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**

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

Topic**:**

«Operations on texture coordinates»

Variant 2

Executed by:

student of the TP-41mp group

Boreichuck M.P.

Checked by:

Kaleniuk O. O.

Kyiv – 2024

**1. Task**

1. Reuse texture mapping from Control task.
2. Implement texture scaling (texture coordinates) scaling / rotation around user specified point- odd variants implement scaling, even variants implement rotation
3. 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**

#### **Utilizing Multiple Buffers in WebGL**

In WebGL, the performance and flexibility of rendering 3D scenes can be significantly improved by utilizing multiple buffers. Each buffer stores different types of data, such as vertex positions, normals, tangents, texture coordinates (UVs), and indices, allowing for more organized data management and optimized rendering performance.

**Advantages of Using Multiple Buffers:**

1. **Enhanced Modularity and Organization:** By storing different types of data in separate buffers, developers can easily modify individual elements of a 3D model. For instance, if only the UV coordinates need to be updated (such as for applying a different texture), the other buffers can remain unchanged, making updates more efficient.
2. **Rendering Optimization:** WebGL allows selective data updates with functions like gl.bufferSubData(), which ensures that only the data that has changed gets updated. This can drastically reduce the number of redraws and improve performance, especially in large or complex scenes.
3. **Increased Flexibility:** Multiple buffers enable rendering the same model with different datasets, such as textures, normals, or materials, without the need to recreate the entire model. This can be particularly useful for instances where models need to be rendered in different states or with different textures.

This process can be described textually as follows:

1. **Step 1:** Bind the buffer to the appropriate target using gl.bindBuffer.
2. **Step 2:** Load the UV data into the buffer using gl.bufferData, ensuring the data is in the correct format, typically a Float32Array, and setting the buffer usage to gl.STATIC\_DRAW.

To work with these buffers, it is essential to use shaders capable of interpreting the data stored in them. This is done using the gl.vertexAttribPointer() function, which tells WebGL how to map the buffer data to the appropriate shader attributes.

#### **Implementing Flat Shading**

Flat shading is a shading technique where the color or lighting is applied uniformly across the entire surface of a triangle. Unlike smooth shading techniques like Gouraud or Phong, which calculate lighting per vertex or pixel, flat shading assigns a single lighting intensity to each triangle.

**Core Characteristics of Flat Shading:**

1. **Single Normal Per Triangle:** Flat shading calculates a single normal vector for each triangle, which is used across the entire surface of the triangle. This creates sharp transitions between adjacent triangles, making the object appear more angular and faceted.
2. **Lighting Calculation:** Lighting is computed once per triangle rather than per pixel. This reduces computational overhead but also leads to a less smooth appearance compared to other shading techniques.

**Steps to Implement Flat Shading in WebGL:**

1. During data preparation, calculate the normal for each triangle. The normal is typically calculated as the cross product of two tangent vectors, and it must be normalized to ensure it has unit length.
2. In the vertex shader, pass the normal vector to the fragment shader as a flat variable. The flat qualifier in GLSL ensures that the normal value is not interpolated between vertices.
3. In the fragment shader, use the normal to calculate the lighting for the entire triangle, typically applying a simple lighting model like Lambertian reflection.

By simplifying the lighting model and removing interpolation, flat shading can result in a stylized, low-polygon look that is often used for geometric shapes, video games, or artistic renderings.

### **3. Implementation Details**

This feature allows textures to rotate interactively around a pivot point defined by the user, providing control and flexibility for texture manipulation in WebGL.

**Key Components and Logic**

1. Defining the Pivot Point: The rotation pivot is specified through a point attribute in the model:

this.point = [0.5, 0.5];

This point is passed as a uniform (iPoint) to the shader for rotation calculation on the GPU.

2. Controlling the Rotation Angle: The angle is adjustable via an HTML input element. The user inputs the angle in degrees, which is converted to radians and sent to the shader:

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

3. Shader Logic for Rotation: The vertex and fragment shaders apply the rotation by modifying UV coordinates. A 2D rotation matrix is used to compute the new UV positions. The pivot point is subtracted from the UV coordinates, rotated, and then added back to maintain the rotation around the correct point.

4. Updating UV Coordinates in the Shader: The rotation is applied to the UV coordinates in the vertex shader, which updates how the texture is mapped onto the surface. When the angle changes, the shader dynamically recalculates the UVs, ensuring smooth rotation.

5. Rendering the Model: Before rendering the model, the texture and rotation parameters (iPoint and iAngle) are passed to the shader:

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

gl.uniform1f(shProgram.iAngle, rotationAngle);

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

**UI Interaction**

The UI consists of HTML elements allowing the user to control the rotation angle:

1. An input field (#Angle) to specify the rotation angle.
2. Event listeners capture changes and trigger a redraw of the scene when the angle is updated.

