Análisis y comparativa de un sistema principal Postprocesamiento

Jordi Borraz
Nil Muns
Ricard Ruiz
Héctor Vergel

Índex

Introducción

LÖVE fue lanzado el 13 de enero de 2008 por la misma compañía llamada *LOVE*. Este es un motor gratuito con código abierto, es multiplataforma y fue publicado bajo la licencia conocida como *zlib*, una licencia para hacer videojuegos 2D. El framework está escrito en C++, pero utiliza Lua como lenguaje de *script*.

Algunos juegos creados con LÖVE son: *BLUE REVOLVER*, *Move Or Die* o *Warlock's Tower*. Además, este motor está disponible para estas plataformas: Microsoft Windows, Linux, macOS, iOS, y Android.

Requisitos del Sistema

Tomando como punto inicial el lenguaje de programación de Lua (el utilizado por LÖVE), este es un lenguaje que tiene unos requisitos muy bajos para poder ejecutarse, como volverá a mencionarse más en detalle posteriormente en este documento. Empleando un lenguaje vulgar, podría decirse que "lo tira hasta mi lavadora" debido a que los requisitos mínimos para poder utilizar Lua son los siguientes:

- Sistema operativo: cualquier sistema que soporte la API de ANSI C.
- Procesador: cualquiera compatible con x86 y x86-64.
- RAM: 64 Mb.
- Espacio en disco: 1 Mb.

Aunque todos estos requisitos son muy bajos, cuando hablamos de LÖVE tenemos que tener en cuenta que las necesidades aumentan sutilmente. Sin embargo, no es necesario tener un gran poder computacional para ejecutarlo. Los requisitos mínimos de LÖVE son los siguientes:

- Sistema operativo: Windows, MacOS, Linux, Android, iOS.
- Procesador: cualquiera compatible con x86 y x86-64.
- RAM: 512 Mb.
- Espacio en disco: 20 Mb.

- Tarjeta gráfica: cualquier tarjeta compatible con OpenGL 2.1 o posterior.
- Versión de Lua: 5.1, 5.2 o 5.3.
- Biblioteca de tiempo de ejecución de Microsoft Visual C++ 2015.

Implementación del Sistema

El postprocesado que implementa LÖVE es realmente una librería de *shaders* que crean varios efectos de postprocesado, algunos de ellos los veremos en el apartado posterior de Ejemplos de Uso. Esta librería se llama "Moonshine" y está totalmente escrita en Lua, aunque es importante mencionar también a GLSL (OpenGL Shading Language), ya que los *shaders* que utiliza la librería son los que implementa LÖVE en su módulo de "love.graphics" y están escritos con GLSL.

Cabe destacar que LÖVE realiza algunas modificaciones en algunos puntos de entrada del código de GLSL para mayor comodidad. Véase en la imagen:

GLSL	LÖVE shader language
sampler2D	Image
sampler2DArray	ArrayImage
samplerCube	CubeImage
sampler3D	VolumeImage
texture2D(tex, uv) (in GLSL 1)	Texel(tex, uv)
texture(tex, uv) (in GLSL 3)	Texel(tex, uv)
float	number (deprecated)
uniform	extern (deprecated)

Ejemplos de Uso

Actualmente, la librería de postprocesamiento "Moonshine" para LÖVE tiene varias funciones, las cuales modifican con efectos las imágenes que posteriormente se verán en la consola de LÖVE. Estos efectos se llaman mediante funciones en el código del programa, empleando variables para especificar los efectos que tendrán estas sobre las imágenes y vídeo del programa. Algunas de las funciones que permite la librería son las siguientes:

BoxBlur: esta función básicamente crea un efecto de borrosidad sobre la imagen.

- Se llama con la función ***moonshine.effects.boxblur***.
- Tiene 3 parámetros que son: **radius** general, en el que se ve el efecto y, si se quiere hacer que la X y la Y sean diferentes, existen las variables **radiusX** y **radiusY**.

```
return function(moonshine)
  local radius_x, radius_y = 3, 3
  local shader = love.graphics.newShader[[
    extern vec2 direction;
    extern number radius;
    vec4 effect(vec4 color, Image texture, vec2 tc, vec2 _) {
      vec4 c = vec4(0.0);

      for (float i = -radius; i <= radius; i += 1.0)
      {
        c += Texel(texture, tc + i * direction);
      }
      return c / (2.0 * radius + 1.0) * color;
    }]]
```

Chromasep: esta función crea un efecto cromático dentro de la imagen.

