BBM414 Computer Graphics Lab.
Assignment 5

Uygar ERSOY
2220356114
Department of Computer Engineering
Hacettepe University
Ankara, Turkey

Overview

This project involves creating a 3D scene with interactive camera controls and dynamic object animations. In Part 1, a cube is rendered, and its view is updated using event listeners, providing a foundation for exploring 3D transformations and user input handling. Part 2 builds upon this by introducing lighting, and more complex animations. A scene is created with a sphere and an indoor plant, where the sphere has a radius of 15 units and is positioned at the center of the world. The plant object, read from external files, is placed on the sphere. A directional light is positioned above the sphere, rotating dynamically around the y-axis while illuminating the scene. Both the sphere and plant rotate around the y-axis, adding further complexity to the animation. A controllable camera allows users to navigate and interact with the scene. This project emphasizes mastering 3D object manipulation, dynamic lighting, and responsive camera controls.

1 Part 1 - WebGL2 Cube with Texture and Lighting

In this section, a 3D rotating cube with texture mapping and dynamic lighting is rendered. The cube's faces are created using the `quad` function, and lighting is adjustable via buttons that modify the light's position. The cube can be rotated along different axes, with an applied texture ("umbrella.png"). Shaders handle the lighting and texture effects to enhance the 3D appearance.

1.1 Shader Setup

Two shaders are used in this project: a vertex shader and a fragment shader.

1.1.1 Vertex Shader

The vertex shader is responsible for transforming the vertices of the cube and computing the lighting effects based on the Phong reflection model. It takes the vertex position, normal, and texture coordinates as inputs, computes the ambient, diffuse, and specular lighting contributions, and outputs the final color and texture coordinates.

```
#version 300 es
in vec4 vPosition;
in vec3 vNormal;
in vec2 vTexCoord;

out vec4 fColor;
out vec2 fTexCoord;
```

```
uniform vec4 ambientProduct, diffuseProduct, specularProduct;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec4 lightPosition;
uniform float shininess;

void main()
{
    vec3 pos = -(modelViewMatrix * vPosition).xyz;
    vec3 light = lightPosition.xyz;
    vec3 L = normalize( light - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    vec4 NN = vec4(vNormal, 0);
    vec3 N = normalize( (modelViewMatrix * NN).xyz );

    vec4 ambient = ambientProduct;
    float Kd = max( dot(L, N), 0.0 );
    vec4 diffuse = Kd * diffuseProduct;

    float Ks = pow( max(dot(N, H), 0.0), shininess );
    vec4 specular = Ks * specularProduct;

    if (dot(L, N) < 0.0) {
        specular = vec4(0.0, 0.0, 0.0, 1.0);
    }

    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
    fColor = ambient + diffuse + specular;
    fColor.a = 1.0;

    fTexCoord = vTexCoord;
}
```

1.1.2 Fragment Shader

The fragment shader is responsible for combining the lighting effects computed by the vertex shader with the texture color. It samples the texture using the texture coordinates and multiplies the texture color by the final lighting color to produce the final pixel color.

```
#version 300 es

precision mediump float;

in vec4 fColor;
in vec2 fTexCoord;

out vec4 oColor;

uniform sampler2D tex;

void main()
{
    vec4 texColor = texture(tex, fTexCoord);
    oColor = texColor * fColor;
}
```

1.2 WebGL2 Cube with Lighting and Textures Implementation

This section explains the WebGL code for rendering a rotating cube with lighting effects and textures applied. The code is divided into different segments for clarity.

1.2.1 Global Variables

The global variables define the properties of the cube, lighting, and material. This includes the vertices, normals, textures, light and material properties, and transformation matrices.

```
var canvas;
var gl;

var numVertices  = 36;
var pointsArray = [];
var normalsArray = [];
var textureArray = [];

var vertices = [
    vec4( -0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5,  0.5,  0.5, 1.0 ),
    vec4( 0.5,  0.5,  0.5, 1.0 ),
    vec4( 0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5, -0.5, -0.5, 1.0 ),
    vec4( -0.5,  0.5, -0.5, 1.0 ),
    vec4( 0.5,  0.5, -0.5, 1.0 ),
    vec4( 0.5, -0.5, -0.5, 1.0 )
];
var textureCoordinates = [
    vec2(0, 0),
    vec2(0, 1),
    vec2(1, 1),
    vec2(1, 0)
];
var ctm;
var ambientColor, diffuseColor, specularColor;
var modelView, projection;
var viewerPos;
```

This segment defines global variables and arrays, including the vertices for the cube, lighting and material properties.