Helper Functions

1. normalizeUV: Ensures UV coordinates are within the valid range of [0, 1].
2. LoadTexture: Handles texture loading and configuration, including wrapping and filtering.

**4. User's instruction**

### **Application UI Overview**

Upon loading the application, a 3D surface will be displayed on the canvas, accompanied by interactive controls beneath it. These controls include input fields and sliders that allow users to adjust various settings:

1. An input field for setting the number of segments.
2. An input field where the user can specify the number of circles to be rendered.
3. Sliders to adjust the position of the light source affecting the 3D model.
4. A slider to modify the rotation angle for the texture.

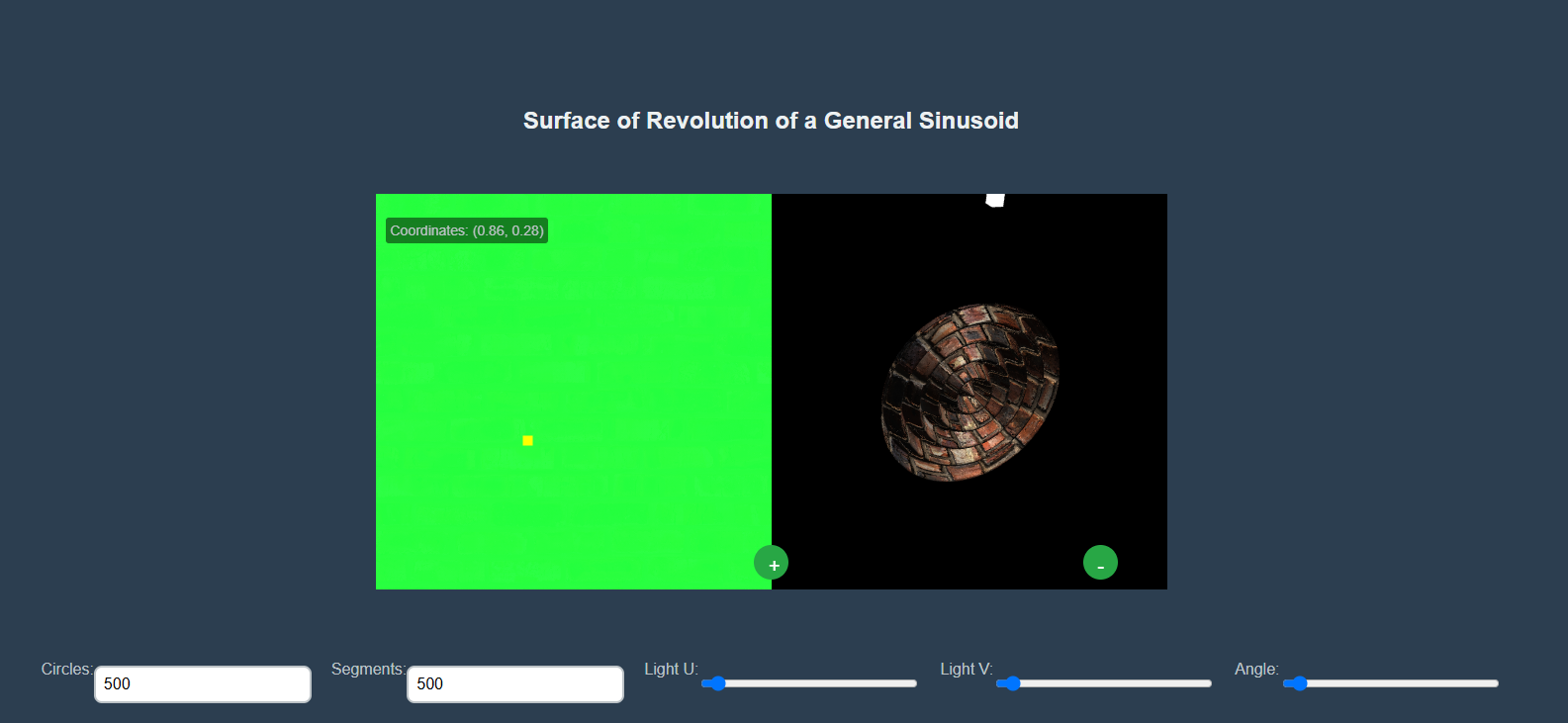
****

Image 1 - Application UI

### **Texture and UV Mapping Area**

Directly beneath the control panel, the UV Mapping View on the left side of the interface displays a 2D texture applied to the 3D surface. A yellow point is highlighted to indicate the selected UV coordinates, and the real-time coordinates of the chosen UV point (e.g., (0.13, 0.94)) are shown in the display.

