Ministry of education and science of Ukraine
National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnical Institute”
Educational and Scientific Institute
of Atomic and Thermal Energy

Department DCE

Calculation Graphical Work
credit module:
“Visualisation of graphical and geometrical information”

Done by:
student of 1-st course, IATE
group TR-42mp
Ivashchuk V.A.
Verified by:
Demchyshin A.A
Kalenyk A.S

Kyiv – 2024

Task description

Requirements

- 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.

Report

Prepare a digital report that holds:

- a title page;
- a chapter describing the task (1 page);
- a chapter describing theory (2 pages);
- a chapter describing the implementation details (2 pages);
- a chapter of user's instruction with screenshots (2 pages);
- a sample of source code (2 pages).

Theory

WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering 2D and 3D graphics within any compatible web browser without the need for plug-ins. It is based on OpenGL ES and enables GPU-accelerated graphics rendering, allowing developers to create rich, interactive visual experiences directly in the browser. WebGL is widely used in applications such as games, simulations, data visualization, and interactive websites. It integrates seamlessly with other web technologies like HTML5 and CSS, making it a powerful tool for web developers.

Textures in WebGL

Textures are images that are applied to geometric objects. WebGL uses 2D textures that can be applied to the surface of an object, creating realistic effects. Texture mapping uses a coordinate system called texture coordinates (UV coordinates). These coordinates specify which part of the texture should be used for each fragment of the object (pixel). The texture is thus "projected" onto the object using these coordinates.

Transformation Matrices

In 3D graphics, matrices are used to transform objects, including their rotation, scaling, and translation. In this project, matrices are used to:

- Model texture rotation: To rotate a texture around a certain point, a standard rotation matrix is used.
- Scaling texture: To change the size of a texture, a scaling matrix is used.

Scaling is the process of changing the dimensions of an object. In the context of texturing, scaling can change the dimensions of a texture that is applied to an object, creating the effect of it moving closer or farther away.

To scale a texture, you need to apply a scaling matrix, which changes the texture coordinates. An example of a scaling matrix:

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where s_x,s_y,s_z - — scaling factors along the X, Y and Z axes.

Rotating an object around a specific axis or point is also an important aspect of working in 3D graphics. In the context of texturing, rotation is applied to texture coordinates to rotate the texture around a selected point (for example, around the center of the object).

An example of a rotation matrix for rotation around the Z axis:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where θ rotation angle. When texture coordinates are multiplied by this matrix, the texture is rotated around the Z axis by an angle θ

Implementation Details

<u>Preparing the texture</u>

The first step is to load the texture image, create a texture object in WebGL, and then pass it to the shaders to be applied to the object. The texture coordinates indicate what part of the texture will be displayed on the surface of the object.

Scaling and Rotation

Scaling changes the texture coordinates, allowing you to increase or decrease its display. Rotation changes the coordinates depending on the rotation angle, which affects the location of the texture on the object. It is important that the rotation occurs around the point selected by the user.

<u>Managing the rotation point</u>

One of the key features of this lab is the ability to control the texture's rotation point. This is done using a keyboard input system where the user can move the rotation point along two axes (u and v). For example, the "A" and "D" keys can move the point along the U axis, and the "W" and "S" keys can move the point along the V axis.

Each key press changes the coordinates of the rotation point, allowing the position of the point around which the rotation occurs to change. This allows the user to observe how the texture's rotation changes depending on the position of the rotation point on the object's surface.

<u>Handling keyboard input</u>

To implement keyboard control of the pivot point, you need to track keystrokes and update the corresponding coordinates of the point. In WebGL, keyboard input is processed through standard JavaScript events. Each keystroke calls a function that changes the position of the pivot point based on the current coordinates.

When processing keystrokes, it is also important to update the scene state and redraw the object with a new texture so that the changes are visible to the user. This requires working with buffers, which are responsible for storing data about vertices and texture coordinates, as well as shaders for rendering the object with a new texture.

<u>Rendering the scene</u>

Once all the transformations and pivot point changes have been performed, it is necessary to render the object with the texture applied to it. This is done through shaders that interpret the texture coordinates and apply the texture to the surface of the object. It is important that the texture coordinates are updated every time the user changes the pivot point or performs transformations.

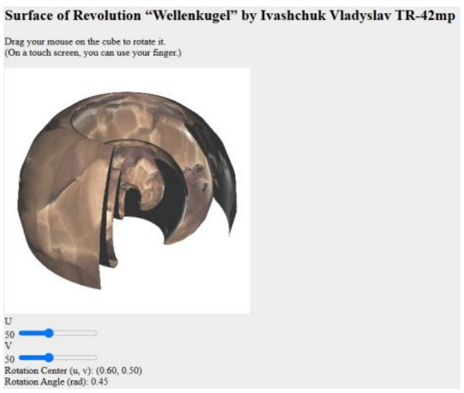
The rendering process includes the following steps:

- Updating the buffer with vertices and texture coordinates.
- Applying new texture coordinates taking into account the transformations performed.
- Sending data to shaders to render the object with the texture.