1.2.2 Lighting and Material Setup

The lighting and material properties are defined in this section. These values are used in the shading calculations to achieve realistic lighting effects on the cube's surfaces.

```
var lightPosition = vec4(1.0, 1.0, 1.0, 0.0 );
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0 );
var lightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var lightSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );

var materialAmbient = vec4(1.0, 1.0, 1.0, 1.0);
var materialDiffuse = vec4(1.0, 1.0, 1.0, 1.0);
var materialSpecular = vec4(1.0, 1.0, 1.0, 1.0);

var materialShininess = 100.0;
```

The light and material properties, such as ambient, diffuse, and specular reflection, as well as material shininess, are defined to control how the cube reacts to light.

1.2.3 Lighting Position Updates

This section includes event handlers that update the light source position based on button clicks. These changes are sent to the shader program using the 'updateLighting' function.

```
document.getElementById("ButtonXPlus").onclick = function() {
    lightPosition = vec4(1.0, 0.0, 0.0, 0.0);
    updateLighting();
```

```
};

document.getElementById("ButtonXMinus").onclick = function() {
    lightPosition = vec4(-1.0, 0.0, 0.0, 0.0);
    updateLighting();
};
```

These functions adjust the light position along the X, Y, and Z axes when the corresponding buttons are pressed, updating the scene in real-time.

1.2.4 Cube Geometry (Quad Function)

This section defines the cube's geometry. The 'quad' function generates the vertices and normals for each face of the cube. The cube has 6 faces, and each face consists of two triangles.

```
function quad(a, b, c, d) {
    var t1 = subtract(vertices[b], vertices[a]);
    var t2 = subtract(vertices[c], vertices[b]);
    var normal = cross(t1, t2);
    normal = vec3(normal);

    pointsArray.push(vertices[a]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[0]);

    pointsArray.push(vertices[b]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[1]);

    pointsArray.push(vertices[c]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[2]);

    pointsArray.push(vertices[a]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[0]);

    pointsArray.push(vertices[c]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[2]);

    pointsArray.push(vertices[d]);
    normalsArray.push(normal);
    textureArray.push(textureCoordinates[3]);
}
```

The 'quad' function computes the vertex positions, normals, and texture coordinates for each face of the cube, adding them to their respective arrays.

1.2.5 Cube Color Setup

This function uses the 'quad' function to generate all the faces of the cube. It is called in the 'colorCube' function to complete the cube's geometry.

```
function colorCube() {
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

The ‘colorCube‘ function calls ‘quad‘ for each face of the cube, ensuring that the cube is fully rendered.

1.2.6 Shader and Buffer Initialization

This segment initializes WebGL shaders and buffers, including normal, vertex, and texture coordinate buffers. It also loads the texture image for the cube.

```
window.onload = function init() {
    canvas = document.getElementById( "gl-canvas" );
    gl = canvas.getContext("webgl2");
    if ( !gl ) { alert( "WebGL isn't available" ); }

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 1.0, 1.0, 1.0 );
    gl.enable(gl.DEPTH_TEST);

    program = initShaders( gl, "vertex-shader", "fragment-shader" );
    gl.useProgram( program );

    colorCube();
    ...
}
```

1.2.7 Texture Mapping

This section sets up the texture mapping for the cube. It binds the texture, sets texture parameters, and loads the texture image into the shader.

```
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
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);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);

var image = new Image();
image.onload = function() {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE
        , image);
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.uniform1i(gl.getUniformLocation(program, "texture"), 0);
}
image.src = "umbrella.png";
```

The texture image is loaded and applied to the cube's faces. The texture is bound and sent to the shader.

1.2.8 Rendering Loop

The ‘render‘ function is the main rendering loop, which updates the rotation of the cube and re-renders it continuously.

