Ministry of Education and Science of Ukraine
National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”
Educational and Research Institute of Nuclear and Thermal Energy
Department of Digital Technologies in Energy

Calculation and graphics work

Credit module
Visualization of graphical and geometric information

Performed by: student of group TR-41mp
Posternak A. V. (variant #17)
Checked by: Candidate of Technical Sciences, Associate Professor
Demchyshyn A. A.

Kyiv – 2024

1. TASK DEFINITION

Assignment presentation:

– share the link to GIThub repository with implemented assignments through the form (https://docs.google.com/forms/d/e/1FAIpQLSfnPxxMWAqfMGMseu7YBJ7zXLIWXIglVatnw1pslh97iTusGA/viewform);

– make sure the repository contains a paper presenting assignment;

– the practical assignment has to reside in a branch named CGW.

Operations on texture coordinates – scaling a texture around a user specified point.

Requirements:

– reuse texture mapping from Control task;

– implement texture scaling (texture coordinates) scaling around user specified point;

– 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 FOUNDATIONS

Flat shading is a fundamental shading technique in computer graphics used to render 3D surfaces with a uniform color per polygon. In this method, each polygon, typically a triangle or a quad, is shaded based on a single normal vector, which is perpendicular to the surface of the polygon. This contrasts with smooth shading techniques, such as Gouraud or Phong shading, where normals are interpolated across the vertices to create gradual lighting transitions and a more realistic appearance.

The primary characteristic of flat shading is its ability to clearly delineate each polygon, resulting in a faceted look where individual faces are distinctly visible. This can be advantageous for rendering low-polygon models, architectural structures, or stylized graphics where a clear geometric form is desired. Additionally, flat shading is computationally less intensive than smooth shading since it requires fewer calculations for lighting per pixel.

Texturing is a critical technique in computer graphics that enhances the visual richness of 3D models without increasing their geometric complexity. It involves mapping a 2D image, known as a texture, onto the surface of a 3D object. This process adds intricate details such as colors, patterns, and surface imperfections, making the model appear more realistic and visually appealing.

The process begins with assigning texture coordinates, or UV coordinates, to each vertex of the 3D model. These coordinates define how the 2D texture image wraps around the 3D geometry. The U coordinate typically represents the horizontal axis, while the V coordinate represents the vertical axis of the texture image. By interpolating these coordinates across the surface of the polygons, each fragment (or pixel) can accurately sample the corresponding texel (texture pixel) from the texture image.

Advanced texturing techniques include the use of multiple texture maps, such as diffuse maps for base color, specular maps for shininess, and normal maps for simulating surface details. Normal mapping, for instance, perturbs the surface normals based on the

texture, allowing for detailed lighting effects without altering the actual geometry. This enhances the perception of depth and complexity on the surface, contributing to more realistic renderings.

Additionally, texture filtering methods, such as bilinear and trilinear filtering, are employed to improve the visual quality of textures when they are minified or magnified. These techniques smooth out the texture appearance, reducing aliasing artifacts and enhancing the overall image quality.

Texture scaling is a technique that adjusts the size of a texture applied to a 3D surface. This allows for zooming in on specific areas, creating dynamic visual effects, or highlighting particular details without changing the model's geometry. By manipulating the U and V texture coordinates, textures can be scaled uniformly or non-uniformly, offering a variety of visual outcomes.

The scaling process involves altering the texture coordinates based on a scaling factor and a chosen center point. A scaling factor greater than one enlarges the texture, effectively zooming in and repeating the texture pattern over the surface. Conversely, a factor less than one shrinks the texture, fitting more of it within the same area. To scale around a specific point, the texture coordinates are first translated so that the center point becomes the origin, scaled, and then translated back. This localized scaling maintains texture alignment and prevents distortion across the entire surface.

Clamping the scaled texture coordinates within the [0, 1] range ensures that texture sampling stays within the texture image boundaries, avoiding wrapping or stretching artifacts. Texture scaling is particularly valuable in interactive applications, enabling users to focus on specific texture regions, such as in mapping tools, detailed 3D model inspections, or dynamic visualizations that respond to user input. Real-time control over texture scaling parameters enhances user engagement and customization.

3. IMPLEMENTATION DETAILS

In the WebGL CGW project, shading is managed using the Phong lighting model within the fragment shader, which performs smooth shading by interpolating normals across vertices. This approach calculates lighting for each pixel based on the interpolated normals, resulting in gradual lighting changes and a more realistic appearance. While the current implementation uses smooth shading, transitioning to flat shading would involve modifying the vertex shader to pass a single normal per polygon to the fragment shader, eliminating normal interpolation and creating a faceted look.

