BBM414 Computer Graphics Lab.
Programming Assignment 5 # - 2024

Ahmet Uman
2210356129
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b2210356129@cs.hacettepe.edu.tr

Overview

The purpose of this assignment is get familiar with textures and lighting in WebGL2. We will explore two different methods for implementing textures and use directional lighting to illuminate your scene effectively.

1 Part 1

Within the scope of this part, I created a 3D cube model using WebGL and performed two main tasks:

- Texturing the cube: A texture image in PNG format was applied to the cube's surfaces. - Changing the position of the light: Buttons were added to change the position of the light to increase user interaction.

Our vertex shader processes each vertex of the cube and transforms their positions to make them suitable for the projection matrix. It also processes their normal vectors, which form the basis of lighting calculations. Phong Lighting Model was used as the lighting model. This model is based on three main components:

- Ambient: Represents global illumination.
- Diffuse: Provides shading of the surface according to the light source.
- Specular: Defines the bright areas of the surface.

Vertex Shader calculates the lighting effects and then passes this information to Fragment Shader.

```
<script id="vertex-shader" type="x-shader/x-vertex">#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);
    }

    fColor = ambient + diffuse + specular;
    fColor.a = 1.0;
    fTexCoord = vTexCoord;

    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
}
```

Then, fragment shader calculates the final color for each pixel. In this project, the color of the texture applied to the cube is combined with the lighting effects.

To apply a texture map to each surface, texture coordinates were added to the texCoordsArray array. The texture was loaded with the configureTexture function. The PNG file was used to transfer the texture to WebGL:

```
function configureTexture(image) {
    var texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
    gl.generateMipmap(gl.TEXTURE_2D);
}
```

For user interaction, a function was defined that changes the light position and prints into screen when the buttons are pressed.

```
    function updateLightPosition() {
        gl.uniform4fv(gl.getUniformLocation(program, "lightPosition"),
            flatten(lightPosition));

        const lightDisplay = document.getElementById("light-position-display");
        lightDisplay.textContent = `Light Position:
            (${lightPosition[0].toFixed(2)}, ${lightPosition[1].toFixed(2)},
            ${lightPosition[2].toFixed(2)})`;
    }
```

2 Part 2

In this project, we will apply sand textures (albedo, ambient occlusion, height, metallic, normal and roughness) to a sphere (like a planet) surface with a physically based shading (PBR) approach and

also loaded a plant model (.obj + .mtl) to show how we positioned and lit the scene. All tasks, their explanations and how they were done are explained in detail in the subheadings below.

2.1 Applying Sand Textures and Physically Based (PBR) Shading on Sphere

In the project, a sphere geometry was created and different texture maps were used to give this sphere a "Mars-like" surface. These textures and their explanations are:

1. *Albedo*: The basic color of the surface
2. *Metallic*: Whether the surface is metallic or dielectric (0.0 → insulator, 1.0 → metal)
3. *Roughness*: How rough or shiny the surface is
4. *Normal*: Micro-scale curvatures of the surface in light calculations
5. *AO (Ambient Occlusion)*: Darkening of the environment

I placed these texture maps on the sphere by projecting them using the planar projection mapping method. Planar projection mapping actually determines how the texture coordinates taken from the plane will be mapped to the UV coordinates of the sphere. When creating the sphere in the code, UVs are calculated according to the x-y-z positions in each (lat, lon) band. It is possible to blend these UV coordinates with the planning approach, and it is also a typical method to produce spherical UV coordinates directly during the sphere creation phase.

```
function createSphere(radius, latBands, lonBands) {
    const positions = [];
    const normals = [];
    const texcoords = [];
    const indices = [];

    for(let lat=0; lat<=latBands; lat++){
        const theta = lat*Math.PI/latBands;
        const sinT = Math.sin(theta);
        const cosT = Math.cos(theta);

        for(let lon=0; lon<=lonBands; lon++){
            const phi = lon*2.0*Math.PI/lonBands;
            const sinP= Math.sin(phi);
            const cosP= Math.cos(phi);

            const x = cosP * sinT;
            const y = cosT;
            const z = sinP * sinT;
            const u = 1 - (lon / lonBands);
            const v = 1 - (lat / latBands);

            positions.push(radius*x, radius*y, radius*z);
            normals.push(x, y, z);
            texcoords.push(u,v);
        }
    }

    for(let lat=0; lat<latBands; lat++){
        for(let lon=0; lon<lonBands; lon++){
            const first = lat*(lonBands+1)+lon;
            const second= first + lonBands+1;
            indices.push(first, second, first+1);
            indices.push(second, second+1, first+1);
        }
    }

    return { positions, normals, texcoords, indices };
}
```