- Se llama con la función **moonshine.effects.chromasep.**
- Para llamar a la función es necesario usar dos parámetros, los cuales son: el **angle** en radianes y el propio **radius** de efecto de la función.

```
return function(moonshine)
  local shader = love.graphics.newShader[[
    extern vec2 direction;
    vec4 effect(vec4 color, Image texture, vec2 tc, vec2 _)
    {
      return color * vec4(
        Texel(texture, tc - direction).r,
        Texel(texture, tc).g,
        Texel(texture, tc + direction).b,
        1.0);
    }]]

  local angle, radius = 0, 0
  local setters = {
    angle  = function(v) angle  = tonumber(v) or 0 end,
    radius = function(v) radius = tonumber(v) or 0 end
  }
```

Colorgradesimple: esta función sirve para cambiar la tonalidad de los canales de color.

- Se llama con la función **monshine.effects.colorgradesimple**.
- Para usar la función es necesario usar una variable de tipo **Array** para crear una lista con valores numéricos para definir la tonalidad de los colores que quieres cambiar.

```
return function(moonshine)
  local shader = love.graphics.newShader[[
    extern vec3 factors;
    vec4 effect(vec4 color, Image texture, vec2 tc, vec2 _) {
      return vec4(factors, 1.0) * Texel(texture, tc) * color;
    }]]

  local setters = {}
```

Desaturate: esta función sirve para desaturar y de alguna forma teñir la imagen. Hace más o menos el mismo efecto que la función anterior pero en este caso tiene un variable de fuerza, la cual puede editar aún más el efecto de cambiar el color.

- Se llama con la función **moonshine.effects.desaturate**.
- Las variables necesarias para llamar a la función son: una **Array** para establecer los colores deseados y una variable llamada **Strengh** que es de tipo numérica y que tiene que tener un valor de entre 0.1 y 0.5.

```
return function(moonshine)
  local shader = love.graphics.newShader[[
    extern vec4 tint;
    extern number strength;
    vec4 effect(vec4 color, Image texture, vec2 tc, vec2 _) {
      color = Texel(texture, tc);
      number luma = dot(vec3(0.299, 0.587, 0.114), color.rgb);
      return mix(color, tint * luma, strength);
    }]]
```

DMG: esta función sirve para aplicar unos colores ya preestablecidos por el autor de "Moonshine". Estos colores hacen referencia a los colores de la Gameboy.

- ❖ Se llama con la función **moonshine.effects.dmg**.
- ❖ Necesita como parámetro de entrada una variable **Array**, aunque el hecho especial de esta función reside en que el autor ya tiene creada 7 Arrays que él las llama "Palette", que pueden ser usadas sin tener que crear tu propia paleta de colores, son las siguientes:
  - ➢ 1. Default/ Dark Yellow / Light Yellow/ Green/ Greyscale/ Star Bw/ Pocket

```
local lookup_palette = function(name)
  for _,palette in pairs(palettes) do
    if palette.name == name then
      return palette
    end
  end
end

local is_valid_palette = function(v)
  -- Needs to match: {{R,G,B},{R,G,B},{R,G,B},{R,G,B}}
  if #v ~= 4 then return false end

  for i = 1,4 do
    if type(v[i]) ~= "table" or #v[i] ~= 3 then return false end
    for c = 1,3 do
      if type(v[i][c]) ~= "number" then return false end
      local x = v[i][c]
      if x > 1 then x = x / 255 end
      if x < 0 or x > 1 then return false end
      v[i][c] = x
    end
  end
  return true
end
```

```
name = "green",
colors = {
  {8/255,56/255,8/255},
  {48/255,96/255,48/255},
  {136/255,168/255,8/255},
  {183/255,220/255,17/255}
}
```

```
name = "greyscale",
colors = {
  {56/255,56/255,56/255},
  {117/255,117/255,117/255},
  {178/255,178/255,178/255},
  {239/255,239/255,239/255}
}
```

```
name = "dark_yellow",
colors = {
  {33/255,32/255,16/255},
  {107/255,105/255,49/255},
  {181/255,174/255,74/255},
  {255/255,247/255,123/255}
}
```

```
name = "stark_bw",
colors = {
  {0/255,0/255,0/255},
  {117/255,117/255,117/255},
  {178/255,178/255,178/255},
  {255/255,255/255,255/255}
}
```