Texturing is implemented by assigning texture coordinates to each vertex of the 3D surface. The vertex shader receives these coordinates through the aTexCoord attribute and passes them to the fragment shader via the vTexCoord varying. The fragment shader then samples the diffuse, specular, and normal textures using these coordinates to determine the final color and lighting of each pixel. This setup allows the application of multiple texture maps to define different material properties, enhancing the visual complexity and realism of the rendered surface.

Texture scaling is achieved by adjusting the texture coordinates within the vertex shader using the uTextureScale and uStartScalePoint uniforms. The shader first translates the texture coordinates relative to the scaling center (uStartScalePoint), applies the scaling factor (uTextureScale), and then translates them back. This process magnifies or reduces the texture around the specified point without altering the entire texture uniformly. Clamping ensures that the scaled texture coordinates remain within the valid [0,1] range, preventing visual artifacts.

The project incorporates interactive controls that allow users to move the scaling center dynamically along the (u, v) parameter space using keyboard inputs. Specifically, the A and D keys move the scaling point along the u parameter, while the W and S keys adjust its position along the v parameter. These inputs update the paramU and paramV variables, which are then converted to texture coordinates and passed to the shader. A

visual red sphere is rendered at the scaling center to provide real-time feedback, helping users understand the exact location of the scaling operation as they manipulate the texture scaling parameters.

A red sphere is rendered at the current scaling center on the surface, offering immediate visual feedback to users as they adjust the (u, v) parameters. This visual indicator helps users precisely locate and manipulate the area of the texture they wish to scale, enhancing the interactivity and usability of the texture scaling feature.

The integration of flat shading concepts, comprehensive texturing, and dynamic texture scaling within the WebGL CGW project provides a robust framework for rendering detailed and interactive 3D graphics. By leveraging shader programs and user input mechanisms, the project effectively manages visual complexity and offers users intuitive control over texture manipulation, resulting in an engaging and customizable rendering experience.

4. USAGE INSTRUCTIONS

To utilize the WebGL Texture Scaling project, it is recommended to run the application on a local server to ensure all resources load correctly. This can be easily achieved, for example, using Python's built-in server or Visual Studio Code Online Server extension. Once the server is active, accessing http://localhost:port in a web browser will launch the application.

Upon launching (Image 4.1), the application displays a 3D model with applied textures. Interaction with the texture scaling feature is facilitated through keyboard controls: pressing the A and D keys moves the scaling center left and right along the u parameter, while pressing the W and S keys shifts it upward and downward along the v parameter. These inputs modify the paramU and paramV variables in real-time, resulting in dynamic updates to the texture scaling around the designated point.

