Tarea 1.2: Elaboración de un programa recursivo con WebGL

Alumno: Jeanne LE GALL-BOTTE

El objetivo de esta tarea estaba de elaborar un programa recursivo y he elegido de hacer un árbol fractal. El usador puede controlar el nombre de iteraciones de la recursión con una interfaz sencilla que es un cursor.

```
function createShader(gl, type, src) {
  const sh = gl.createShader(type);
  gl.shaderSource(sh, src); gl.compileShader(sh);
  if (!gl.getShaderParameter(sh, gl.COMPILE_STATUS)) {
    throw new Error("Shader compile error: " + gl.getShaderInfoLog(sh));
  }
  return sh;
}
function createProgram(gl, vs, fs) {
  const p = gl.createProgram();
  gl.attachShader(p, vs); gl.attachShader(p, fs); gl.linkProgram(p);
  if (!gl.getProgramParameter(p, gl.LINK_STATUS)) {
    throw new Error("Program link error: " + gl.getProgramInfoLog(p));
  }
  return p;
}
```

El código define dos programas pequeños escritos en GLSL que son el shader de vértices y el de fragmentos. La única función del shader de vértices es recibir las coordenadas de cada punto de la geometría (atributo *a_pos*) y pasarlas al pipeline gráfico. No realiza transformaciones, simplemente coloca los puntos en el espacio de coordenadas de la pantalla. El shader de fragmentos se encarga de dar color a cada fragmento (pixel) dibujado. En este caso, utiliza una variable uniforme *u_color*, lo que significa que todo el árbol se renderiza con un color único y constante. Estos shaders son compilados mediante la función *createShader*, y luego enlazados en un programa de WebGL con la función *createProgram*. Si ocurre algún error de compilación o enlace, se muestra un mensaje de advertencia.

Una vez listo el programa de shaders, se crea un buffer de vértices en el que se almacenarán todas las coordenadas de los segmentos del árbol. El atributo *a_pos* se activa y se le indica cómo debe interpretar los datos: cada vértice tiene dos componentes

(x, y) de tipo flotante. De esta manera, cuando WebGL reciba el conjunto de vértices, sabrá cómo leerlos correctamente para dibujar las líneas.

```
function addSegment(out, x1, y1, x2, y2){
  out.push(x1, y1, x2, y2);
}

function buildTree(out, x, y, length, angle, depth){
  const x2 = x + length * Math.cos(angle);
  const y2 = y + length * Math.sin(angle);
  addSegment(out, x, y, x2, y2);

  if (depth <= 0) return;

  buildTree(out, x2, y2, length*0.7, angle+Math.PI/6, depth-1);
  buildTree(out, x2, y2, length*0.7, angle-Math.PI/6, depth-1);
}
```

La función *buildTree* construye el árbol fractal según las etapas siguientes. En primer lugar, es el cálculo de una rama: a partir de un punto de partida con coordenadas (x, y) se calculen las coordenadas del punto de llegada utilizando la longitud de la rama y su ángulo (aquí siempre 30°). Después, se hace la adición de un nuevo segmento, que se registre en una tabla *out* donde están todas las coordenadas de las ramas. En caso de que la profundidad no está igual a 0, la función se llama ella misma dos veces, una vez para la rama de la derecha y una vez para la de izquierda. A cada nueva profundidad se reduce la longitud de las ramas. En caso de que la profundidad esta igual a 0, la recursión se detiene y ninguna rama esta creada.

```
function buildGeometry(){
  const data = [];
  const startX = 0.0;
  const startY = -1.0;
  const length = 0.5;
  const angle = Math.PI / 2;

  buildTree(data, startX, startY, length, angle, maxDepth);

  const f32 = new Float32Array(data);
  gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
  gl.bufferData(gl.ARRAY_BUFFER, f32, gl.STATIC_DRAW);
  vertCount = f32.length / 2;
}
```

La función *buildGeometry* se encarga de preparar todos los datos que representarán el árbol. Ellq define el punto inicial del tronco en la parte inferior de la pantalla (startX = 0.0, startY = -1.0) y establece la longitud inicial y el ángulo vertical hacia arriba. Luego, llama a la función recursiva *buildTree*, que genera todos los segmentos según la profundidad máxima elegida. Finalmente, convierte el resultado en un arreglo de Float32Array para que pueda ser entendido por WebGL, y lo carga en el buffer.

De esta forma, cada vez que cambia el número de iteraciones, se reconstruye la geometría completa del árbol.