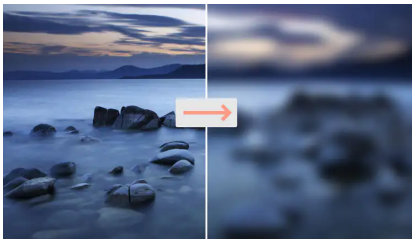
```
name = "light_yellow",
colors = {
  {102/255,102/255,37/255},
  {148/255,148/255,64/255},
  {208/255,208/255,102/255},
  {255/255,255/255,148/255}
}
```

```
name = "pocket",
colors = {
  {108/255,108/255,78/255},
  {142/255,139/255,87/255},
  {195/255,196/255,165/255},
  {227/255,230/255,201/255}
}
```

GaussianBlur: esta función usa la fórmula de Gaussian para calcular a qué píxeles hay que añadir *blur* para crear este efecto.

- Para llamar la función se usa el comando **moonshine.effects.gaussianBlur**.
- Para invocar la función es necesario un parámetro de tipo **int** que tiene como nombre **Sigma**.

```
return function(moonshine)
  local shader

  local setters = {}
  setters.sigma = function(v)
    shader = resetShader(math.max(0,tonumber(v) or 1))
  end
```



Glow: esta función sirve para crear un efecto de luminosidad en un punto determinado de la pantalla.

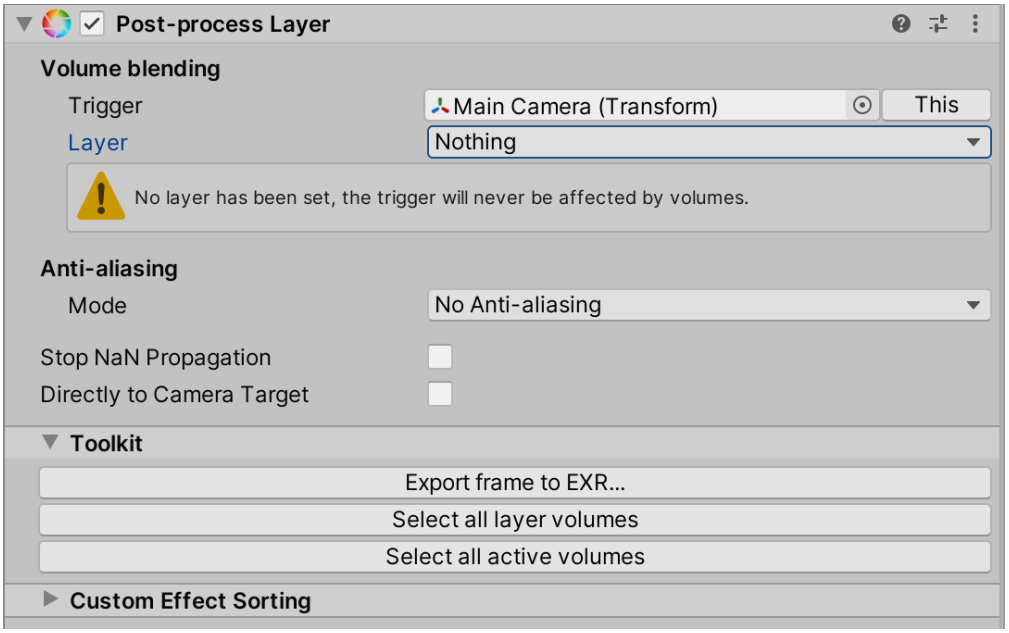
- Para llamar la función se usa el comando **moonshine.effects.glow**.
- En cuanto a parámetros, la función Glow necesita dos variables: la **min_luma**, que sirve para establecer la mínima iluminación (debiendo ser un número de entre 0 a 1) y como segunda variable se ha de poner el valor de la variable **Strength**.

```
  -- 1st pass: draw scene with brightness threshold
  love.graphics.setCanvas(front)
  love.graphics.clear()
  love.graphics.setShader(threshold)
  love.graphics.draw(scene)

  -- 2nd pass: apply blur shader in x
  blurshader:send('direction', {1 / love.graphics.getWidth(), 0})
  love.graphics.setCanvas(back)
  love.graphics.clear()
  love.graphics.setShader(blurshader)
  love.graphics.draw(front)

  -- 3nd pass: apply blur shader in y and draw original and blurred scene
  love.graphics.setCanvas(front)
  love.graphics.clear()

  -- original scene without blur shader
  love.graphics.setShader()
  love.graphics.setBlendMode("add", "premultiplied")
  love.graphics.draw(scene) -- original scene

  -- second pass of light blurring
  blurshader:send('direction', {0, 1 / love.graphics.getHeight()})
  love.graphics.setShader(blurshader)
  love.graphics.draw(back)

  -- restore things as they were before entering draw()
  love.graphics.setBlendMode("alpha", "premultiplied")
  scene = back
end
```