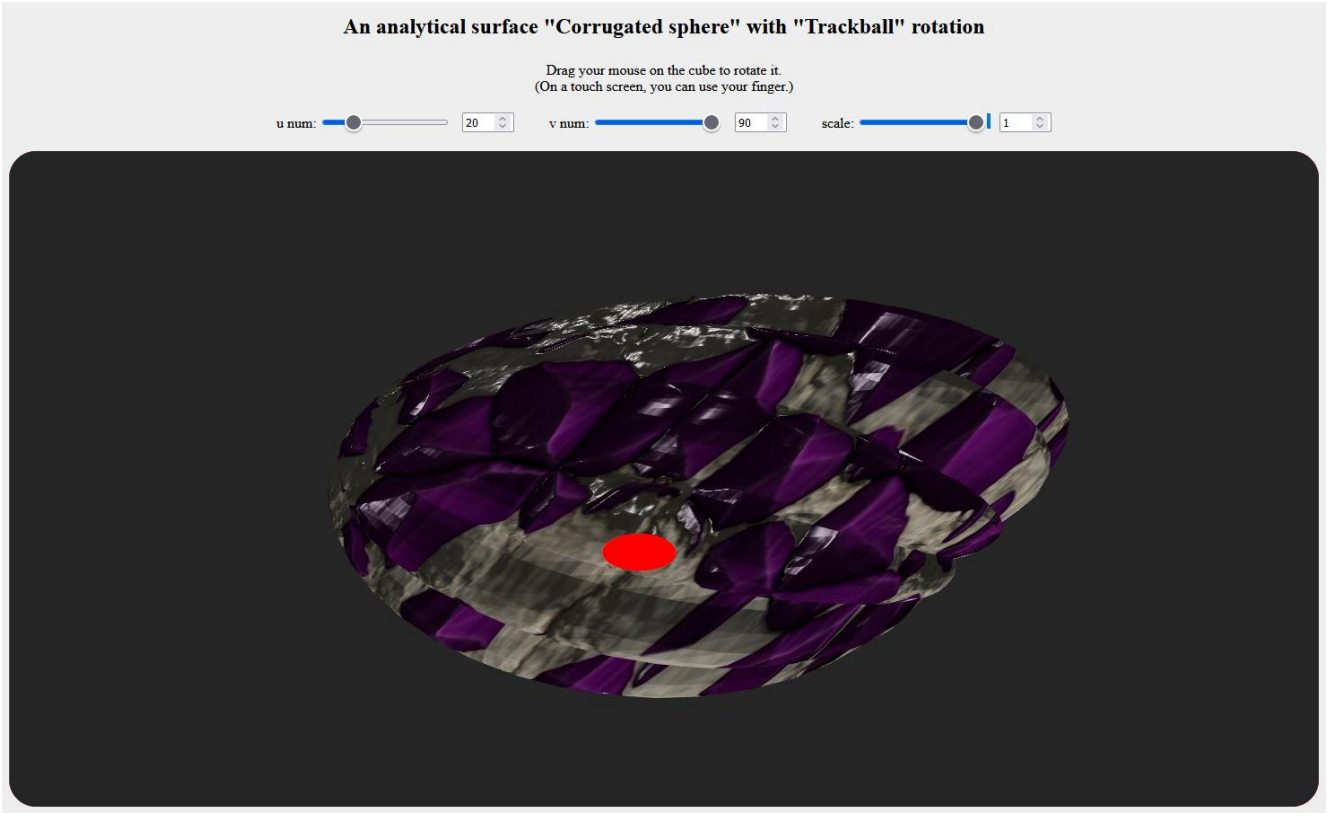

Image 4.1 – Visualization with scale value 1

A red sphere is rendered on the surface at the current scaling center, providing immediate visual feedback of the adjustments made.

As the texture scaling parameters are manipulated (Image 4.2), the texture on the model dynamically zooms in or out around the red sphere, allowing for focused examination of specific areas or the creation of various visual effects. The 3D model can be rotated to view the effects of texture scaling from different perspectives.

For the best experience, it is advisable to avoid making rapid or excessive changes to the scaling parameters, especially on devices with limited graphics capabilities, to maintain smooth rendering performance. By following these instructions, users can effectively explore and utilize the texture scaling features of the WebGL project, enhancing their understanding of interactive 3D graphics rendering.