```
var render = function(){
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    if(flag) theta[axis] += 2.0;

    modelView = mat4();
    modelView = mult(modelView, rotate(theta[xAxis], [1, 0, 0] ));
    modelView = mult(modelView, rotate(theta[yAxis], [0, 1, 0] ));
```

```
    modelView = mult(modelView, rotate(theta[zAxis], [0, 0, 1] ));

    gl.uniformMatrix4fv( gl.getUniformLocation(program,
            "modelViewMatrix"), false, flatten(modelView) );

    gl.drawArrays( gl.TRIANGLES, 0, numVertices );

    requestAnimFrame(render);
}
```

The ‘render‘ function applies the rotation transformations to the model-view matrix and draws the cube. It then requests the next animation frame to continue rendering.

1.2.9 Conclusion

This code demonstrates the rendering of a textured 3D cube in WebGL2, with interactive lighting and transformations. Users can manipulate light position and rotate the cube, providing a dynamic and engaging 3D experience.

2 Part 2 - 3D Scene with Animated Plant, Sphere and Controllable Camera

2.1 Overview

In this part, a 3D scene consisting of a sphere and an indoor plant will be created. The sphere, with a radius of 15 units, will be placed at the center of the scene, with the plant positioned on top. A directional light will be positioned above the sphere and will rotate around the y-axis, dynamically shifting the illumination on both the sphere and the plant. A camera will be used for navigation, and both the sphere and plant will rotate around the y-axis, creating an interactive and dynamic 3D environment.

2.2 Shader Setup

Two shaders are used in this project: a vertex shader and a fragment shader.

2.2.1 Vertex Shader

The vertex shader is responsible for transforming the vertices of the 3D object and passing the normal and position data to the fragment shader. It accepts vertex position and normal as inputs, performs transformations using the model-view matrix and projection matrix, and outputs the transformed normal and position.

```
#version 300 es
precision highp float;

in vec3 a_position;
in vec3 a_normal;

uniform mat4 u_mvpMatrix;

out vec3 v_normal;
out vec3 v_position;

void main() {
    gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
    v_normal = a_normal;
    v_position = a_position;
}
```

In this code, the vertex positions are transformed by the model-view-projection matrix before being passed to the fragment shader. The normals and positions are also passed for lighting calculations.

2.2.2 Fragment Shader

The fragment shader calculates the color of each fragment based on the lighting and material properties. It takes the normal and position data from the vertex shader and computes the diffuse lighting based on the light direction and surface normal.

```
#version 300 es
precision highp float;

in vec3 v_normal;
in vec3 v_position;
uniform vec3 u_lightPosition;
out vec4 fragColor;

void main() {
    vec3 normal = normalize(v_normal);
    vec3 lightDir = normalize(u_lightPosition - v_position);

    float diffuse = max(dot(normal, lightDir), 0.0);
    vec3 color = vec3(1.0, 1.0, 1.0) * diffuse;
    fragColor = vec4(color, 1.0);
}
```

3 Part 2 - WebGL2 Implementation: 3D Scene with Animated Plant and Sphere with a Controllable Camera

In this section, the WebGL2 implementation for rendering a 3D scene with a plant object on top of a sphere, along with a controllable camera that responds to mouse inputs will be analyzed. The following describes the structure and algorithms used to create this interactive 3D scene.

3.1 Canvas and WebGL2 Context Setup

The code initializes a WebGL2 context for the canvas element and ensures WebGL2 is supported by the browser. The canvas dimensions are set to match the window size, and the WebGL2 context is configured for depth testing and shader program use.

```
let canvas = document.getElementById("glCanvas");
let gl = canvas.getContext("webgl2");

if (!gl) {
    console.error("WebGL2 is not available in your browser.");
    throw new Error("WebGL2 is required.");
}
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
program = initProgram(gl, vsSource, fsSource);
if (!program){
    return;
}

gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.enable(gl.DEPTH_TEST);
gl.useProgram(program);
```

3.2 Camera and View Matrix Setup

The camera's position, zoom, rotation, and translation are controlled by user inputs. The camera is initially set at a fixed zoom and positioned along the z-axis. The camera view matrix is updated dynamically based on mouse movements, allowing for rotation, translation, and zooming effects.