2.2 Reading Plant Model and Material File (.mtl) and Placing It on the Sphere

The second object in the scene is a plant model and is loaded via .obj and .mtl files:

.obj file: Geometry (position, texture coordinate, normal) information is read from v, vt and vn lines and transferred to WebGL buffers.

.mtl file: File paths such as diffuse/color texture from mapKd line, normal map from mapbump (or bump) line are determined, and real images (e.g. .jpg or .png) are loaded into WebGL with these paths.

When this process is completed, the plant model is covered with the textures specified in the .mtl file. Texture samplers (e.g. albedo, normal) are assigned to the appropriate uniform sampler2D variables in the shader and the plant's surface is prepared.

Also, in order to apply the textures here, I changed the paths in the .mtl source file provided to us and used them. Also, since I was getting an error because of the spaces in between, I updated the file names as follows:

```
map_Kd textures/indoor_plant_2_COL.jpg
map_Bump textures/indoor_plant_2_NOR.jpg
map_Ks textures/indoor_plant_2_COL.jpg
```

I'm not sure if this error was intentionally placed for us to fix, but after making these changes, it was fixed.

2.3 Placing Directional Light and Illuminating the Sphere

The project uses the concept of directional light. Here, the light is assumed to come from an infinite source in a fixed direction:

uLightDirection is defined in the shader (Details for Shaders subsection).

Unlike point light, the light vector going to the surface point is calculated as -uLightDirection.

This light vector is included in the diffuse and specular calculation in the PBR shader.

The sphere and the plant are shaded and illuminated according to the light direction in the task. For example, when the light is adjusted from top to bottom, the upper part of the sphere is illuminated brighter while the lower parts remain relatively dark. In this way, realistic lighting is provided in the scene.

2.4 Rotating the Globe and Plant on the Y Axis

In order to animate the scene, both the sphere and the plant model are rotated around the Y axis. To do this, a certain angle is calculated in each animation frame (requestAnimationFrame):

Sphere Model Matrix: rotated with rotateY.

Plant Model Matrix: rotated on the Y axis by the same angle or another angle. Then the plant model is moved up so that it is above the center of the sphere and scaled if necessary.

This achieves a dynamic movement of the scene. Both the sand-covered sphere and the plant slowly rotate on their own axes.

```
function updateObjectRotations() {
    let angle = performance.now() * 0.0005;

    mat4.identity(sphereModel);
    mat4.rotateY(sphereModel, sphereModel, angle);

    mat4.identity(plantModel);
    mat4.rotateY(plantModel, plantModel, angle);
```

```
    mat4.translate(plantModel, plantModel, [0,15,0]);
    mat4.scale(plantModel, plantModel, [2,2,2]);
}
```

