitations for V (usual -Pi/2;Pi/2).
* U steps, V steps: Input field to set the number of steps for U and V.
* Angle: Input field to set the rotation angle for the texture.

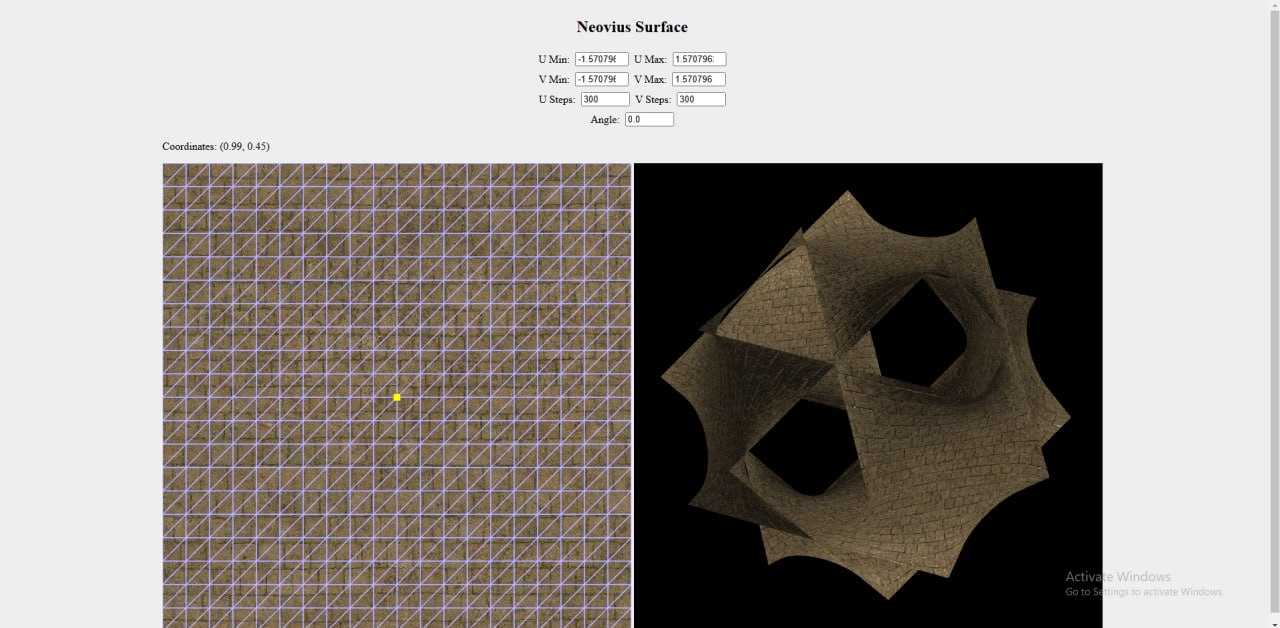
****

Figure 1 - Full application interface

### **4.2. Texture and UV Mapping Area**

Below the input controls panel, spanning the left side of the interface.

**UV Mapping View** shows the 2D texture mapped to the 3D surface. It includes a highlighted yellow point indicating the selected UV coordinates. Displays the real-time coordinates of the selected UV (Fig. 2).



Figure 2 - **UV Mapping View**

### **4.3. 3D Surface Visualization**

We have an **3D Surface o**n the right side of the interface. It is a rendered surface with the applied texture. It has real-time updates based on user inputs from the controls and from the keyboard (Fig. 3).