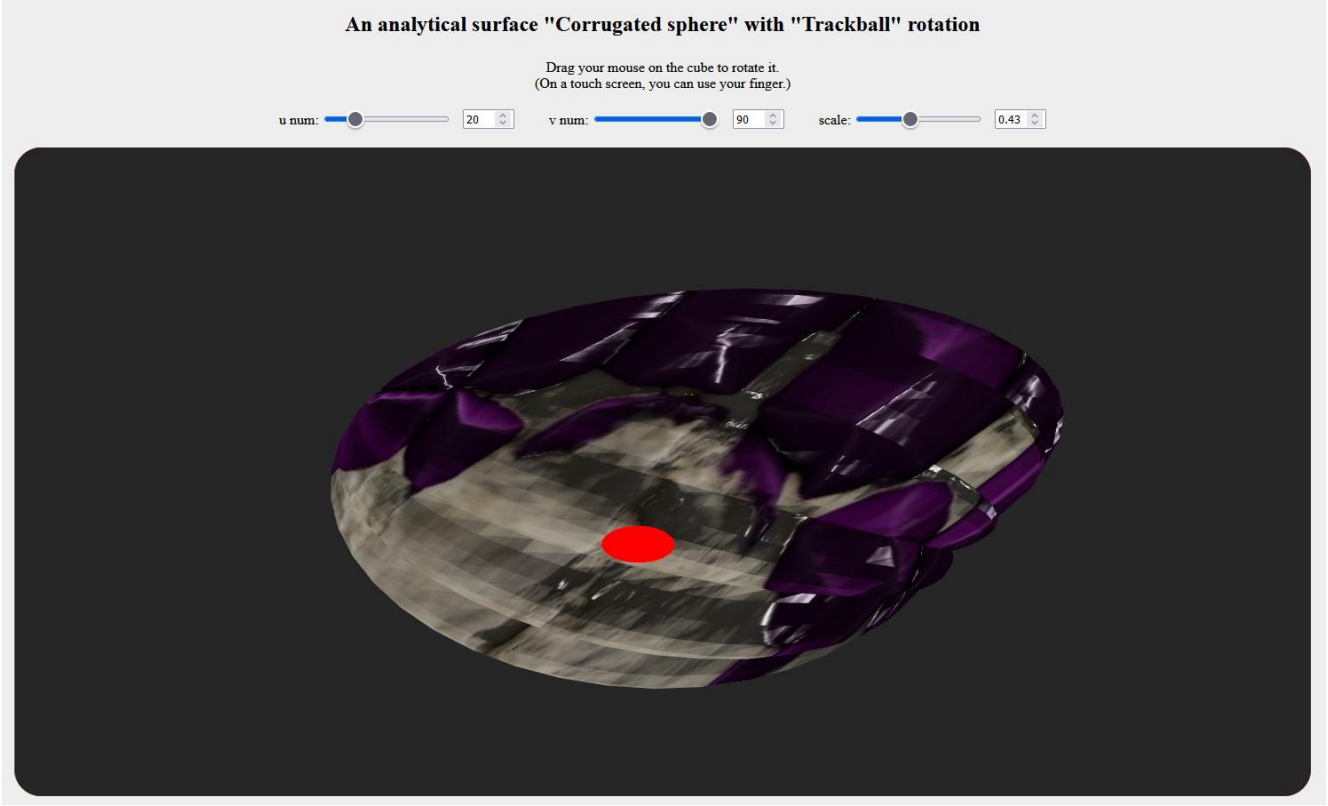

Image 4.2 – Visualization with scale value 0,43

5 SOURCE CODE SAMPLES

shader.gpu

```
// Vertex Shader
const vertexShaderSource =
`attribute vec3 aPosition;
attribute vec3 aNormal;
attribute vec3 aTangent;
attribute vec2 aTexCoord;

uniform mat4 uModelViewMatrix;
uniform mat4 uProjectionMatrix;
uniform mat4 uNormalMatrix;

// Additional uniforms for texture scaling
uniform float uTextureScale;
uniform vec2 uStartScalePoint;

// Varying variables to fragment shader
varying vec3 vPosWorld;
varying vec2 vTexCoord;
varying mat3 vTBN;

void main() {
    // 1) World-space position
```

```
    vec4 posWorld4 = uModelViewMatrix * vec4(aPosition, 1.0);
    vPosWorld = posWorld4.xyz;

    // 2) Transform normal
    vec3 N = normalize(mat3(uNormalMatrix) * aNormal);

    // 3) Transform tangent, orthogonalize
    vec3 T = normalize(mat3(uNormalMatrix) * aTangent);
    T = normalize(T - dot(T, N) * N);

    // 4) Bitangent
    vec3 B = cross(N, T);

    // 5) TBN
    vTBN = mat3(T, B, N);

    // 6) Texture scaling around center
    vec2 shifted = aTexCoord - uStartScalePoint;

    shifted *= uTextureScale;
    shifted += uStartScalePoint;
    vTexCoord = clamp(shifted, 0.0, 1.0); // Clamping to [0,1]

    // 7) Final position
    gl_Position = uProjectionMatrix * posWorld4;
}`;
```

```
// Fragment Shader
const fragmentShaderSource =
`#ifdef GL_FRAGMENT_PRECISION_HIGH
   precision highp float;
#else
   precision mediump float;
#endif

// Lighting
uniform vec3  uLightPos;
uniform float uAmbientFactor;
uniform float uDiffuseFactor;
uniform float uSpecularFactor;
uniform float uShininess;

// Base color
uniform vec4 uColor;

// Camera direction
uniform vec3 uViewDir;

// Textures
uniform sampler2D uDiffuseSampler;
uniform sampler2D uSpecularSampler;
uniform sampler2D uNormalSampler;
```

```
// Varyings
varying vec3 vPosWorld;
varying vec2 vTexCoord;
varying mat3 vTBN;

void main() {
    // 1) Normal from normal map, transform to world space
    vec3 textureNormal = texture2D(uNormalSampler, vTexCoord).rgb;
    textureNormal = 2.0 * textureNormal - 1.0;
    vec3 N = normalize(vTBN * textureNormal);

    // 2) Light direction
    vec3 L = normalize(uLightPos - vPosWorld);

    // 3) View direction
    vec3 V = normalize(uViewDir);

    // 4) Reflection vector
    vec3 R = reflect(-L, N);

    // 5) Phong components
    float ambient  = uAmbientFactor;
    float diff     = max(dot(N, L), 0.0);
    float diffuse  = uDiffuseFactor * diff;
    float specTerm = max(dot(R, V), 0.0);
```

```
            float specular = uSpecularFactor * pow(specTerm, uShininess);

            // Textures
            vec3 diffuseTex  = texture2D(uDiffuseSampler,  vTexCoord).rgb;
            vec3 specularTex = texture2D(uSpecularSampler, vTexCoord).rgb;

            vec3 color = diffuseTex * diffuse + specularTex * specular + (ambient
* diffuseTex);

            // Write final pixel color
            gl_FragColor = vec4(color, 1.0) * uColor;
        }`;
```