Tarea 2.2: Dos texturas

Alumno: Jeanne LE GALL-BOTTE

El objetivo de este programa es representar un cilindro tridimensional con tapas superior e inferior en WebGL, aplicar texturas diferentes a sus caras y permitir al usuario explorar la escena mediante un sistema de cámara controlado con teclado y botones en pantalla. Además, se implementa un sistema de interpolación suave para que los movimientos de la cámara sean fluidos.

```
<body>
    <canvas id="canvas" width="700" height="700"></canvas>
    <div id="controls">
        <div>
            <button class="smallButton" onclick="zoomIn()">+</button>
            <button class="smallButton" onclick="up()">↑</button>
            <button class="smallButton" onclick="zoomOut()">−</button>
        </div>
        <div>
            <button class="smallButton" onclick="left()">←</button>
            <button class="smallButton" onclick="down()">↓</button>
            <button class="smallButton" onclick="right()">→</button>
        </div>
        <div>
            <button class="reset" onclick="reset()">Reset</button>
        </div>
    </div>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.4.0/gl-matrix.js"></script>
    <script>
        var canvas = document.getElementById('canvas');
        var gl = canvas.getContext('webgl');
```

El programa se renderiza en un canvas de 700×700 píxeles. En la esquina superior izquierda se encuentra un panel de control con botones de flechas para mover la cámara en coordenadas esféricas y botones + y – para acercar o alejar la cámara. Hay también un botón Reset que devuelve la cámara a su estado inicial. Estos mismos controles se pueden activar desde el teclado con las teclas de flechas y los símbolos + y -.

```
// Cámara
var radius = 20;
var theta = Math.PI / 2;
var phi = 0.0;

var targetRadius = radius;
var targetTheta = theta;
var targetPhi = phi;

function up() {
    targetPhi += 0.1;
    if (targetPhi > Math.PI / 2) targetPhi = Math.PI / 2;
}

function down() {
    targetPhi -= 0.1;
    if (targetPhi < -Math.PI / 2) targetPhi = -Math.PI / 2;
}

function left() {
    targetTheta -= 0.1;
}

function right() {
    targetTheta += 0.1;
}

function zoomIn() {
    targetRadius = Math.max(2, targetRadius - 0.5);
}

function zoomOut() {
    targetRadius += 0.5;
}

function reset() {
    radius = 20; theta = Math.PI / 2; phi = 0.0;
    targetRadius = radius; targetTheta = theta; targetPhi = phi;
}
```

La cámara se define en coordenadas esféricas mediante tres parámetros, que son *radius* que corresponde a la distancia de la cámara al objeto, *theta* que es el ángulo de rotación horizontal alrededor del eje Y, y phi el ángulo de elevación vertical. Para lograr movimientos fluidos, cada parámetro tiene una versión objetivo (*targetRadius*, *targetTheta*, *targetPhi*). En cada frame, los valores actuales se interpolan progresivamente hacia los objetivos, generando una transición suave en lugar de un salto inmediato. Para hacer los movimientos de la cámara, se utilizan 6 funciones. Las funciones *up()* y *down()* aumentan y disminuyen phi, limitado entre –π/2 y π/2, las funciones *left()* y *right()* modifican theta para desplazar la vista horizontalmente, y las funciones *zoomIn()* y *zoomOut()* acercan o alejan la cámara variando *radius*. Finalmente, la función *reset()* devuelve todos los parámetros a sus valores iniciales.

```
function Cylinder(radius = 1, height = 2, radialSegments = 36) {
    var positionsSide = [], texSide = [], indicesSide = [];

    // Side
    for (let i = 0; i <= radialSegments; i++) {
        var th = (i / radialSegments) * Math.PI * 2;
        var x = Math.cos(th) * radius, z = Math.sin(th) * radius;
        positionsSide.push(x, -height / 2, z); texSide.push(i / radialSegments, 1.0);
        positionsSide.push(x, height / 2, z); texSide.push(i / radialSegments, 0.0);
    }
    for (let i = 0; i < radialSegments; i++) {
        var a = i * 2, b = a + 1, c = a + 2, d = a + 3;
        indicesSide.push(a, b, c); indicesSide.push(b, d, c);
    }

    // Caps (top and bottom)
    function makeCap(y, flip) {
        var positions = [0, y, 0], tex = [0.5, 0.5], indices = [];
        for (let i = 0; i <= radialSegments; i++) {
            var th = (i / radialSegments) * Math.PI * 2;
            var x = Math.cos(th) * radius, z = Math.sin(th) * radius;
            positions.push(x, y, z);
            tex.push(0.5 + 0.5 * Math.cos(th), 0.5 + 0.5 * Math.sin(th));
        }
        for (let i = 1; i <= radialSegments; i++) {
            var next = (i % radialSegments) + 1;
            if (!flip) indices.push(0, i, next); else indices.push(0, next, i);
        }
        return { positions, tex, indices };
    }

    var top = makeCap(height / 2, false), bottom = makeCap(-height / 2, true);
    return {
        side: { positions: new Float32Array(positionsSide), tex: new Float32Array(texSide), indices: new Uint16Array(indicesSide) },
        top: { positions: new Float32Array(top.positions), tex: new Float32Array(top.tex), indices: new Uint16Array(top.indices) },
        bottom: { positions: new Float32Array(bottom.positions), tex: new Float32Array(bottom.tex), indices: new Uint16Array(bottom.indices) }
    };
}

var cylindre = Cylinder(1, 2, 64);
var side = cylindre.side, topCap = cylindre.top, bottomCap = cylindre.bottom;
```