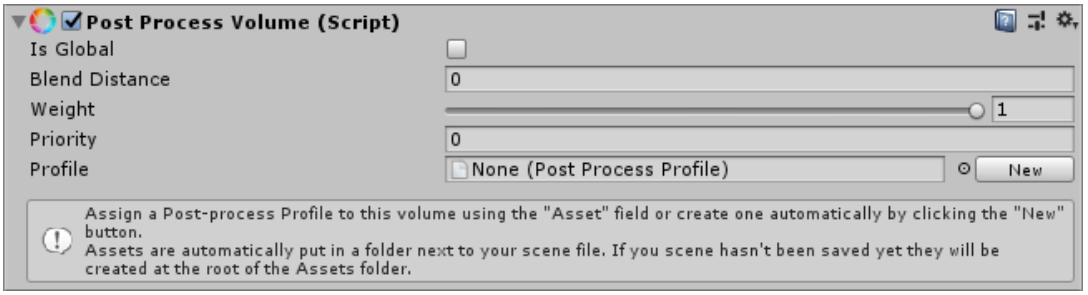
Comparativa LÖVE vs Unity

Después de ver algunos de los efectos de postprocesado que podemos utilizar haciendo uso de la librería “Moonshine" explicada anteriormente, en esta sección vamos a exponer la forma en la que los mismos efectos se pueden recrear en Unity.

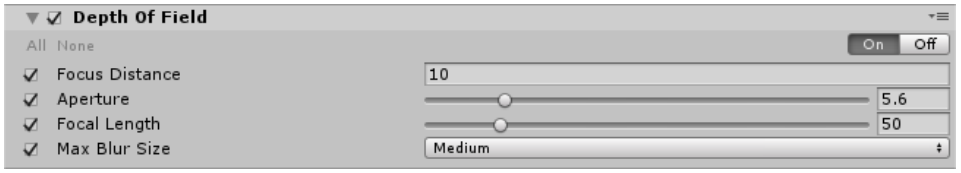
Cabe mencionar que en Unity no disponemos de todos los efectos al mismo nivel, por así determinarlo, que con la librería “Moonshine”, ya que Unity nos facilita las herramientas para crearlos, pero de una forma más versátil y completa.



Partiendo de la creación y configuración mediante el uso del *Editor* de *Layer* de postprocesado y acabando con el uso de los efectos añadidos al componente “Volume”.



Teniendo como referencia los efectos *BoxBlur* y *GuassianBlur* de LÖVE mencionados anteriormente, se pueden recrear utilizando la herramienta de *Motion Blur* y *Depth of Field* de Unity.

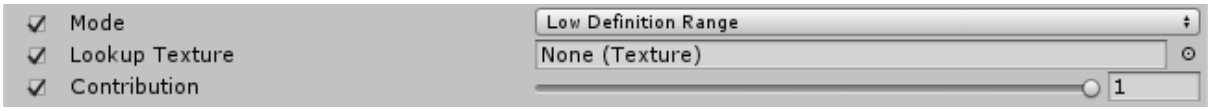


Como se puede observar en la configuración rápida del componente "Volume", estos valores pueden ser modificados para recrear el efecto de *Blur* sin ningún problema, facilitando el uso y la incorporación en el proyecto.

Esta facilidad que incorpora Unity para crear de forma sencilla y rápida los efectos de postprocesado es muy potente y permite a un gran abanico de roles trabajar sobre su funcionamiento, como sería el caso de diseñadores o artistas, no limitando así su uso a los programadores.