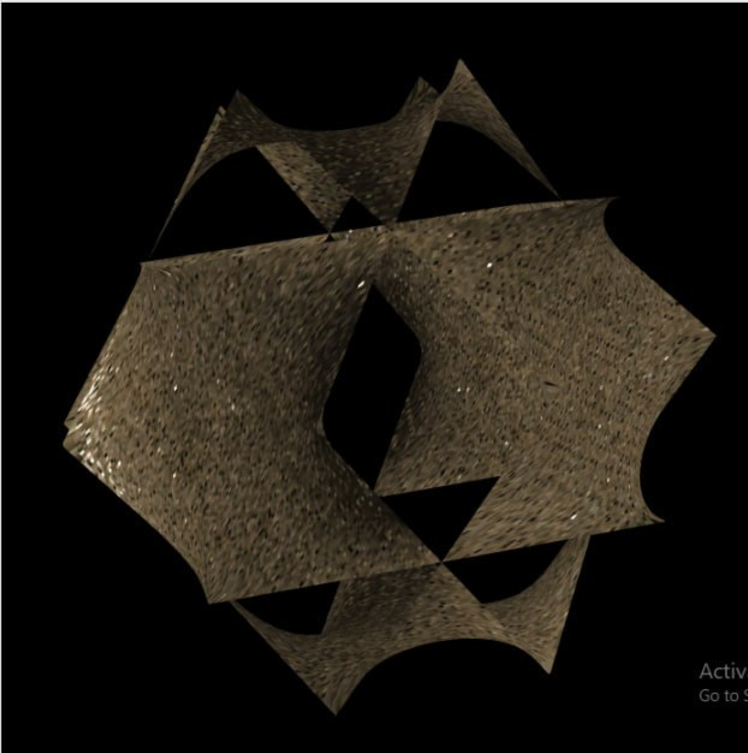


Figure 3 - **3D Surface Visualization**

**4.4 Interactivity with Texture and 3D Model**

We can click anywhere on the green UV mapping area (on the left side of the interface) to select a point. After the selected point will be highlighted with a yellow square. The Coordinates Display will update to show the real-time UV coordinates of the selected point. Changing the value of the angle rotates the texture dynamically on the 3D surface.

This rotation lets us see how the texture aligns with the geometry of the surface (fig. 4).

