Tarea 2.3: Ordenamiento

Alumno: Jeanne LE GALL-BOTTE

https://github.com/jeannelgb/Graficas-por-Computadora/tree/main/Tarea%202.3

El objetivo de este proyecto es crear una visualización interactiva en WebGL en la que cinco instancias de un mismo objeto se ordenen automáticamente según su tamaño. En este caso, el objeto elegido es un cruce. Además, se implementa un algoritmo de ordenamiento tipo burbuja que permite ver cómo los objetos cambian de posición gradualmente hasta que quedan correctamente ordenados. La animación se realiza de forma progresiva, para que el usuario pueda observar el proceso de ordenamiento paso a paso.

```
<body>
    <canvas id="canvas" width="800" height="800"></canvas>
    <div id="controls">
        <button class="smallButton" onclick="activateSort()">Sort</button>
        <button class="smallButton" onclick="reset()">Reset</button>
    <div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/gl-matrix/2.4.0/gl-matrix.js"></script>
    <script>
        const canvas = document.getElementById('canvas');
        const gl = canvas.getContext('webgl');
        const ext = gl.getExtension('ANGLE_instanced_arrays');
```

El programa se renderiza en un canvas de 800×800 píxeles. En la esquina superior izquierda se encuentra un panel de control con un botón *Sort* para accionar el ordenamiento y un botón *Reset* que restablece con nuevos elementos. Adicionalmente, se habilita la extensión *ANGLE_instanced_arrays*, que permite dibujar múltiples instancias del mismo objeto de manera eficiente sin necesidad de enviar los datos de cada objeto repetidamente a la GPU. Esta extensión es crucial para animar varias copias de un objeto con distintas transformaciones y colores.

```
var vertices = [
    -0.5, 0.15, 0.0,
    0.5, 0.15, 0.0,
    -0.5, -0.15, 0.0,
    -0.5, -0.15, 0.0,
    0.5, -0.15, 0.0,
    0.5, 0.15, 0.0,

    -0.15, 0.5, 0.0,
    0.15, 0.5, 0.0,
    -0.15, -0.5, 0.0,
    -0.15, -0.5, 0.0,
    0.15, -0.5, 0.0,
    0.15, 0.5, 0.0
];
```

```
var indices = [
    0, 1, 2,
    3, 4, 5,
    6, 7, 8,
    9, 10, 11
];

var colors = [
    1, 0, 0,
    0, 1, 0,
    0, 0, 1,
    1, 0, 1,
    0, 1, 1
];
```

El objeto que se renderiza está compuesto por dos rectángulos que forman una especie de cruz. Los datos geométricos se almacenan en tres arreglos que son los vértices, los índices y las colores. La lista *vertices* contiene las coordenadas tridimensionales de cada vértice. Cada vértice está definido por tres valores (x, y, z). La lista *indices* especifica cómo se deben conectar los vértices para formar los triángulos que componen el objeto. Esto evita duplicar vértices y optimiza la renderización. Finalmente, la lista *colors* define el color RGB de cada instancia. Cada objeto puede tener un color distinto, lo que facilita la identificación visual durante el proceso de ordenamiento.

```
// Vertex buffer
var vertexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

// Index buffer
var indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices), gl.STATIC_DRAW);

// Color buffer
var colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);
```

Estos datos se envían a la GPU mediante buffers. Se crean tres buffers principales: *vertexBuffer* para los vértices, *indexBuffer* para los índices y *colorBuffer* para los colores. Cada buffer se llena utilizando *gl.bufferData*, garantizando que la GPU tenga acceso eficiente a los datos.

```
// Shaders
var vertCode =
    'uniform vec4 translation;' +
    'uniform mat4 Pmatrix;' +
    'uniform mat4 MVmatrix;' +
    'attribute vec3 position;' +
    'attribute vec3 color;' +
    'varying vec3 vColor;' +
    'void main(void) {' +
    ' gl_Position = Pmatrix*MVmatrix*vec4(position, 1.0) + translation;' +
    ' vColor = color;' +
    '}';

var fragCode =
    'precision mediump float;' +
    'varying vec3 vColor;' +
    'void main(void) {' +
    ' gl_FragColor = vec4(vColor, 1.0);' +
    '}';
```

Se implementan los códigos de los dos shaders, el *vertexShader* transforma los vértices de cada objeto aplicando matrices de proyección *Pmatrix* y modelo-vista *MVmatrix*. También aplica una translación y pasa el color de cada vértice al fragment shader. Esto permite que cada objeto pueda moverse, rotar y escalar independientemente en el espacio tridimensional. El *fragmentShader* determina el color final de cada píxel. Utiliza la información de color transmitida por el vertex shader *vColor* y asigna un valor RGBA completo para el renderizado.

```
var vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, vertCode);
gl.compileShader(vertShader);

var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragCode);
gl.compileShader(fragShader);

var program = gl.createProgram();
gl.attachShader(program, vertShader);
gl.attachShader(program, fragShader);
gl.linkProgram(program);
gl.useProgram(program);
```

Estos shaders se compilan y se enlazan en un programa WebGL que luego se utiliza para dibujar los objetos en la escena.

```
gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var position = gl.getAttribLocation(program, "position");
gl.vertexAttribPointer(position, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(position);

var color = gl.getAttribLocation(program, "color");
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.vertexAttribPointer(color, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(color);
ext.vertexAttribDivisorANGLE(color, 1);

var Pmatrix = gl.getUniformLocation(program, 'Pmatrix');
var MVmatrix = gl.getUniformLocation(program, "MVmatrix");
```