2.5 Mat4 Functions and Camera Controls

In order to examine the planet and the plant from different angles, a camera system had to be built. For this, I used the controllable camera code (slightly updated) from the previous assignment:

```
canvas.addEventListener('mousedown',(e)=>{
    mouseDown=true;
    mouseButtons=e.buttons;
    lastX=e.clientX; lastY=e.clientY;
    e.preventDefault();
});
canvas.addEventListener('mousemove',(e)=>{
    if(!mouseDown) return;
    const dx = e.clientX - lastX;
    const dy = e.clientY - lastY;

    if((mouseButtons & 1)===1){
        // Left = rotate
        cameraYaw  += dx*0.01;
        cameraPitch += dy*0.01;
        cameraPitch = Math.max(-Math.PI/2+0.1, Math.min(Math.PI/2-0.1,
            cameraPitch));
    } else if((mouseButtons & 4)===4){
        // Middle = zoom
        cameraDist += dy*0.2;
        cameraDist = Math.max(2, cameraDist);
    } else if((mouseButtons & 2)===2){
        // Right = pan
        cameraPanX += -dx*0.05;
        cameraPanY += dy*0.05;
    }
    updateViewMatrix();
    lastX=e.clientX; lastY=e.clientY;
});
canvas.addEventListener('mouseup',(e)=>{
    mouseDown=false;
    mouseButtons=0;
});
canvas.addEventListener('mouseout',(e)=>{
    mouseDown=false;
    mouseButtons=0;
});
```

In various parts of the project, I felt the need to get help from mat4 and vec4 functions. However, since I could not use them, I created my own mat4.js file and defined many functions such as rotation, transformation, scaling in this file. Of course, I perform these operations using matrix multiplications. Although I did not use it in the final version of the project, the vec4 definition and functions are also in this file.

2.6 Shaders

One of the most critical components of our project is shader programs.

- **Vertex Shader** Vertex shader performs the following tasks:
    1. **Model, View, Projection Transformations:** It allows the transformation of a vertex from local model coordinates to world coordinates, and from there to view and

projection space. In our project, for example, the multiplications

$$\mathbf{uModelMatrix} \times \mathbf{uViewMatrix} \times \mathbf{uProjectionMatrix}$$

are used.

2. **Transformation of Normal Vectors to World Space:** Rotation of the surface normal (e.g. $\mathbf{aNormal}$) with respect to the model matrix or reorientation with `mat3` multiplication. This normal is necessary for determining which direction the surface is facing in the light calculation.
3. **Transferring Texture Coordinates (UV):** The vertex shader transfers the texture coordinates associated with the vertex (e.g. $\mathbf{aTexCoord}$) as "varying" variables to the fragment shader.
4. **Calculating World Position:** The position of a vertex in world space is calculated as $\mathbf{vPosition}$ and passed to the fragment shader as *varying*. Comparing $\mathbf{vPosition}$ with the position of the camera or light plays an important role in light calculations.

- **Fragment Shader**

  The actual physically based shading process in our project is done here. The fragment shader achieves these processes:

  1. **Texture Sampling:** Albedo (color), Normal (surface microdetail orientation), Metallic, Roughness and Ambient Occlusion maps are read one by one. The values obtained from each map define the physical properties of the material.
  2. **Determining Light Vectors:**
     If *directional light* is used in the project, it comes from the $\mathbf{uLightDirection}$ uniform. The camera vector is also found by normalizing between $\mathbf{uCameraPosition}$ and $\mathbf{vPosition}$.
  3. **PBR Calculation Steps:**
     - *NDF (Normal Distribution Function)*: Represents the light distribution of the microfacet surface.
     - *Geometry (G) Factor*: Shows the efficiency of the intersection of the light and view vector with the surface.
     - *Fresnel (F) Equation*: Determines the amount of reflection due to the angle of view of the light to the surface
     - *kD and kS* mixture: Mixture of diffuse and specular components; diffuse contribution decreases on metallic surfaces.

2.7 Notes For An Error-Free Operation of Project

Resources and textures files were not uploaded due to file size. The file structure I used while designing the project:

```
Project/
 index.html
 js/
    app.js
    mat4.js
    initialize.js
    shaders.js
 resources/
    bitki.obj
    bitki.mtl
 textures/
    sand-dunes1_albedo.png
    sand-dunes1_normal-dx.png
```

```
sand-dunes1_roughness.png
sand-dunes1_metallic.png
sand-dunes1_ao.png
indoor_plant_2_COL.jpg
indoor_plant_2_NOR.jpg
```