```
let cameraPosition = [30, 40, cameraZoom];
let cameraRotation = [0, 0];
let cameraTranslation = [0, 0, 0];

mat4.lookAt(viewMatrix, cameraPosition, [0, 0, 0], [0, 1, 0]);
```

The `mat4.lookAt` function generates the view matrix, which positions the camera at the correct location and orientation relative to the scene. The camera is positioned using the `cameraPosition` array, and the target is set to the origin of the scene $[0, 0, 0]$ with the up vector set along the y-axis.

3.3 Loading and Parsing the OBJ Model

The 3D plant model is loaded from an OBJ file and parsed to extract the vertices, normals, textures, and faces. The vertex, normal, and texture data are then flattened and prepared for rendering by storing them in appropriate buffers.

```
const objResponse = await fetch("bitki.obj");
const objText = await objResponse.text();
const { vertices, normals, textures, faces } = parseOBJ(objText);
const { flatVertices, flatNormals, flatTextures } = flattenOBJData(
    vertices, normals, textures, faces);
```

3.4 Creating and Setting Up Buffers for the Sphere and Plant Models

The sphere is created with a radius of 15 units and subdivided into 30 segments in both latitude and longitude directions. The data for the sphere, along with the parsed plant model data, is used to set up buffers for the vertices and normals. These buffers are then used in the WebGL rendering pipeline.

```
const sphereData = createSphere(15, 30, 30);

const buffers = setupBuffers(gl, { flatVertices, flatNormals },
    sphereData);
const { plantVertexBuffer, plantNormalBuffer, sphereVertexBuffer,
    sphereNormalBuffer, sphereIndexBuffer } = buffers;
```

3.5 Projection and View Matrix Calculation

The projection and view matrices are crucial components of the rendering pipeline, defining how 3D objects are projected onto a 2D screen and how the camera views the scene. In this implementation, the projection matrix is calculated using the `mat4.perspective` function, while the view matrix is created using `mat4.lookAt`.

```
mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width / canvas.
    height, 0.1, 100.0);
mat4.lookAt(viewMatrix, cameraPosition, [0, 0, 0], [0, 1, 0]);
```

The projection matrix uses a perspective projection to simulate the effect of objects appearing smaller as they move farther away from the camera. The field of view (FoV) is set to $\pi/4$ radians (45 degrees), and the aspect ratio is dynamically calculated based on the canvas dimensions (`canvas.width / canvas.height`). The near and far clipping planes are set to 0.1 and 100.0 units, respectively, defining the depth range visible to the camera.

The view matrix is constructed using `mat4.lookAt`, which positions the camera at `cameraPosition` and orients it to look at the origin of the scene ($[0, 0, 0]$), with the up vector set to $[0, 1, 0]$, ensuring the camera remains upright.

3.6 Generating the Model-View-Projection (MVP) Matrix

The MVP matrix combines the model, view, and projection transformations into a single matrix that transforms object coordinates into clip space. The `generateMVPMatrix` function encapsulates this

process by creating a model matrix through translation and rotation, computing a normal matrix, and multiplying it with the predefined view and projection matrices to generate the final transformation.

```
function generateMVPMatrix(position, rotation) {
    const modelMatrix = mat4.create();
    mat4.identity(modelMatrix);
    mat4.translate(modelMatrix, modelMatrix, position);

    mat4.rotateY(modelMatrix, modelMatrix, rotation);

    const normalMatrix = mat4.create();
    mat4.invert(normalMatrix, modelMatrix);
    mat4.transpose(normalMatrix, normalMatrix);

    const mvpMatrix = mat4.create();
    mat4.multiply(mvpMatrix, projectionMatrix, viewMatrix);
    mat4.multiply(mvpMatrix, mvpMatrix, modelMatrix);

