PEC2: Fuego en el Shader

Pedro Jesús Sánchez Illescas

May 31, 2024

Índice

1 Librerías utilizadas

- SDL 2.30.0
- SDL-image 2.8.2

2 Proceso de creación

Este proyecto se ha desarrollado partiendo del código del reto 15 del cuaderno junto con el minimotor de efectos desarrollado en la práctica anterior, con leves modificaciones para poder dibujar usando ventanas OpenGL en lugar de superficies y la posibilidad de usar input del usuario en los efectos. El primer paso ha consistido en refactorizar el código inicial con los objetivos de tener un código más compacto y profundizar en la comprensión del procedimiento. Gracias a este paso, el agua se ha renderizado usando un solo objeto *WaterObject* que engloba a las clases *WaterPlane*, manteniendo *Object3D* como base, además de separando la funcionalidad de los frame buffers en la clase *FrameBuffer* en una clase separada, aumentando la legibilidad de *WaterObject*haciendo que, por ejemplo, los métodos *bindReflectionFrameBuffer()* y *unbindReflectionFrameBuffer()* pasen a ser *reflectionFrameBuffer-¿bind()* y *reflectionFrameBuffer-¿unbind()*. También se ha mantenido las clases *Shader* y *Camera3D*, extrayendo la lógica de actualización de matrices al proceso de renderizado de los objetos, simplificando el proceso de actualización de los shaders, que siempre se harán en sus propios objetos.

Toda la lógica de la PEC está en la clase *Pec2Effect*, con la mayor parte del trabajo en el método *render*, puesto que *update* se ocupa de gestionar la entrada del usuario.

La clase *WaterObject* final ha sido un buen sustrato para los objetos creados para esta PEC: una llama y un río de lava, que se explicarán en los próximos apartados.

2.1 Llama

Este objeto con parte C++ *FlameObject* y shader *Flame* modela un fuego con dos partes bien definidas: la llama y una cortina de humo. Ambas partes se controlan desde el mismo shader vía una textura de máscara con forma de llama cuya superficie blanca delimita la llama y la negra será el fondo o cortina de humo, controladas por dos partes distintas del shader, aunque ambas tienen algo de transparencia implementada mediante una textura de refracción.

2.1.1 Sección de llama

La llama es simplemente una superficie de un color amarillento salpicada con tonos negruzcos vía ruido de Perlin dinámico, cuya generación se ha obtenido de [1] y se ha abstraído en la función *getPerlin* dentro del shader *Flame.fragment*.

2.1.2 Sección de humo

La sección de humo es un poco más compleja que la llama, puesto que además de realizar una pequeña distorsión del fondo, tiene un difuminado desde el centro de la pantalla de forma exponencial tipo $y = e^{-\alpha d^2}$, siendo d la distancia del punto al centro de la pantalla, en coordenadas de textura (función *getAbsorptionFactor*). En este caso, el valor de α es tal que nos da una atenuación de 0.01 a $d^2 = 1.2$, aproximadamente 1.01 unidades de textura del centro de la pantalla.

2.1.3 Shader: Flame

Tanto para la llama como para la lava, el shader de vértices es realmente simple, se limita a aplicar las matrices para hallar la posición en el espacio donde se encuentra el quad de dibujado y las coordenadas UV de la textura que le corresponde a ese vértice.

El grueso del procesado se encuentra en el shader de fragmento, donde la mayor parte del código se utiliza para el cálculo del ruido de Perlin usado en la llama extraído de github [1]. Como esa parte del código es externa al proyecto, no está incluida en la transcripción del shader más abajo. Primeramente se calcula el valor del fondo que corresponde al píxel, puesto que tanto la llama como el humo tiene transparencia, y dependiendo si el píxel es de llama o fondo se le aplica a este fondo su procesado.

En el caso de que el píxel sea de llama, simplemente se hace un mix con el color del fuego aplicando el ruido de Perlin para simular el movimiento de la llama con una intensidad de 3.0 para realzar la luz que emite el fuego. Y por último, si el píxel es del fondo, como se comentó antes, se le aplica una distorsión y una decoloración para simular el humo.