User instructions

In order to run the project, you need to follow the link and download the repository.

Open the branch of the CGV, then run any HTTP server in the project folder. Then connect to your local HTTP using a browser. Here you will see the page with figure like on image 1.



Surface of Revolution “Wellenkugel” by Ivashchuk Vladyslav TR-42mp

Drag your mouse on the cube to rotate it.
(On a touch screen, you can use your finger.)

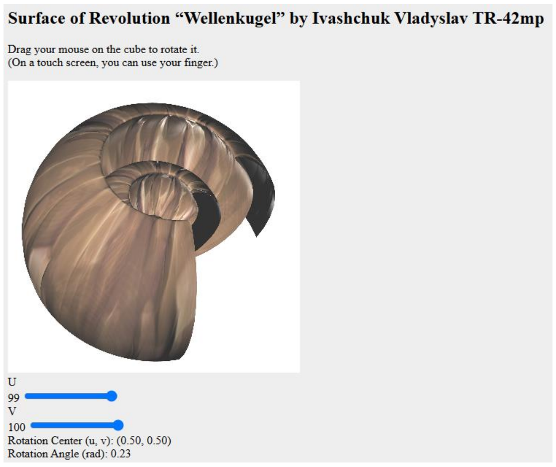
U
50
V
50
Rotation Center (u, v): (0.60, 0.50)
Rotation Angle (rad): 0.45

image 1

Here you can rotate the figure by holding LMB and moving the mouse. Also you can change granularity by decreasing or increasing value on sliders for U and V.

By pressing A,D,W,S you can change the position of the point on the surface, around which texture scaling occurs.

By pressing R user can rotate the texture. Example of object with changed rotation angle is on the image 2.



Surface of Revolution “Wellenkugel” by Ivashchuk Vladyslav TR-42mp

Drag your mouse on the cube to rotate it.
(On a touch screen, you can use your finger.)