    return mvpMatrix;
}
```

The function begins by creating and initializing the `modelMatrix` to an identity matrix. It then applies a translation to position the object at the specified coordinates. Then, it applies a rotation around the y-axis using `mat4.rotateY`.

A normal matrix is also generated for transforming normal vectors during lighting calculations. This matrix is derived by inverting and transposing the `modelMatrix`.

Finally, the `mvpMatrix` is computed by multiplying the projection matrix, view matrix, and model matrix in sequence. This matrix transforms object coordinates from model space (local object coordinates) to clip space, preparing them for rendering on the screen.

The `generateMVPMatrix` function ensures that transformations are consistently applied across all objects, allowing for flexible positioning and rotation within the 3D scene.

3.7 Setting Up Buffers

The `setupBuffers` function is responsible for creating and binding buffers for the plant and sphere models. It initializes the vertex and normal buffers for both the plant and the sphere. Additionally, it creates an index buffer for the sphere model, which is essential for rendering with indexed drawing. The buffers are then returned for use in the rendering process.

```
function setupBuffers(gl, plantData, sphereData) {
    const { flatVertices, flatNormals } = plantData;
    const { vertices: sphereVertices, normals: sphereNormals, indices:
        sphereIndices } = sphereData;

    const plantVertexBuffer = initBuffer(gl, flatVertices);
    const plantNormalBuffer = initBuffer(gl, flatNormals);

    const sphereVertexBuffer = initBuffer(gl, sphereVertices);
    const sphereNormalBuffer = initBuffer(gl, sphereNormals);

    const sphereIndexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, sphereIndexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(
        sphereIndices), gl.STATIC_DRAW);

    return {
        plantVertexBuffer,
        plantNormalBuffer,
        sphereVertexBuffer,
        sphereNormalBuffer,
        sphereIndexBuffer,
```

```
    };
}
```

3.8 Flattening OBJ Data

The `flattenOBJData` function processes the data from an OBJ file by flattening the vertices, normals, and texture coordinates based on the faces of the model. It iterates over each face, extracting the corresponding vertex, normal, and texture information and consolidates them into flattened arrays. These arrays are then returned for use in WebGL buffers.

```
function flattenOBJData(vertices, normals, textures, faces) {
    const flatVertices = [];
    const flatNormals = [];
    const flatTextures = [];

    for (let i = 0; i < faces.length; i++) {
        const face = faces[i];
        for (let j = 0; j < face.length; j++) {
            const vertexIndex = face[j].vertex;
            const normalIndex = face[j].normal;
            const textureIndex = face[j].texture;

            const vertex = vertices.slice(vertexIndex * 3, vertexIndex
                * 3 + 3);
            const normal = normals.slice(normalIndex * 3, normalIndex
                * 3 + 3);
            const texture = textures.slice(textureIndex * 2,
                textureIndex * 2 + 2);

            flatVertices.push(...vertex);
            flatNormals.push(...normal);
            flatTextures.push(...texture);
        }
    }

    return { flatVertices, flatNormals, flatTextures };
}
```

3.9 Rendering the Sphere

The `renderSphere` function is used to render the sphere model using the buffers created earlier. It binds the vertex and normal buffers, sets the attribute pointers, and enables them for use in the vertex shader. It also binds the index buffer for indexed rendering and computes the Model-View-Projection (MVP) matrix for the sphere's position and rotation. The MVP matrix is passed to the shader program, and the sphere is drawn using the `drawElements` method.

```
function renderSphere(vertexBuffer, normalBuffer, indexBuffer,
    position, rotation, mvpLocation, indexCount) {
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false, 0, 0)
        ;
    gl.enableVertexAttribArray(positionLocation);

    gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
    gl.vertexAttribPointer(normalLocation, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(normalLocation);

    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);

    const mvpMatrix = generateMVPMatrix(position, rotation);
    gl.uniformMatrix4fv(mvpLocation, false, mvpMatrix);
```

```
    gl.drawElements(gl.TRIANGLES, indexCount, gl.UNSIGNED_SHORT, 0);
}
```

3.10 Rendering the Plant

The `renderPlant` function is responsible for rendering the plant model in the scene. It binds the vertex and normal buffers and enables the respective attribute arrays for use in the vertex shader. The function computes the Model-View-Projection (MVP) matrix based on the plant's position and rotation, then passes this matrix to the shader program. Finally, the plant is drawn using the `drawArrays` method, which renders the plant's triangles.

