BBM414 Assignment-4 WebGL2 Model Drawing and Viewing

b2220356044 Muzaffer Berke Savas

15/12/2024

1 Object Loading and Buffer Management

1.1 Object Loading

Below function reads the ".obj" file line by line and parse each line with respect to It's identifiers like 'v, vn, f'.

```
async function loadObj(fileUrl) {
    const response = await fetch(fileUrl);
    const text = await response.text();
    const vertices = [];
    const normals = [];
    const indices = [];

    const lines = text.split("\n");
    for (const line of lines) {
        if (line.startsWith("v ")) {
            const [, x, y, z] = line.split(" ");
            vertices.push(parseFloat(x), parseFloat(y),
               parseFloat(z));
        } else if (line.startsWith("vn ")) {
            const [, nx, ny, nz] = line.split(" ");
            normals.push(parseFloat(nx), parseFloat(ny),
               parseFloat(nz));
        } else if (line.startsWith("f ")) {
            const [, ...faceIndices] = line.split(" ");
            for (let i = 1; i < faceIndices.length - 1; i++) {
                indices.push(
                    parseInt(faceIndices[0].split("//")[0]) - 1,
                    parseInt(faceIndices[i].split("//")[0]) - 1,
                    parseInt(faceIndices[i + 1].split("//")[0]) -
                        1
                );
            }
        }
    }
    return { vertices, normals, indices };
}
```

Listing 1: Loading the Object

1.2 Buffer Initialization

Binding the object vertices, normals and indices that we read above part.

```
const positionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(objData.vertices)
   , gl.STATIC_DRAW);

const normalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(objData.normals),
    gl.STATIC_DRAW);

const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(objData.
   indices), gl.STATIC_DRAW);
```

Listing 2: Buffer Initialization

2 Camera and User Interaction

User interaction with the 3D scene is implemented through mouse events, allowing for camera control such as rotation and zooming:

```
canvas.addEventListener("mousedown", (event) => {
    isDragging = true;
    lastMouseX = event.clientX;
    lastMouseY = event.clientY;
});

canvas.addEventListener("mouseup", () => {
    isDragging = false;
});

canvas.addEventListener("mousemove", (event) => {
    if (isDragging) {
        const deltaX = event.clientX - lastMouseX;
        const deltaY = event.clientY - lastMouseY;

        if (event.button === 0) {
            cameraRotation[1] -= deltaX * 0.01;
            cameraRotation[0] -= deltaY * 0.01;
            cameraRotation[0] = Math.max(-Math.PI / 2, Math.min(
               Math.PI / 2, cameraRotation[0]));
        } else if (event.button === 1) {
            cameraZoom -= deltaY * 0.01;
            cameraZoom = Math.max(0.1, cameraZoom);
        } else if (event.button === 2) {
            cameraPosition[0] -= deltaX * 0.01;
            cameraPosition[1] += deltaY * 0.01;
        }

        lastMouseX = event.clientX;
        lastMouseY = event.clientY;
    }
});
```

Listing 3: Mouse Event Handlers

3 Rendering Loop

Updating camera and model transfromations as necessary.

```
function render() {
    rotationAngle += 0.01;
    distance += 0.02;
    upDown += 0.01;

    mat4.identity(viewMatrix);
    mat4.rotateX(viewMatrix, viewMatrix, cameraRotation[0]);
    mat4.rotateY(viewMatrix, viewMatrix, cameraRotation[1]);
    mat4.translate(viewMatrix, viewMatrix, cameraPosition);
    mat4.scale(viewMatrix, viewMatrix, [cameraZoom, cameraZoom,
       cameraZoom]);

    mat4.perspective(
        projectionMatrix,
        Math.PI / 4,
        canvas.width / canvas.height,
        0.1,
        100
    );

    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.enable(gl.DEPTH_TEST);
    gl.useProgram(program);
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    gl.enableVertexAttribArray(positionLocation);
    gl.vertexAttribPointer(positionLocation, 3, gl.FLOAT, false,
       0, 0);

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

    for (let i = 0; i < modelMatrices.length; i++) {
        mat4.multiply(modelViewProjectionMatrix, projectionMatrix
           , viewMatrix);
        mat4.multiply(modelViewProjectionMatrix,
           modelViewProjectionMatrix, modelMatrices[i]);
        gl.uniformMatrix4fv(mvpMatrixLocation, false,
           modelViewProjectionMatrix);
        gl.drawElements(gl.TRIANGLES, objData.indices.length, gl.
           UNSIGNED_SHORT, 0);
    }
    requestAnimationFrame(render);
}
```

Listing 4: Rendering Loop

4 Conclusion

This WebGL2 program demonstrates how to create a dynamic 3D rendering application using shaders, matrices, and user interaction. The use of multiple objects, lighting, and camera controls creates an engaging and realistic experience.

5 References

* https://cs418.cs.illinois.edu/website/text/obj.html

* https://webgl2fundamentals.org/webgl/lessons/webgl-3d-camera.html

* https://glmatrix.net