El cilindro se genera con la función *Cylinder(radius, height, radialSegments)*, que produce una superficie lateral, con coordenadas de textura desplegadas horizontalmente y las tapas superior e inferior, construidas como triángulos en abanico con coordenadas de textura centradas. Cada parte devuelve sus posiciones de vértices, coordenadas de textura e índices, encapsulados en objetos para facilitar su renderizado.

```
function makeVAO(positions, texcoords, indices) {
    if (!positions || !texcoords || !indices) {
        console.error("Error VAO, datos faltantes", { positions, texcoords, indices });
        return null;
    }
    var vao = {};
    vao.positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vao.positionBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, positions, gl.STATIC_DRAW);

    vao.texBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vao.texBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, texcoords, gl.STATIC_DRAW);

    vao.indexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vao.indexBuffer);
    gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);

    vao.numElements = indices.length;
    return vao;
}

var sideVAO = makeVAO(side.positions, side.tex, side.indices);
var topVAO = makeVAO(topCap.positions, topCap.tex, topCap.indices);
var bottomVAO = makeVAO(bottomCap.positions, bottomCap.tex, bottomCap.indices);
```

La función *makeVAO(positions, texcoords, indices)* crea tres buffers en la GPU: uno para las posiciones de los vértices, otro para las coordenadas de textura y un tercero para los índices que definen los triángulos. Todos estos datos se agrupan en un objeto llamado VAO (Vertex Array Object). En WebGL, este VAO es una estructura simulada que

simplemente guarda juntos los buffers y el número de elementos. Su utilidad es simplificar el renderizado. En efecto, en lugar de manejar cada buffer por separado, basta con pasar el VAO a la función *drawVAO()* para activar los datos correctos y dibujar la parte del cilindro con su textura.

```
// Shaders
var vertCode = `
    attribute vec3 position;
    attribute vec2 texCoord;

    uniform mat4 Pmatrix;
    uniform mat4 MVmatrix;

    varying vec2 vTexCoord;

    void main(void){
        gl_Position = Pmatrix*MVmatrix*vec4(position,1.0); vTexCoord=texCoord;
    }
`;

var fragCode = `
    precision mediump float;

    varying vec2 vTexCoord;

    uniform sampler2D textura;

    void main(void){
        gl_FragColor = texture2D(textura,vTexCoord);
    }
`;
```

Para las texturas, el vertex shader recibe las coordenadas de textura y el fragment shader toma las coordenadas de textura interpoladas y aplica la muestra correspondiente desde la textura cargada.

```
function loadTexture(url, callback) {
    var texture = gl.createTexture();
    var image = new Image();
    image.onload = function () {
        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);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
        if (callback) callback();
    };
    image.src = url;
    image.onerror = () => console.error("Impossible de charger la texture:", url);
    return texture;
}

let texturesLoaded = 0;
function onTextureLoad() {
    texturesLoaded++; if (texturesLoaded === 2) requestAnimationFrame(animate);
}
var texSide = loadTexture("side.jpg", onTextureLoad);
var texCap = loadTexture("cap.jpg", onTextureLoad);
```

Las dos texturas se cargan mediante la función *loadTexture(url)*, la imagen *side.jpg* es la textura de la superficie lateral y la imagen *cap.jpg* esta aplicada a las tapas superior y inferior. Así, la función *loadTexture(url)* permite de crear un objeto de textura en WebGL y asocia una imagen que se carga de forma asíncrona. En esta función ocurren varias cosas:

- *gl.createTexture()* reserva un espacio de memoria para la textura en la GPU
- *gl.bindTexture(gl.TEXTURE_2D, texture)* activa la textura para que las siguientes operaciones la modifiquen
- *gl.texParameteri* configura cómo se repite la textura y cómo se interpola, en este caso se limita a los bordes y se usa interpolación lineal para evitar el efecto pixelado
- *gl.texImage2D* transfiere los datos de la imagen al objeto de textura en la GPU

```
function drawVAO(vao, texture) {
    if (!vao) return;
    gl.bindBuffer(gl.ARRAY_BUFFER, vao.positionBuffer);
    gl.vertexAttribPointer(position, 3, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, vao.texBuffer);
    gl.vertexAttribPointer(texCoord, 2, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vao.indexBuffer);
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.activeTexture(gl.TEXTURE0);
    gl.uniform1i(textura, 0);
    gl.drawElements(gl.TRIANGLES, vao.numElements, gl.UNSIGNED_SHORT, 0);
}

drawVAO(sideVAO, texSide);
drawVAO(topVAO, texCap);
drawVAO(bottomVAO, texCap);
```

Antes de dibujar cada parte del cilindro, el programa selecciona la textura adecuada mediante la función *drawVAO()*. Para ello, en primer lugar activa la unidad de textura 0 *TEXTURE0* y asocia en ella la imagen que corresponde a la pieza que se va a renderizar, ya sea la superficie lateral *texSide* o una de las tapas *texCap*. A continuación, se actualiza la variable uniforme textura en el shader, indicándole que debe leer los datos de color desde la unidad de textura actualmente activa.

```
// Dibujar escena
let timeOld = 0;
function animate(time) {
    var dt = time - timeOld; timeOld = time;

    var projMatrix = mat4.create();
    mat4.perspective(projMatrix, 40 * Math.PI / 180, canvas.width / canvas.height, 1, 100);
    gl.uniformMatrix4fv(Pmatrix, false, projMatrix);

    theta += (targetTheta - theta) * 0.1;
    phi += (targetPhi - phi) * 0.1;
    radius += (targetRadius - radius) * 0.1;

    var eye = [
        radius * Math.cos(theta) * Math.cos(phi),
        radius * Math.sin(phi),
        radius * Math.sin(theta) * Math.cos(phi)
    ];
    var viewMatrix = mat4.create();
    mat4.lookAt(viewMatrix, eye, [0, 0, 0], [0, 1, 0]);

    var modelViewMatrix = mat4.clone(viewMatrix);
    mat4.translate(modelViewMatrix, modelViewMatrix, [4 * Math.cos(0.001 * time), 4 * Math.sin(0.001 * time), 0]);
    mat4.rotateZ(modelViewMatrix, modelViewMatrix, 0.001 * time + Math.PI / 2);
    mat4.rotateY(modelViewMatrix, modelViewMatrix, 0.004 * time);
    gl.uniformMatrix4fv(MVmatrix, false, modelViewMatrix);

    gl.enable(gl.DEPTH_TEST); gl.depthFunc(gl.LEQUAL);
    gl.clearColor(0.5, 0.5, 0.5, 0.9);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    function drawVAO(vao, texture) {
        if (!vao) return;
        gl.bindBuffer(gl.ARRAY_BUFFER, vao.positionBuffer);
        gl.vertexAttribPointer(position, 3, gl.FLOAT, false, 0, 0);
        gl.bindBuffer(gl.ARRAY_BUFFER, vao.texBuffer);
        gl.vertexAttribPointer(texCoord, 2, gl.FLOAT, false, 0, 0);
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, vao.indexBuffer);
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.activeTexture(gl.TEXTURE0);
        gl.uniform1i(textura, 0);
        gl.drawElements(gl.TRIANGLES, vao.numElements, gl.UNSIGNED_SHORT, 0);
    }

    drawVAO(sideVAO, texSide);
    drawVAO(topVAO, texCap);
    drawVAO(bottomVAO, texCap);

    requestAnimationFrame(animate);
}
```

La animación se ejecuta en la función *animate()*, llamada continuamente con *requestAnimationFrame*. Se construye la matriz de proyección Pmatrix con mat4.perspective, después se calcula la posición de la cámara *eye* a partir de *radius*, *theta* y *phi*. Luego, se genera la matriz de vista con mat4.lookAt. Al modelo se le aplican transformaciones dinámicas de *mat4.translate* para desplazar el cilindro en una trayectoria circular dependiente del tiempo, y *mat4.rotateZ* y *mat4.rotateY* que hacen girar el cilindro sobre sus propios ejes. Finalmente, se dibuja cada parte del cilindro con la textura correspondiente.

Para que las texturas se carguen correctamente, es necesario ejecutar el código desde un servidor local, ya que los navegadores suelen bloquear la carga de archivos de imagen cuando se abren directamente con file://. Una forma sencilla de hacerlo es utilizando Python, que incluye un pequeño servidor web integrado. Los pasos son:

Abre una terminal en la carpeta donde se encuentra el archivo .html y las imágenes de textura (side.jpg, cap.jpg).

Si tienes Python 3 instalado, escribe el siguiente comando:

python3 -m http.server 8000

Luego, abre un navegador web y escribe en la barra de direcciones:

http://localhost:8000/Tarea2-2.html