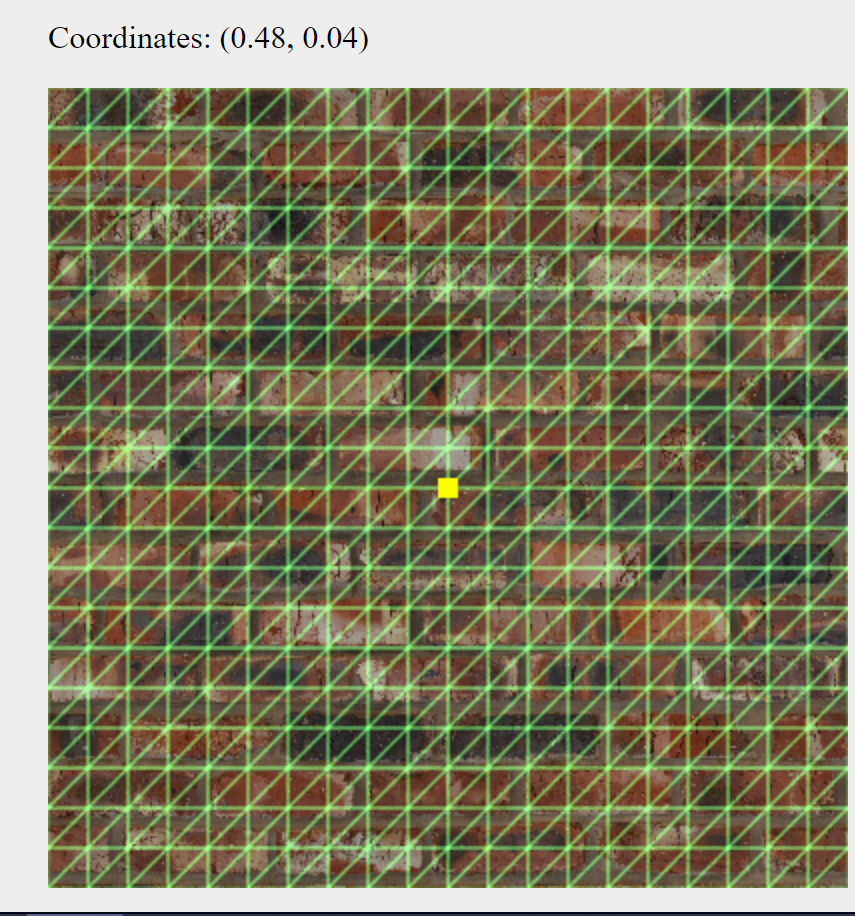


Image 2-3 - **UV Mapping View**

**Interactivity with Texture and 3D Model**

The user can interact with the texture and 3D model by clicking anywhere on the green UV mapping area on the left side. The clicked point will be marked with a yellow square. The Coordinates Display below the UV mapping view will then update to reflect the real-time UV coordinates of the selected point. Adjusting the slider allows for dynamic rotation of the texture on the 3D surface, helping to visualize how the texture aligns with the geometry of the object.