```
function renderPlant(vertexBuffer, normalBuffer, position, rotation,
    mvpLocation) {
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false, 0, 0)
        ;
    gl.enableVertexAttribArray(positionLocation);

    gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
    gl.vertexAttribPointer(normalLocation, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(normalLocation);

    const mvpMatrix = generateMVPMatrix(position, rotation);
    gl.uniformMatrix4fv(mvpLocation, false, mvpMatrix);
    gl.drawArrays(gl.TRIANGLES, 0, flatVertices.length / 3);
}
```

3.11 Rendering the Scene

The `render` function is the main rendering loop that continuously updates the scene. It begins by clearing the color and depth buffers to prepare for the new frame. The function then calculates the projection and view matrices to set up the camera. The light's position is dynamically updated based on the current time, making it rotate around the scene. The `renderSphere` and `renderPlant` functions are called to draw the sphere and plant at their respective positions, with each rendered frame updating the scene. The rotation value is also incremented, causing the sphere and plant to rotate.

```
function render(time) {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width /
        canvas.height, 0.1, 100.0);
    mat4.lookAt(viewMatrix, cameraPosition, [0, 0, 0], [0, 1, 0]);

    const lightRadius = 20.0;
    const lightAngle = time * 0.001;
    const lightPosition = [
        lightRadius * Math.sin(lightAngle),
        5.0,
        lightRadius * Math.cos(lightAngle)
    ];

    gl.uniform3fv(lightPositionLocation, lightPosition);
    renderSphere(
        sphereVertexBuffer, sphereNormalBuffer, sphereIndexBuffer,
        [0, 0, 0], rotation, mvpLocation, sphereData.indices.length
    );
    renderPlant(plantVertexBuffer, plantNormalBuffer, [0, 15, 0],
        rotation, mvpLocation);
    rotation += 0.01;
    requestAnimationFrame(render);
}
```

3.12 Creating the Sphere

The `createSphere` function generates the vertex data, normals, and indices for a sphere based on the given radius, latitude, and longitude values. The function calculates the positions of vertices in spherical coordinates, converts them into Cartesian coordinates, and then calculates the corresponding normals. The indices are generated to form the triangles that make up the surface of the sphere.

```
function createSphere(radius, latitude, longitude) {
    const vertices = [];
    const normals = [];
    const indices = [];

    for (let i = 0; i <= latitude; i++) {
        const theta = i * Math.PI / latitude;
        const sin = Math.sin(theta);
        const cos = Math.cos(theta);

        for (let j = 0; j <= longitude; j++) {
            const phi = j * 2 * Math.PI / longitude;
            const s = Math.sin(phi);
            const c = Math.cos(phi);

            const x = c * sin;
            const y = cos;
            const z = s * sin;

            normals.push(x, y, z);
            vertices.push(radius * x, radius * y, radius * z);
        }
    }

    for (let i = 0; i < latitude; i++) {
        for (let j = 0; j < longitude; j++) {
            const f = (i * (longitude + 1)) + j;
            const s = f + longitude + 1;

            indices.push(f, s, f + 1);
            indices.push(s, s + 1, f + 1);
        }
    }

    return { vertices, normals, indices };
}
```

3.13 Parsing the OBJ Model

The `parseOBJ` function processes an OBJ file in textual format and extracts the vertices, normals, textures, and faces. It reads each line of the OBJ file, identifies the type of data (vertices, normals, etc.), and parses the corresponding values. The faces are handled specially to account for the vertex, texture, and normal indices, ensuring the model can be properly rendered. This function returns the parsed data for later use in creating the model buffers.