Después, se encarga de preparar los datos de los objetos para que el shader los pueda procesar. Primero, vincula el buffer de vértices y asocia el atributo *position* del shader, indicando cómo leer las coordenadas de cada vértice y activando este atributo para el renderizado. Luego, hace lo mismo con el buffer de colores, conectando el atributo *color* y usando *vertexAttribDivisorANGLE* para que cada instancia pueda tener un color

diferente. Finalmente, obtiene las ubicaciones de las matrices uniformes *Pmatrix* y *MVmatrix*, que se usarán para proyectar y transformar los objetos en el espacio 3D.

```
var instances = [];
for (let i = 0; i < numInstances; i++) {
    var scale = Math.random();
    instances.push({
        size: scale,
        pos: -2 + i,
        targetPos: -2 + i
    });
}
```

Se crean cinco instancias del objeto, cada una con un tamaño aleatorio. Así, cada instancia se representa como un objeto que contiene tres propiedades que son *size*, *pos* y *targetPos*. La propiedad *size* define la escala de la instancia en los tres ejes, la propiedad *pos* define la posición actual en el eje X y *targetPos* la posición a la que debe moverse durante el proceso de ordenamiento. Inicialmente, cada objeto se coloca en una posición secuencial sobre el eje X, asegurando que estén separados y sean visibles.

```
var sorting = true;
var step = 0;
var moving = false;
var sortActivated = false;

function activateSort() {
    sortActivated = true
}

function reset() {
    instances = [];
    sorting = true;
    step = 0;
    moving = false;
    sortActivated = false;

    for (let i = 0; i < numInstances; i++) {
        var scale = Math.random();
        instances.push({
            size: scale,
            pos: -2 + i,
            targetPos: -2 + i
        });
    }
}
```

La función *activateSort()* se ejecuta al presionar el botón Sort. Su único objetivo es activar la variable *sortActivated*, que indica al programa que debe iniciar el proceso de ordenamiento en la siguiente iteración del bucle de animación. La función *reset* reinicia la escena. Genera nuevas instancias con tamaños aleatorios, restablece todas las

variables de control y coloca los objetos en sus posiciones iniciales. Esto permite repetir la animación de ordenamiento sin recargar la página.

```
function sort() {
    if (!sorting || moving) return;

    if (step < instances.length - 1) {
        if (instances[step].size > instances[step + 1].size) {

            let tempTarget = instances[step].targetPos;
            instances[step].targetPos = instances[step + 1].targetPos;
            instances[step + 1].targetPos = tempTarget;

            let tempInstance = instances[step];
            instances[step] = instances[step + 1];
            instances[step + 1] = tempInstance;

            moving = true;
        }
        step++;
    } else {
        step = 0;
        let sorted = true;
        for (let i = 0; i < instances.length - 1; i++) {
            if (instances[i].size > instances[i + 1].size) {
                sorted = false;
                break;
            }
        }
        if (sorted) sorting = false;
    }
}
```

El algoritmo de ordenamiento utilizado es una burbuja adaptada a animación. En cada llamada, compara pares consecutivos de instancias y si la instancia de la izquierda es más grande que la de la derecha, se intercambian sus *targetPos* y se actualiza el orden en el arreglo *instances*. La variable *moving* se activa para indicar que hay objetos en movimiento y que la animación debe continuar. Se avanza al siguiente par de objetos. Cuando se termina un ciclo completo, se verifica si la lista está ordenada. Si todas las instancias están en el orden correcto, *sorting* se establece en false, deteniendo el algoritmo. Esta implementación permite visualizar cada paso del ordenamiento, haciendo evidente cómo los objetos más grandes "suben" o "bajan" hasta su posición final.

```
function updatePositions(){
    const speed = 0.2;
    moving = false;
    for(let inst of instances){
        inst.pos += (inst.targetPos - inst.pos) * speed;
        if (Math.abs(inst.targetPos - inst.pos) > 0.001) moving = true;
    }
}
```

La función *updatePosition()* realiza la animación de las instancias, moviéndolas suavemente hacia sus posiciones objetivo. La variable *speed* controla la rapidez de desplazamiento. La función verifica si alguna instancia sigue en movimiento y actualiza la variable *moving*. Este movimiento progresivo crea una animación fluida, evitando cambios bruscos de posición.

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

    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);

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

    if(sortActivated) {
        sort();
        updatePositions();
    }

    for (let i = 0; i < instances.length; i++) {
        let instance = instances[i];
        var mv = mat4.create();
        mat4.translate(mv, mv, [instance.pos/2, 0.0, -4.0]);
        mat4.rotateZ(mv, mv, time * 0.001 * (0.5 + 0.2 * i));
        mat4.scale(mv, mv, [instance.size, instance.size, instance.size]);
        gl.uniformMatrix4fv(MVmatrix, false, mv);
        gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
        ext.drawElementsInstancedANGLE(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT, 0, i + 1);
    }

    window.requestAnimationFrame(animate);
};
animate(0);
```

El bucle de animación realiza las siguientes tareas. Primero, se hace la limpieza del canvas y activa el depth test para asegurar el renderizado correcto de objetos 3D. Luego, se calcula la matriz de proyección perspectiva, adecuada para simular una cámara tridimensional. Si *sortActivated* es true, se llama a *sort()* y *updatePositions()*. Esto garantiza que el ordenamiento y la animación se actualicen en cada frame. Para cada instancia, se aplica traslación, rotación y escala mediante matrices de transformación, se envía la matriz de modelo-vista a la GPU y se dibuja la instancia usando *drawElementsInstancedANGLE*. Finalmente, se solicita el siguiente frame con *requestAnimationFrame(animate),* manteniendo la animación continua.