BBM414 Computer Graphics Lab.
Assignment 4

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

Overview

This project involves creating a 3D scene with interactive camera controls. In Part 1, a cube is rendered, and its view is updated using event listeners. Part 2 expands this by loading and animating three monkey heads with different motions, while implementing a controllable camera. The project focuses on learning 3D object manipulation, camera navigation, and responsiveness to user input and window resizing.

1 Part 1 - WebGL2 Cube with Camera Control and Shaders

This section analyzes the WebGL2 code used to render a 3D cube with interactive camera control, utilizing shaders and transformations. The code includes functions for setting up the 3D environment, handling user inputs, and rendering the cube with basic camera controls.

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 passing color data to the fragment shader. It accepts vertex position and color as inputs, performs transformations using the model-view matrix and projection matrix, and outputs the transformed color.

```
#version 300 es
in  vec4 vPosition;
in  vec4 vColor;
out vec4 fColor;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
    fColor = vColor;
}
```

In this code, the vertex positions are transformed by the combined model-view and projection matrices before being passed to the fragment shader.

1.1.2 Fragment Shader

The fragment shader simply assigns the color passed from the vertex shader to the final fragment color for rendering.

```
#version 300 es
precision highp float;
in vec4 fColor;
out vec4 fragColor;

void main()
{
    fragColor = fColor;
}
```

1.2 Cube Vertices and Colors

The cube is defined using 8 vertices and 6 faces (each consisting of two triangles), resulting in 36 vertices in total. The colors are specified as RGBA vectors.

```
var vertices = [
    vec4(-0.5, -0.5,  1.5, 1.0),
    vec4(-0.5,  0.5,  1.5, 1.0),
    vec4(0.5,  0.5,  1.5, 1.0),
    vec4(0.5, -0.5,  1.5, 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 vertexColors = [
    vec4( 0.0, 0.0, 0.0, 1.0 ),  // black
    vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
    vec4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
    vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    vec4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    vec4( 0.0, 1.0, 1.0, 1.0 ),  // cyan
    vec4( 1.0, 1.0, 1.0, 1.0 )   // white
];
```

1.3 Cube Construction

The `colorCube()` function constructs the cube by calling the `quad()` function multiple times, which defines the faces of the cube by combining sets of 4 vertices. Each face is composed of two triangles, with each vertex assigned the color of the corresponding corner.

```
function quad(a, b, c, d) {
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[a]);
    pointsArray.push(vertices[b]);
    colorsArray.push(vertexColors[a]);
    pointsArray.push(vertices[c]);
    colorsArray.push(vertexColors[a]);
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[a]);
    pointsArray.push(vertices[c]);
    colorsArray.push(vertexColors[a]);
    pointsArray.push(vertices[d]);
    colorsArray.push(vertexColors[a]);
}
```

1.4 Camera Setup and Mouse Controls

The camera in the scene is controlled by mouse inputs. The camera's position is updated based on the `theta` and `phi` angles, which are modified using the mouse movements. The function `handleMouseMove()` handles the mouse events, adjusting the camera's angle based on the movement.

```
function handleMouseMove(event) {
    if (!isPointerLocked) return;
    theta -= event.movementY * sensitivity;
    phi -= event.movementX * sensitivity;
}
```

1.5 Rendering and Animation

The `render()` function is called in a loop to continuously update and render the scene. It clears the canvas, updates the camera matrices, and draws the cube using the `gl.drawArrays()` function. The camera is updated each frame to allow for smooth user interaction.

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

    eye = vec3(radius*Math.sin(theta)*Math.cos(phi),
        radius*Math.sin(theta)*Math.sin(phi), radius*Math.cos(theta));
    modelViewMatrix = lookAt(eye, at , up);
    projectionMatrix = perspective(fovy, aspect, near, far);

    gl.uniformMatrix4fv( modelViewMatrixLoc, false, flatten(
        modelViewMatrix) );
    gl.uniformMatrix4fv( projectionMatrixLoc, false, flatten(
        projectionMatrix) );

    gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
    requestAnimFrame(render);
}
```

1.6 Pointer Lock and User Interaction

Pointer lock is used to capture the mouse cursor for more intuitive camera control. The mouse events and keyboard events are set up to toggle pointer lock and handle mouse movements for controlling the camera.

```
canvas.addEventListener('click', function() {
    if (checkPointerLockStatus()) {
        disablePointerLock();
    } else {
        enablePointerLock();
    }
});
```

This ensures that the camera movement is fluid and that the user can interact with the 3D scene without being distracted by the mouse cursor.

1.7 Conclusion

This code demonstrates the use of WebGL2 for rendering a 3D cube, implementing interactive camera control, and applying transformations using shaders. It also allows user interaction with the 3D scene through mouse movements, providing an immersive experience for navigating the cube.

2 Part 2 - 3D Scene with Animated Monkeys and Controllable Camera

This section analyzes the WebGL2 code used to render a 3D scene with three animated monkey heads and a controllable camera. The code includes functions for setting up the 3D environment, handling user inputs for camera control, and animating the monkeys in the scene.

2.1 Shader Setup

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

2.1.1 Vertex Shader

The vertex shader is responsible for transforming the vertices of the 3D objects (the monkey heads) 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.

2.1.2 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);
}
```

The fragment shader calculates the diffuse lighting based on the surface normal and light direction. The final color is determined by the lighting effect and is output as the fragment color.