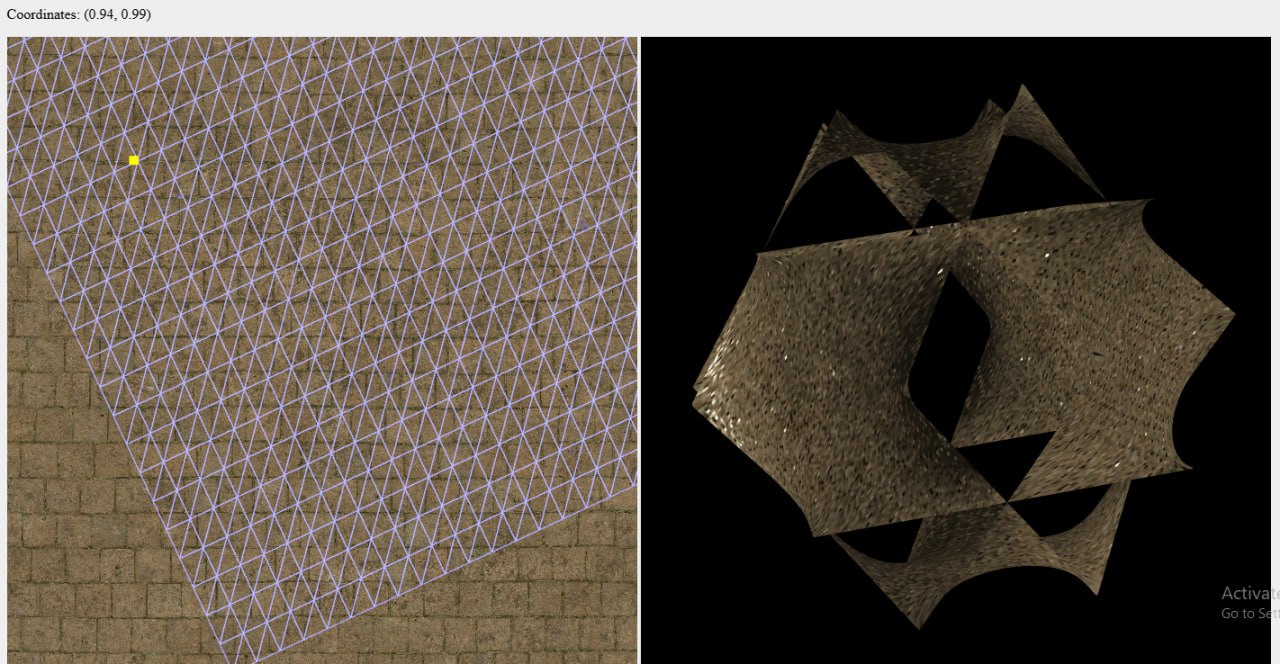
****

Figure 4 - Demonstration of moving a point relative to a figure

**5. Source code**

function derivative\_z\_u(u, v) {

const numerator = 9 \* Math.asin((3 \* Math.cos(u) \* Math.cos(v)) / (3 + 4 \* Math.cos(u) \* Math.cos(v))) \* Math.sin(u) \* Math.cos(v);

const denominator = Math.pow((3 + 4 \* Math.cos(u) \* Math.cos(v)), 2);

return numerator / denominator;

}

function derivative\_z\_v(u, v) {

const numerator = 9 \* Math.acos((3 \* Math.cos(u) \* Math.cos(v)) / (3 + 4 \* Math.cos(u) \* Math.cos(v))) \* Math.cos(u) \* Math.sin(v);

const denominator = Math.pow((3 + 4 \* Math.cos(u) \* Math.cos(v)), 2);

return numerator / denominator;

}

function clamp(v, min, max) {

return Math.max(min, Math.min(v, max));

}

function normalizeUV(value, min, max) {

return (value - min) / (max - min);

}

function generateNeoviusSurface(uSteps, vSteps, uRange, vRange) {

const vertices = [];

const indices = [];

const normals = [];

const tangents = [];

const uvs = [];

const [uMin, uMax] = uRange;

const [vMin, vMax] = vRange;

const du = (uMax - uMin) / uSteps;

const dv = (vMax - vMin) / vSteps;

function generator(swizzle) {

const firstIndex = vertices.length / 3;

for (let i = 0; i <= uSteps; i++) {

const u = uMin + i \* du;

for (let j = 0; j <= vSteps; j++) {

const v = vMin + j \* dv;

const x = u;

const y = v;

const cosU = Math.cos(u);

const cosV = Math.cos(v);

const z = Math.acos(clamp(-3 \* ((cosU + cosV) / (3 + 4 \* cosU \* cosV)), -1.0, 1.0));

vertices.push(...swizzle(x, y, z));

vertices.push(...swizzle(x, y, -z));

const tangent\_u = m4.normalize([1, 0, derivative\_z\_u(u, v)], []);

const tangent\_v = m4.normalize([0, 1, derivative\_z\_v(u, v)], []);

const [nx, ny, nz] = m4.normalize(m4.cross(tangent\_u, tangent\_v, []), [0, 0, 1]);

normals.push(...swizzle(nx, ny, nz));

normals.push(...swizzle(nx, ny, -nz));

tangents.push(...swizzle(tangent\_u[0], tangent\_u[1], tangent\_u[2]));

tangents.push(...swizzle(tangent\_u[0], tangent\_u[1], -tangent\_u[2]));

const texture\_u = normalizeUV(u, uMin, uMax);

const texture\_v = normalizeUV(v, vMin, vMax);

uvs.push(texture\_u, texture\_v, texture\_u, texture\_v);

}

}

for (let i = 0; i < uSteps; i++) {

for (let j = 0; j < vSteps; j++) {

const topLeft = 2 \* (i \* (vSteps + 1) + j) + firstIndex;

const topRight = 2 \* (i \* (vSteps + 1) + (j + 1)) + firstIndex;

const bottomLeft = 2 \* ((i + 1) \* (vSteps + 1) + j) + firstIndex;

const bottomRight = 2 \* ((i + 1) \* (vSteps + 1) + (j + 1)) + firstIndex;

indices.push(topLeft, bottomLeft, bottomRight);

indices.push(topLeft, bottomRight, topRight);

indices.push(topLeft + 1, bottomLeft + 1, bottomRight + 1);

indices.push(topLeft + 1, bottomRight + 1, topRight + 1);