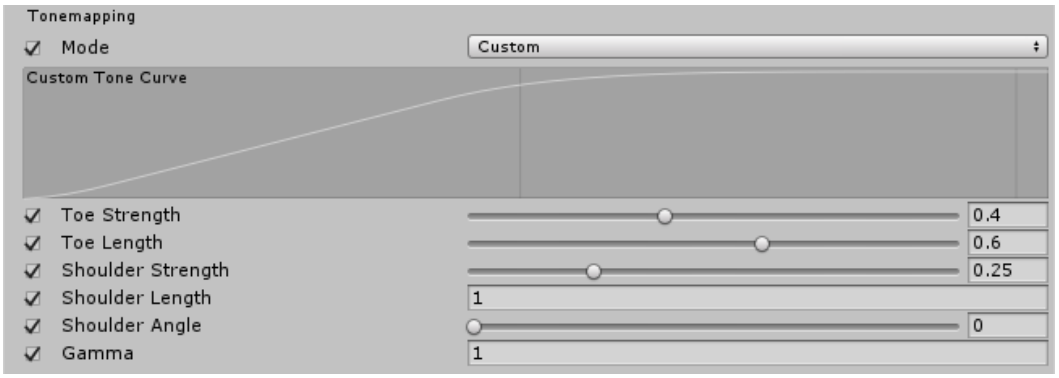
Lo mismo ocurre con los efectos de *Chromasep*, *Colorgradesimple*, *Desaturate* y *DMG*, que son efectos de color muy específicos creados en la librería y ya adaptados para su uso. Sin embargo, en Unity de forma "core" tenemos acceso a la herramienta de *Color-Grading* de postprocesador que nos permite replicar estos efectos e incluso crear muchos más, siendo más complejos y completos.

Este componente del "Volume" tiene unas configuraciones globales y después se pueden añadir diferentes subcomponentes para conseguir el efecto o efectos deseados, teniendo la simplicidad de crear desde una desaturación básica hasta un complejo arco iris de colores.



Los principales subcomponentes de la herramienta de *Color-Grading* son los siguientes:

- *Tonemapping*

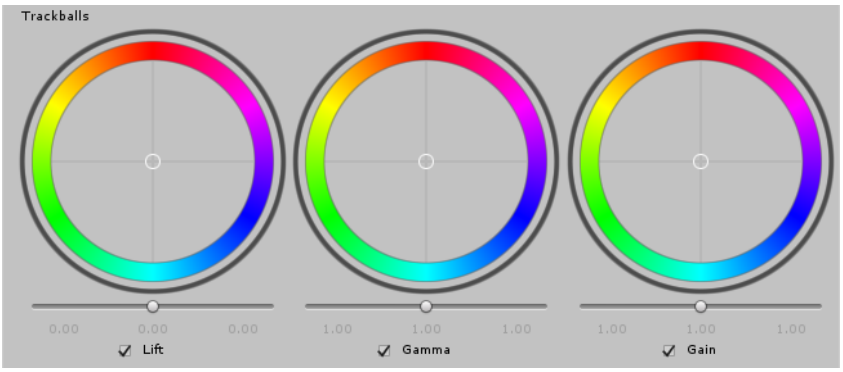


- *White Balance*

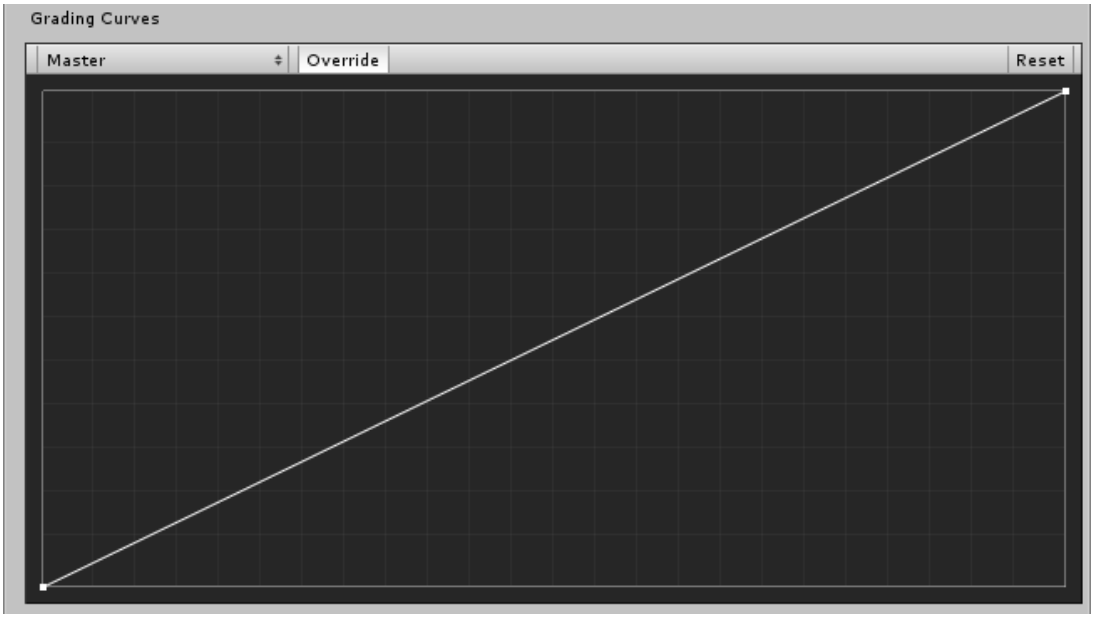
- *Tone*

- *Channel Mixer*

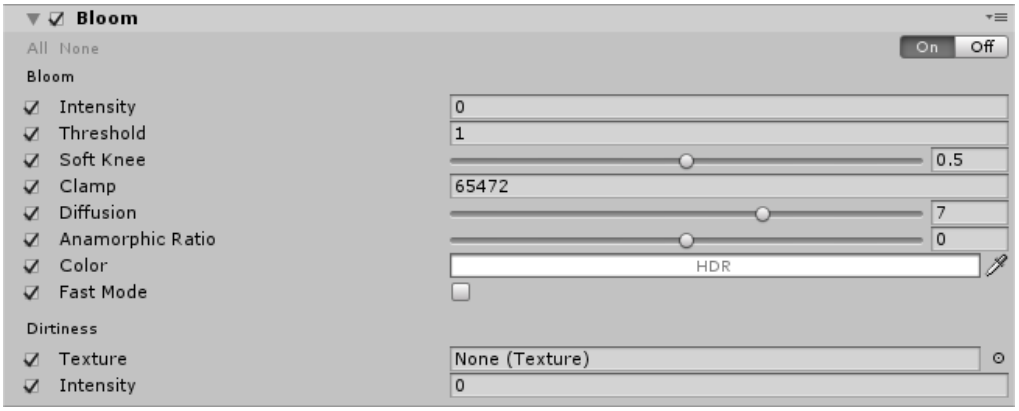
- *Trackballs*



- *Grading Curves*



Por otra parte, *Bloom* es el efecto de postprocesado de Unity que permite recrear el *Glow* de "Moonshine", como también se ha mencionado en el resto de herramientas, con una gran variedad de posibilidades mucho mayor de las que ofrece "Moonshine", pero con la necesidad de tener que trabajar más sobre el efecto.



Todos estos componentes son de un uso muy sencillo e intuitivo. Sin embargo, se necesita conocimiento y trabajo para conseguir grandes y buenos resultados.

Para finalizar con esta comparativa, también hay que mencionar que estos efectos, junto a muchos otros, también pueden recrearse e instanciar mediante el uso de código, como sería en el caso exclusivo de LÖVE, con la utilización de "Quick Volumes".