```
function parseOBJ(objText) {
    const vertices = [];
    const normals = [];
    const textures = [];
    const faces = [];

    const lines = objText.split("\n");
    for (let i = 0; i < lines.length; i++) {
        const line = lines[i].trim();
        if (line === "") {
            continue;
```

```
        }
        const parts = line.split(" ");
        const type = parts[0];
        const data = parts.slice(1);

        if (type === "v") {
            for (let j = 0; j < data.length; j++) {
                vertices.push(parseFloat(data[j]));
            }
        } else if (type === "vn") {
            for (let j = 0; j < data.length; j++) {
                normals.push(parseFloat(data[j]));
            }
        } else if (type === "vt") {
            for (let j = 0; j < data.length; j++) {
                textures.push(parseFloat(data[j]));
            }
        } else if (type === "f") {
            const face = [];
            for (let j = 0; j < data.length; j++) {
                const indices = data[j].split("/");
                face.push({
                    vertex: parseInt(indices[0], 10) - 1,
                    texture: indices[1] ? parseInt(indices[1], 10) - 1
                        : null,
                    normal: indices[2] ? parseInt(indices[2], 10) - 1
                        : null
                });
            }
            if (face.length === 4) {
                faces.push([face[0], face[1], face[2]]);
                faces.push([face[0], face[2], face[3]]);
            }
        }
    }

    return { vertices, normals, textures, faces };
}
```

3.14 Handling Mouse Inputs for Camera Control

Mouse events are used to control the camera's rotation, zoom, and translation:

- **Left Mouse Button**: Rotates the camera around the center of the scene.
- **Middle Mouse Button**: Zooms the camera in and out.
- **Right Mouse Button**: Moves the camera laterally (both horizontally and vertically).

The `onMouseMove` function handles these actions by calculating the change in mouse position and adjusting the camera's properties accordingly.

```
function onMouseMove(event) {
    if (!isMouseDown) return;

    const [deltaX, deltaY] = [event.clientX - lastMouseX, event.
        clientY - lastMouseY];

    switch (mouseButton) {
        case 0:
            cameraRotation[0] += deltaX * 0.01;
            cameraRotation[1] = Math.max(-Math.PI / 2, Math.min(Math.
                PI / 2, cameraRotation[1] + deltaY * 0.01));
            break;
```

```
            case 1:
                cameraZoom = Math.max(1, Math.min(50, cameraZoom - deltaY
                    * 0.01));
                break;
            case 2:
                cameraTranslation[0] += deltaX * 0.01;
                cameraTranslation[1] -= deltaY * 0.01;
                break;
        }

        [lastMouseX, lastMouseY] = [event.clientX, event.clientY];
}
```

The mouse events are captured and processed to adjust the camera's position and orientation. The left mouse button controls the rotation, the middle mouse button handles zooming, and the right mouse button moves the camera horizontally and vertically.

3.15 Canvas Resizing and Aspect Ratio Adjustment

When the window is resized, the canvas dimensions are updated, and the camera's projection matrix is recalculated to maintain the correct aspect ratio and prevent distortion of the 3D objects. The `resizeCanvas` function handles these updates.

```
function resizeCanvas() {
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;
    gl.viewport(0, 0, canvas.width, canvas.height);
    mat4.perspective(projectionMatrix, Math.PI / 4, canvas.width /
        canvas.height, 0.1, 100.0);
}
```

This function adjusts the canvas size whenever the window is resized, recalculating the aspect ratio of the projection matrix to ensure the 3D scene is displayed correctly.

3.16 Dynamic View Matrix Updates

The `updateViewMatrix` function dynamically recalculates the view matrix to reflect changes in the camera's position, rotation, and zoom level. This ensures that user interactions, such as mouse movements, immediately update the camera's perspective in the 3D scene.