}

}

}

generator((x, y, z) => [x, y, z]);

generator((x, y, z) => [x, z, y]);

generator((x, y, z) => [z, x, y]);

return { vertices, normals, tangents, uvs, indices };

}

export default function Model(gl, shProgram) {

this.iVertexBuffer = gl.createBuffer();

this.iUVBuffer = gl.createBuffer();

this.iNormalBuffer = gl.createBuffer();

this.iTangentBuffer = gl.createBuffer();

this.iIndexBuffer = gl.createBuffer();

this.idTextureDiffuse = LoadTexture(gl, "./textures/diffuse.jpg");

this.idTextureNormal = LoadTexture(gl, "./textures/normal.jpg");

this.idTextureSpecular = LoadTexture(gl, "./textures/specular.jpg");

this.point = [0.5, 0.5];

this.uvBuffer = [];

this.indexBuffer = [];

this.count = 0;

this.BufferData = function(vertices, normals, tangents, uvs, indices) {

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iVertexBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(vertices), gl.STATIC\_DRAW);

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iUVBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(uvs), gl.STATIC\_DRAW);

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iNormalBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(normals), gl.STATIC\_DRAW);

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTangentBuffer);

gl.bufferData(gl.ARRAY\_BUFFER, new Float32Array(tangents), gl.STATIC\_DRAW);

gl.bindBuffer(gl.ELEMENT\_ARRAY\_BUFFER, this.iIndexBuffer);

gl.bufferData(gl.ELEMENT\_ARRAY\_BUFFER, new Uint32Array(indices), gl.STATIC\_DRAW);

this.uvBuffer = uvs;

this.indexBuffer = indices;

this.count = indices.length;

};

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.iUVBuffer);

gl.vertexAttribPointer(shProgram.iAttribUV, 2, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(shProgram.iAttribUV);

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iNormalBuffer);

gl.vertexAttribPointer(shProgram.iAttribNormal, 3, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(shProgram.iAttribNormal);

gl.bindBuffer(gl.ARRAY\_BUFFER, this.iTangentBuffer);

gl.vertexAttribPointer(shProgram.iAttribTangent, 3, gl.FLOAT, false, 0, 0);

gl.enableVertexAttribArray(shProgram.iAttribTangent);

gl.activeTexture(gl.TEXTURE0);

gl.bindTexture(gl.TEXTURE\_2D, this.idTextureDiffuse);

gl.activeTexture(gl.TEXTURE1);

gl.bindTexture(gl.TEXTURE\_2D, this.idTextureNormal);

gl.activeTexture(gl.TEXTURE2);

gl.bindTexture(gl.TEXTURE\_2D, this.idTextureSpecular);

gl.uniform2fv(shProgram.iPoint, this.point);

gl.uniform1f(shProgram.iAngle, parseFloat(document.getElementById('Angle').value) \* (Math.PI / 180.0));

gl.drawElements(gl.TRIANGLES, this.count, gl.UNSIGNED\_INT, 0);

}

this.CreateSurfaceData = function() {

function get(name) {

return document.getElementById(name).value;

}

const uSteps = parseInt(get('USteps'));

const vSteps = parseInt(get('VSteps'));

const uRange = [parseFloat(get('UMin')), parseFloat(get('UMax'))];

const vRange = [parseFloat(get('VMin')), parseFloat(get('VMax'))];

const { vertices, normals, tangents, uvs, indices } = generateNeoviusSurface(uSteps, vSteps, uRange, vRange);

this.BufferData(vertices, normals, tangents, uvs, indices);

}

}