2.1.4 Intento fallido: Máscara dinámica

Para dar un poco más de realismo a la llama, intenté crear una máscara dinámica replicando el efecto de fuego visto en la práctica anterior, usando la matriz de convolución (1), aplicándola en cada frame a un framebuffer donde estaría ahora la máscara, añadiendo una línea de píxeles en la zona baja de la textura, tal y como se hacía en el apartado anterior. Esto se consigue teniendo varios estados en el shader, definidos por las variables *initialize* y *heat*, para indicar si se inicializa la textura (por defecto la textura de máscara original), se aplica la etapa de calor (activación de una línea de píxeles en la zona inferior de la textura), o la convolución del framebuffer con el filtro de fuego (1). Pero el único resultado conseguido ha sido una animación de expansión de la llama hasta llenar la pantalla, con lo que este intento ha sido descartado, pero creo que vale la pena señalarlo en esta memoria. Más abajo se añade el código del fragment shader final de este paso.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{1}$$

```
#version 330 core

in vec2 texCoord;

uniform bool initialize;
uniform bool heat;
uniform sampler2D texture1;
uniform float kernel[9]; // valores de la matriz de filtro
uniform vec2 tex_offset[9]; // coordenadas UV relativas a texCoord del filtro
// limites de pixeles
uniform float stepHeight; // limite superior e inferior
uniform float stepWidth; // limite izquierda y derecha

out vec4 FragColor;

bool isValidCoords(vec2 coords)
{
  return 0 < coords.x && coords.x < 1 && 0 < coords.y && coords.y < 1;
}

bool isValidForConvolution(vec2 v)
{
  return stepHeight < v.y && v.y < (1 - stepHeight) &&
    stepWidth < v.x && v.x < (1 - stepWidth);
}

vec4 convolute()
{
  vec3 result = vec3(0.0, 0.0, 0.0);
  if(isValidForConvolution(texCoord))
  {
    for (int i = 0; i < 9; i++)
    {
      vec2 coords = texCoord + tex_offset[i];
      vec3 texColor = (isValidCoords(coords)) ?
        texture(texture1, texCoord + tex_offset[i]).rgb : vec3(1.0,1.0,1.0);
      result += texColor * kernel[i];
    }
  }

  return vec4(result.xyz,1);
}


void main()
{
  if (heat)
  {
    if (0.1 <= texCoord.y && texCoord.y <= 0.3)
    {
      FragColor = vec4(1,1,1,1);
    }
  }
  else
  {
    FragColor = (initialize) ? texture(texture1, texCoord) :
```

```
        convolute();
    }
}
```

2.2 Lava

Este objeto está formado por la clase C++ *LavaObject* y el shader *Lava*. Este objeto es más simple que el anterior, aun compartiendo la misma estructura y cambiando la textura de refracción por una textura de reflexión. Como ya se dijo antes, el vertex shader es exactamente igual al de la llama, con lo que no vamos a incidir más en ello. Pero el fragment shader tiene como base el shader del agua, usando textura de distorsión y un mapa de normales, sumándole a todo ello una simulación de corriente de convección como en la demo [2].

Por vistosidad y sacrificando realismo, este material es muy reflectivo a diferencia de la lava real, pero me gusta mucho el material tan reflectivo que recuerda a juegos clásicos de PSX.

3 Compilando la aplicación

El ejecutable, una vez compilados todos los fuentes situados en la carpeta */src*, en el directorio del ejecutable hay que copiar la carpeta */assets* donde se encuentran los shaders (*/assets/shaders*) y texturas (*/assets/textures*) necesarias para el correcto funcionamiento del programa, además de los archivos *sdl2.dll* y *sdl2_image.dll*.

4 Conclusiones

En esta PEC se ha desarrollado una pequeña demo de una llama y un río de lava, ambos ejemplos muy simples y poco realistas, pero que reflejan un manejo básico de las funciones OpenGL en C++ y GLSL en los shaders. OpenGL es un poco oscura a la hora de iniciarse, pero se vuelve muy potente cuando poco a poco se van permeando sus distintas abstracciones y filosofía interna, como la renderización en buffers, pudiendo usar exactamente el mismo código para dibujar en pantalla o en una textura que puede usarse posteriormente en la pantalla logrando efectos muy llamativos sin demasiada carga lógica.

También se ha aprovechado un poco el framework creado para la PEC anterior, reutilizando la estructura de renderización para organizar los elementos del programa. Esta reorganización se ha hecho de forma básica porque el framework de la PEC anterior estaba diseñado para usarse con superficies, mientras que esta PEC se basa en ventanas OpenGL, pero se ha podido adaptar a este requisito sin mucho esfuerzo.

References

[1] https://stegu.github.io/webgl-noise/webdemo/

[2] https://www.shadertoy.com/view/WdKcRt