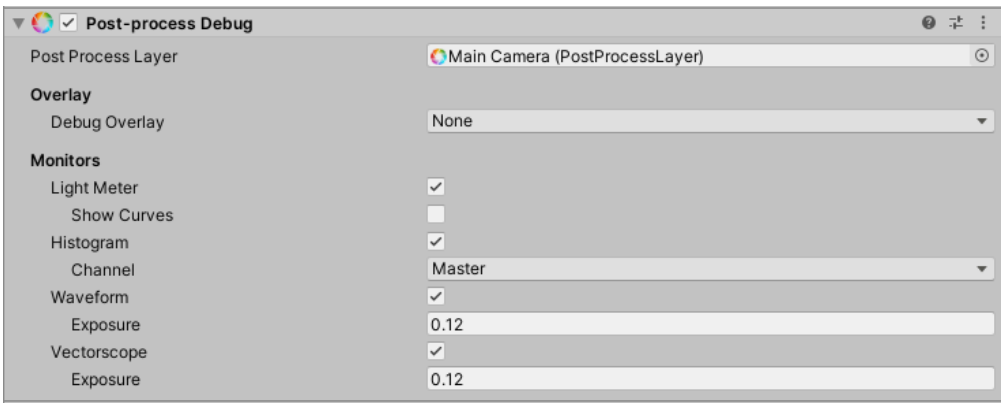
```
[
using UnityEngine;
using UnityEngine.Rendering.PostProcessing;
public class VignettePulse : MonoBehaviour
{
   PostProcessVolume m_Volume;
   Vignette m_Vignette
   void Start()
  {
      // Create an instance of a vignette
       m_Vignette = ScriptableObject.CreateInstance<Vignette>();
       m_Vignette.enabled.Override(true);
       m_Vignette.intensity.Override(1f);
      // Use the QuickVolume method to create a volume with a priority of 100, and assign the
vignette to this volume
       m_Volume = PostProcessManager.instance.QuickVolume(gameObject.layer, 100f, m_Vignette);
   void Update()
  {
       // Change vignette intensity using a sinus curve
        m_Vignette.intensity.value = Mathf.Sin(Time.realtimeSinceStartup);
  }
   void OnDestroy()
  {
       RuntimeUtilities.DestroyVolume(m_Volume, true, true);
  }
}
```

En esta imagen se puede ver la creación de un objeto “Vignette” y su aplicación utilizando el acceso al *scripting* de Unity “PostProcessManager” para crear un efecto de postprocesador de “Vignette”.

También cabe mencionar que, a diferencia de LÖVE, con el sistema de postprocesado de Unity podemos realizar una comprobación de función mucho más exhaustiva y completa mirando los diferentes monitores de uso como son:

- *Light Meter, Histogram, Waveform* y *Vectorscope*.

Esta herramienta también es un componente del propio sistema de Unity.



Conclusión

Después de realizar el trabajo y hacer un análisis exhaustivo de la parte de postprocesado de estos dos motores, hemos llegado a las siguientes conclusiones:

El sistema de postprocesado de LÖVE es un sistema que nos permite realizar menos postprocesado y no hay tantas opciones como en Unity, aunque hay librerías más desarrolladas en LÖVE que en Unity, como por ejemplo la “Moonshine”, que nos permite modificar con efectos las imágenes que se verán después en la consola del motor. Cabe destacar que todo en LÖVE se tiene que hacer por código, cosa que lo complica mucho más.

Si hablamos de Unity, podemos decir que su sistema de postprocesado es más complejo, Unity nos brinda la posibilidad de recrear las librerías que nos ofrece LÖVE con sus herramientas, aunque estas sean recreadas de una forma más genérica, pero a su vez nos permite crear más efectos de postprocesado. También destacar que el hecho de que todo el postprocesado de Unity no sea hecho por código facilita mucho más su uso.

Para concluir podemos decir que ambos sistemas son útiles para realizar postprocesado, el de LÖVE es mucho más complicado de realizar, ya que todo tiene que ser por código, mientras que en Unity todo es un poco más sencillo porque nada va por código y el motor en sí ya nos proporciona las herramientas necesarias para realizar los efectos de postprocesado.

Bibliografía

LÖVE - Free 2D Game Engine. (n.d.). Recupeardo de https://love2d.org/

Shader - LOVE. (s. f.). Recuperado de https://love2d.org/wiki/Shader

V. (s. f.). GitHub - vrld/moonshine: Postprocessing effect repository for LÖVE. Recuperado de https://github.com/vrld/moonshine

Bloom | Post Processing | 3.2.2. (s. f.). Recuperado de http://bit.ly/3ktH0NA

Color Grading | Post Processing | 3.2.2. (s. f.-a). Recuperado de http://bit.ly/3xVlKni

Controlling effects using scripts | Post Processing | 3.2.2. (s. f.). Recuperado de http://bit.ly/3Sz47TN

Debugging Post-processing effects | Post Processing | 3.2.2. (s. f.). Recuperado de http://bit.ly/3xVROr6

Depth of Field | Post Processing | 3.2.2. (s. f.). Recuperado de http://bit.ly/3EF9FWX

Getting started with post-processing | Post Processing | 3.1.1. (s. f.). Recuperado de http://bit.ly/3IWMdHl

Motion Blur | Post Processing | 3.2.2. (s. f.). Recuperado de http://bit.ly/3xVm45u