3 Part 2 - WebGL2 Implementation: 3D Scene with Animated Monkeys and Controllable Camera

In this section, we detail the WebGL2 implementation for rendering a 3D scene with three animated monkey heads, along with a controllable camera that responds to mouse inputs. 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 = [0, 0, 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 monkey head 3D model is loaded from an OBJ file, parsed, and its vertices and normals are extracted for use in the rendering process. The model is split into vertex data and normal data, which are then flattened and stored in buffers.

```
const response = await fetch("monkey_head.obj");
const objText = await response.text();
const { vertices, normals, faces } = parseOBJ(objText);
```

The `parseOBJ` function parses the OBJ file format to extract vertex positions and normal vectors, which are then used to create the model in 3D space. The model's vertex data is flattened into a single array to be passed to WebGL buffers for rendering.

3.4 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.

3.5 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 optional 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);

    if (rotation) {
        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. If a rotation value is provided, 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.

3.6 Animation Logic for the Monkeys

The animation logic defines the behavior of each of the three monkeys. The monkeys are animated as follows:

- The leftmost monkey zooms in and out along the z-axis.
- The middle monkey rotates around the y-axis.
- The rightmost monkey moves up and down along the y-axis.

These animations are implemented using simple trigonometric functions, such as `Math.sin`, to create smooth, oscillating movements. The `renderMonkey` function is responsible for rendering each monkey at its updated position.

```
rotation += 0.05;
zoom = 5 + Math.sin(time * 0.005) * 3;
upDown = Math.sin(time * 0.005) * 3;

renderMonkey([-6, 0, zoom], null);
renderMonkey([0, 0, 0], rotation);
renderMonkey([6, upDown, 0], null);
```

The animation for each monkey is based on sine wave functions to create smooth periodic movements. The first monkey's position is adjusted by the `zoom` value along the z-axis, while the second monkey rotates around the y-axis. The third monkey moves vertically along the y-axis, with smooth oscillations.

3.7 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.

3.8 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.

3.9 Rendering Loop

The rendering loop continuously clears the screen, updates the camera view, and renders the three monkey models with their respective animations. The loop is driven by the `requestAnimationFrame` function, which ensures smooth, synchronized frame updates.

```
function render(time) {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    gl.uniform3fv(cameraPositionLocation, cameraPosition);

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

    rotation += 0.05;
    zoom = 5 + Math.sin(time * 0.005) * 3;
    upDown = Math.sin(time * 0.005) * 3;

    renderMonkey([-6, 0, zoom], null);
    renderMonkey([0, 0, 0], rotation);
    renderMonkey([6, upDown, 0], null);

    updateViewMatrix();
    requestAnimationFrame(render);
}
```

The `render` function is the main rendering loop that continuously updates the scene. It clears the screen, updates the camera's view matrix, and renders the monkeys with their animations. The

requestAnimationFrame ensures smooth rendering and synchronization with the browser's refresh rate.

3.10 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.

3.11 Conclusion

This section covers the implementation of a 3D scene with animated monkeys and a controllable camera in WebGL2. The algorithms for animation, camera control, and rendering ensure an interactive experience. Mouse inputs control the camera's zoom, rotation, and translation, while the rendering loop continuously updates the scene. The canvas resizing functionality ensures that the scene adapts to changes in window size, maintaining the correct aspect ratio and preventing distortion.

3.12 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.

3.12.1 Project Structure

Ensure the project files are organized as follows:

- index.html: The main HTML file for the project.
- monkey.obj: The 3D object file used for rendering the monkey heads.

- `js/`: A folder containing all JavaScript files, including the code for rendering, animation, and camera control.

3.12.2 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.

4 Conclusion

This assignment demonstrated the fundamentals and advanced concepts of 3D rendering using WebGL2, divided into two progressive parts.

4.1 Part 1: Rendering a 3D Cube

In Part 1, we implemented a 3D cube to explore the basics of WebGL2, including shaders, buffer management, and transformation matrices. User interactions, such as dynamic view updates via event listeners, enhanced the interactivity and visualization of the rendered object.

4.2 Part 2: Animated Monkey Heads and Controllable Camera

Part 2 expanded on these foundations by rendering an animated scene with three monkey heads and a user-controllable camera. This part involved loading 3D models, applying transformations for animation, and integrating advanced user interactions for camera control, offering a more immersive experience.

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`