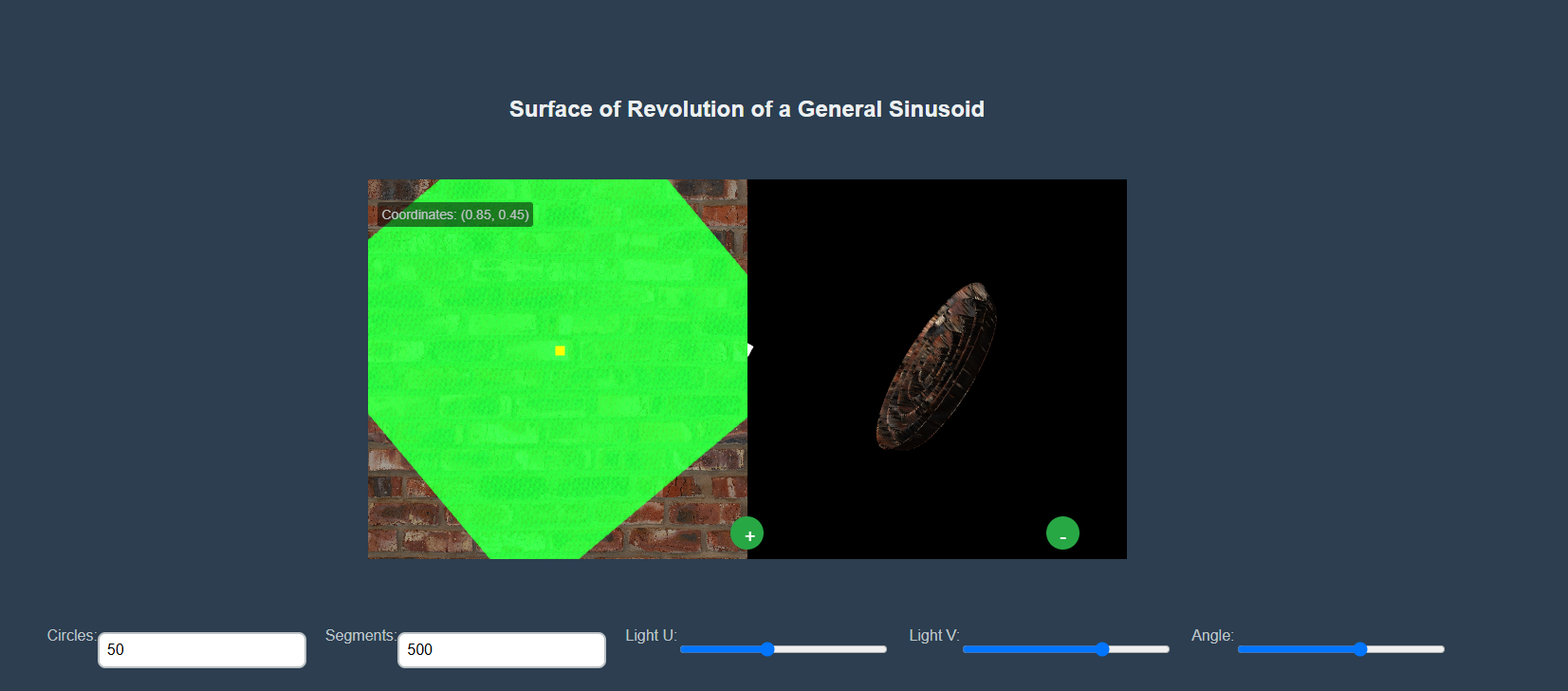
****

Figure 6 - Demonstration of object moving

**5. Sample of source code**

// Method to create a vertex based on parameters

    this.CreateVertex = function(a, n, R, r, b) {

        const x = r \* Math.cos(b),

              y = r \* Math.sin(b),

              z = a \* Math.cos(n \* Math.PI \* r / R);

        return [x, y, z];

    }

    // Method to compute partial derivative with respect to radius (r)

    this.partialDerivativeR = function(a, n, R, r, b) {

        const dx\_dr = Math.cos(b);

        const dy\_dr = Math.sin(b);

        const dz\_dr = -(a \* n \* Math.PI / R) \* Math.sin((n \* Math.PI \* r) / R);

        return m4.normalize([dx\_dr, dy\_dr, dz\_dr], []);

    }

    // Method to compute partial derivative with respect to angle (b)

    this.partialDerivativeB = function (a, n, R, r, b) {

        const dx\_db = -r \* Math.sin(b);

        const dy\_db = r \* Math.cos(b);

        const dz\_db = 0;

        return m4.normalize([dx\_db, dy\_db, dz\_db], []);

    }

    // Method to generate surface data (vertices, normals, tangents, UVs, and indices)

    this.CreateSurfaceData = function() {

        const a = 0.1, n = 1, R = 0.1, radius = 1;

        let vertices = [];

        let normals = [];

        let tangents = [];

        let uvs = [];

        let indices = [];

        const uSteps = parseInt(document.getElementById('circleCount').value);

        const vSteps = parseInt(document.getElementById('segmentsCount').value)

        const du = radius / uSteps;

        const dv = (2.0 \* Math.PI) / vSteps;

        // Generate vertices, normals, tangents, UVs, and indices for the surface

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

            const u = i \* du;

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

                const v = j \* dv;

                // Create vertex and add it to the vertices array

                vertices.push(...this.CreateVertex(a, n, R, u, v));

                // Compute tangent vectors and normal vector

                const tangent\_u = this.partialDerivativeR(a, n, R, u, v);

                const tangent\_v = this.partialDerivativeB(a, n, R, u, v);

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

                // Add normal, tangent, and UV data

                normals.push(...normal);

                tangents.push(...tangent\_u);

                uvs.push(normalizeUV(u, 0.0, radius), normalizeUV(v, 0.0, 2.0 \* Math.PI));

            }

        }

        // Generate indices for drawing the surface as triangles

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

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

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

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

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

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

                // Add two triangles for each quad

                indices.push(topLeft, bottomLeft, bottomRight);

                indices.push(topLeft, bottomRight, topRight);

            }

        }

        // Buffer the data to WebGL buffers

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