U
99
V
100
Rotation Center (u, v): (0.50, 0.50)
Rotation Angle (rad): 0.23

image 2

Code samples

```
export class Model {
    constructor(name, uSteps = 50, vSteps = 50) {
        this.name = name;
        this.uSteps = uSteps;
        this.vSteps = vSteps;
        this.uLines = [];
        this.vLines = [];
        this.vertices = [];
        this.indices = [];
        this.normals = [];
        this.texCoords = [];
        this.tangents = [];
        this.textures = {};
        this.rotationCenter = { u: 0.5, v: 0.5 };
        this.rotationAngle = 0;
    }

    createSurfaceData() {
        this.generateUVLines();
        this.generateVertices();
        this.generateIndices();
        this.generateTexCoords();
        this.generateTangents();
        this.generateNormals();
    }

    generateUVLines() {
```

```
                const uMin = 0, uMax = 14.5, vMin = 0, vMax = 1.5 * Math.PI;
                const du = (uMax - uMin) / this.uSteps;
                const dv = (vMax - vMin) / this.vSteps;

                this.uLines = Array.from({ length: this.uSteps + 1 }, (_, uIndex) => {
                    const u = uMin + uIndex * du;
                    return Array.from({ length: this.vSteps + 1 }, (_, vIndex) => {
                        const v = vMin + vIndex * dv;
                        const x = u * Math.cos(Math.cos(u)) * Math.cos(v);
                        const y = u * Math.cos(Math.cos(u)) * Math.sin(v);
                        const z = u * Math.sin(Math.cos(u));
                        return [x, y, z];
                    });
                });

                this.vLines = this.uLines[0].map((_, vIndex) => this.uLines.map(uLine => uLine[vIndex]));
            }

            generateVertices() {
                this.vertices = this.uLines.flat(2);
            }

            generateIndices() {
                this.indices = [];
                for (let u = 0; u < this.uSteps; u++) {
                    for (let v = 0; v < this.vSteps; v++) {
                        const topLeft = u * (this.vSteps + 1) + v;
                        const topRight = topLeft + 1;
                        const bottomLeft = (u + 1) * (this.vSteps + 1) + v;
                        const bottomRight = bottomLeft + 1;
                        this.indices.push(topLeft, bottomLeft, topRight, topRight, bottomLeft,
bottomRight);
                    }
                }
            }

            generateTexCoords() {
                this.texCoords = [];
                for (let uIndex = 0; uIndex <= this.uSteps; uIndex++) {
                    for (let vIndex = 0; vIndex <= this.vSteps; vIndex++) {
                        this.texCoords.push(uIndex / this.uSteps, vIndex / this.vSteps);
                    }
                }
            }

            generateTangents() {
                this.tangents = Array(this.texCoords.length / 2).fill([1, 0, 0]).flat();
            }

            generateNormals() {
                const normals = new Array(this.vertices.length).fill(0);
                for (let i = 0; i < this.indices.length; i += 3) {
                    const i1 = this.indices[i] * 3, i2 = this.indices[i + 1] * 3, i3 = this.indices[i + 2]
* 3;
                    const v1 = this.vertices.slice(i1, i1 + 3);
                    const v2 = this.vertices.slice(i2, i2 + 3);
                    const v3 = this.vertices.slice(i3, i3 + 3);
                    const normal = this.calculateFaceNormal(v1, v2, v3);
                    for (let j = 0; j < 3; j++) {
                        const idx = this.indices[i + j] * 3;
                        normals[idx] += normal[0];
                        normals[idx + 1] += normal[1];
                        normals[idx + 2] += normal[2];
                    }
                }
                this.normals = this.normalizeVectors(normals);
            }

            calculateFaceNormal(v1, v2, v3) {
                const edge1 = v2.map((val, i) => val - v1[i]);
                const edge2 = v3.map((val, i) => val - v1[i]);
                return [
                    edge1[1] * edge2[2] - edge1[2] * edge2[1],
                    edge1[2] * edge2[0] - edge1[0] * edge2[2],
                    edge1[0] * edge2[1] - edge1[1] * edge2[0],
                ];
            }

            normalizeVectors(vectors) {
                return vectors.reduce((normalized, _, i, arr) => {
                    if (i % 3 === 0) {
                        const length = Math.sqrt(arr[i] ** 2 + arr[i + 1] ** 2 + arr[i + 2] ** 2);
                        if (length > 0) {
                            normalized.push(arr[i] / length, arr[i + 1] / length, arr[i + 2] / length);
                        } else {
                            normalized.push(0, 0, 0);
                        }
                    }
                    return normalized;
                }, []);
            }

            loadTexture(gl, url) {
                const texture = gl.createTexture();
                gl.bindTexture(gl.TEXTURE_2D, texture);
                gl.texImage2D(gl.TEXTURE_2D,  0,  gl.RGBA,  1,  1,  0,  gl.RGBA,  gl.UNSIGNED_BYTE,  new
Uint8Array([0, 0, 0, 0]));

                const image = new Image();
                image.onload = () => {
                    gl.bindTexture(gl.TEXTURE_2D, texture);
                    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
                    if (isPowerOf2(image.width) && isPowerOf2(image.height)) {
                        gl.generateMipmap(gl.TEXTURE_2D);
```

```
        } else {
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
            gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
        }
    };
    image.src = url;
    return texture;
}

loadTextures(gl) {
    this.textures = {
        diffuse: this.loadTexture(gl, "assets/Stylized_Cliff_Rock_006_basecolor.png"),
        specular: this.loadTexture(gl, "assets/Stylized_Cliff_Rock_006_height.png"),
        normal: this.loadTexture(gl, "assets/Stylized_Cliff_Rock_006_normal.png"),
    };
}

bindTextures(gl, program) {
    const textureUnits = ["diffuse", "specular", "normal"];
    textureUnits.forEach((type, idx) => {
        gl.activeTexture(gl[`TEXTURE${idx}`]);
        gl.bindTexture(gl.TEXTURE_2D, this.textures[type]);
        gl.uniform1i(program[`${type}TextureUni`], idx);
    });
}

initBuffer(gl) {
    this.vertexBuffer = this.createBuffer(gl, this.vertices);
    this.indexBuffer = this.createBuffer(gl, this.indices, gl.ELEMENT_ARRAY_BUFFER);
    this.normalBuffer = this.createBuffer(gl, this.normals);
    this.texCoordBuffer = this.createBuffer(gl, this.texCoords);
    this.tangentBuffer = this.createBuffer(gl, this.tangents);
}

createBuffer(gl, data, target = gl.ARRAY_BUFFER) {
    const buffer = gl.createBuffer();
    gl.bindBuffer(target, buffer);
    gl.bufferData(target, new Float32Array(data), gl.STATIC_DRAW);
    return buffer;
}

draw(gl, program) {
    const buffers = [
        { buffer: this.vertexBuffer, attrib: program.vertexAttrib, size: 3 },
        { buffer: this.normalBuffer, attrib: program.normalAttrib, size: 3 },
        { buffer: this.texCoordBuffer, attrib: program.texCoordAttrib, size: 2 },
        { buffer: this.tangentBuffer, attrib: program.tangentAttrib, size: 3 },
    ];

    buffers.forEach(({ buffer, attrib, size }) => {
        gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
```