```
function updateViewMatrix() {
    const forward = [
        Math.sin(cameraRotation[0]) * Math.cos(cameraRotation[1]),
        Math.sin(cameraRotation[1]),
        Math.cos(cameraRotation[0]) * Math.cos(cameraRotation[1]),
    ];

    const target = [0, 0, 0];
    const up = [0, 1, 0];

    cameraPosition = [
        target[0] + forward[0] * cameraZoom + cameraTranslation[0],
        target[1] + forward[1] * cameraZoom + cameraTranslation[1],
        target[2] + forward[2] * cameraZoom,
    ];

    mat4.lookAt(viewMatrix, cameraPosition, target, up);
}
```

The `forward` vector calculates the direction the camera is facing based on its rotation angles (`cameraRotation[0]` and `cameraRotation[1]`), which represent yaw and pitch. Using trigonometric functions, it determines the forward-facing direction in 3D space.

The `cameraPosition` is then updated by adding the `forward` vector, scaled by the `cameraZoom`, to the camera translation offsets. This computation allows the camera to zoom in and out and move within the scene.

Finally, the `mat4.lookAt` function updates the view matrix, positioning the camera at `cameraPosition`, looking at the `target` (the origin of the scene), with the `up` vector ensuring proper orientation. This dynamic recalculation enables real-time interactivity in the 3D scene.

3.17 Conclusion

This section demonstrates how to load, parse, and render 3D models using WebGL2. The sphere and plant models are generated and displayed with dynamic lighting and rotation effects. The `createSphere` function efficiently generates the vertex, normal, and index data for the sphere, while the `parseOBJ` function processes the OBJ file for the plant model. Buffers are set up for efficient rendering, and the models are animated with real-time transformations. The interaction of the dynamic light source, which rotates around the scene, highlights the 3D nature of the objects, enhancing the visual experience.

3.18 Guidance: Setting Up and Running the Project

This subsection provides step-by-step instructions for setting up and running the project in a local development environment. The project structure and the use of Python's http.server module to serve files are detailed below.

3.18.1 Project Structure

Ensure the project files are organized as follows:

- index.html: The main HTML file for the project.
- resources/bitki.obj: The 3D object file used for rendering the plant object.
- `js/`: A folder containing all JavaScript files, including the code for rendering, animation, and camera control.

3.18.2 Serving the Files Locally

To run the project in a browser, you need to serve the files using a local HTTP server. Python's built-in `http.server` module is a simple and efficient way to achieve this. Follow these steps:

1. Open a terminal or command prompt.
2. Navigate to the root directory of your project, where the `index.html` file is located. Use the `cd` command, for example:

   ```
   cd path/to/project
   ```

3. Start a local HTTP server using Python. Run the following command:

   ```
   python -m http.server 8000
   ```

   This starts a server on port `8000`. You can replace `8000` with any other available port number if desired.

4. Open your web browser and navigate to:

   ```
   http://localhost:8000
   ```

   This will load the `index.html` file and initialize the WebGL2 3D scene.

By following these steps, you can successfully set up and run the project in your local environment. If you run into some problems, try pressing Ctrl+F5 to clear the browser cache and rerun the open tab.

4 Conclusion

This assignment demonstrated key concepts of 3D rendering using WebGL2, progressing from basic to more advanced features through two distinct parts.

4.1 Part 1: Cube with Texture Mapping

In Part 1, we rendered a textured 3D cube, introducing the application of texture mapping in WebGL2. This part explored loading and applying textures to a 3D object, managing buffers for vertices and normals, and transforming the cube for dynamic viewing. User interaction through mouse events allowed the cube to be manipulated in real-time, providing a more engaging visualization.

4.2 Part 2: Animated Plant and Rotating Sphere

Part 2 extended the previous work by rendering a dynamic scene with a sphere and a plant model. The sphere was created, while the plant was loaded from an OBJ file and rendered on top of the sphere. The scene featured a rotating directional light, which dynamically changed the lighting effects on the models. Additionally, the plant and sphere rotated around their respective axes, with the camera providing interactive control, offering a more immersive experience of the 3D environment.

References

1. https://github.com/esangel/WebGL
2. https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect
3. https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener
4. https://webgl2fundamentals.org/
5. https://docs.python.org/3/library/http.server.html
6. https://en.wikipedia.org/wiki/Wavefront_.obj_file
7. https://dev.to/ndesmic/webgl-3d-engine-from-scratch-part-6-procedural-sphere-generation-2