# Graphics Shaders

## GLSL (OpenGL Shading Language)

## Post Processing

Ana Gil Luezas
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

## Postprocesado (full screen post-processing effects)

Una vez renderizada la escena se aplican procesos para diversos efectos: filtros, motion blur, visión nocturna, …

1. Se renderiza la escena en una textura (*Render Target Texture*) del tamaño del puerto de vista y, antes de mostrarla (*Frame Buffer Object*)
2. Se aplica un Pixel shader: se renderiza un rectángulo que ocupa todo el puerto de vista (fullscreen quad) con coordenadas de textura para recubrirlo con la textura de la escena (del apartado 1). Así, el proceso de rasterización generará un fragmento por cada pixel del puerto de vista.

   El vertex shader pasa al fragment shader las coordenadas de textura sin transformar y las coordenadas de los vértices en el espacio de recorte.

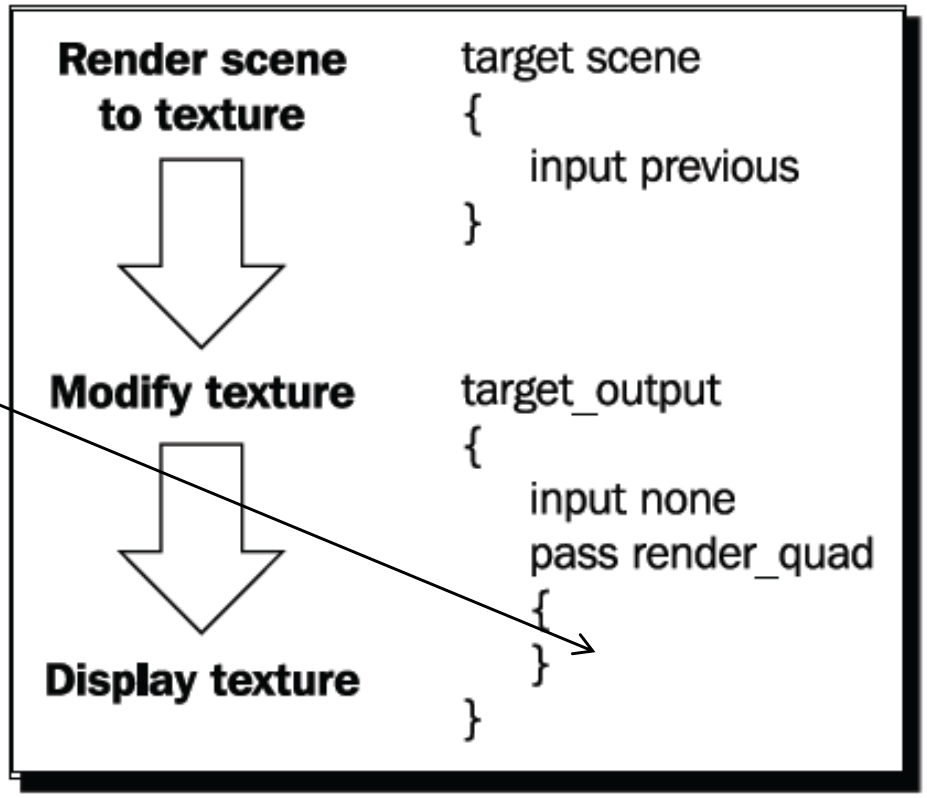
   El fragment shader (que tendrá desactivado el depth buffer) realiza el efecto deseado a la textura generada por la escena.

- En Ogre -> Compositor scripts (archivos con extensión compositor)

  Ogre se encarga de renderizar la escena en una textura y del rectángulo para realizar el postproceso definido en el material.

- El compositor incluye el material para el quad, con el que se realizará el postprocesado a la imagen resultante de renderizar la escena

- Para aplicar un compositor hay que asociarlo a un puerto de vista (vp):




```
CompositorManager::getSingleton().addCompositor( vp,
                                        "Nombre del compositor");
```

# Ogre: Compositor Scripts

- Compositor `// archivo IG2.compositor`

```
compositor Luminance {   // nombre
  technique {
     // Temporary Textures for use in subsequent target passes
     // Dimensions based on the physical dimensions of the viewport
     // to which the compositor is attached
    texture RTT0 target_width target_height PF_R8G8B8A8
    target RTT0 { // Render Target Texture
      input previous //start with the previous content of the viewport
    }     // from original scene or from previous compositor in the chain
    target_output  { // Final render output
      input none // start without initializing
      pass render_quad {  // Render a fullscreen quad with
          material LuminancePS // the luminancePS material
          input 0 RTT0     // texture unit 0 <-> texture RTT0
       }     // para usar la textura en el fragment shader del material
    }
} }
```

# Ejemplo

- Fragment shader -> escala de grises

-> colorGris = (lum, lum, lum) // 0<=lum<=1
3*lum == color.r + color.g + color.b
lum = color.r / 3. + color.g / 3. + color.b / 3.

```
#version 330 core    // archivo LuminancePS.glsl
in vec2 vUv0;
uniform sampler2D RTT0; // textura con la escena
out vec4 fFragColor;
// weight vector for luminance (de suma 1)
const vec3 WsRGB = vec3(0.2125, 0.7154, 0.0721);
void main() {
  vec4 sceneColor = texture(RTT0, vUv0);
  float lum = dot(vec3(sceneColor), WsRGB);
  fFragColor = vec4(lum, lum, lum, sceneColor.a);
}                    // 0<=lum<=1
```

- Material `// LuminancePS`

```
 vertex_program RenderQuadVS glsl
 {
   source Ejemplo2VS.glsl // podemos reutilizar
   …
 }

fragment_program LuminancePS glsl
{
   source LuminancePS.glsl
   default_params  {
      param_named RTT0 int 0    // textura con la escena
   }
}
                                                    // ->
```

# Ogre: Material Scripts

```
// ->
material LuminancePS  {
  technique  {
    pass  {
      depth_check off  // desactivar el depth-buffer
      depth_write off
      vertex_program_ref RenderQuadVS  {
       }
      fragment_program_ref LuminancePS  {
      }
      texture_unit RTT0
      {
         // sin imagen de archivo -> previous render target
        filtering none  // tiene la resolución del viewport
      }
    }
} }
```

# Ogre Compositor

- Añadir a un viewport el compositor `Luminance` (en setupScene)

```
CompositorManager::getSingleton().addCompositor(vp,
                    "Luminance");

CompositorManager::getSingleton().setCompositorEnabled(vp,
                    "Luminance", true);
```

- Se pueden concatenar varios compositors: Según se van añadiendo se van concatenando. `input previous` -> escena o anterior compositor

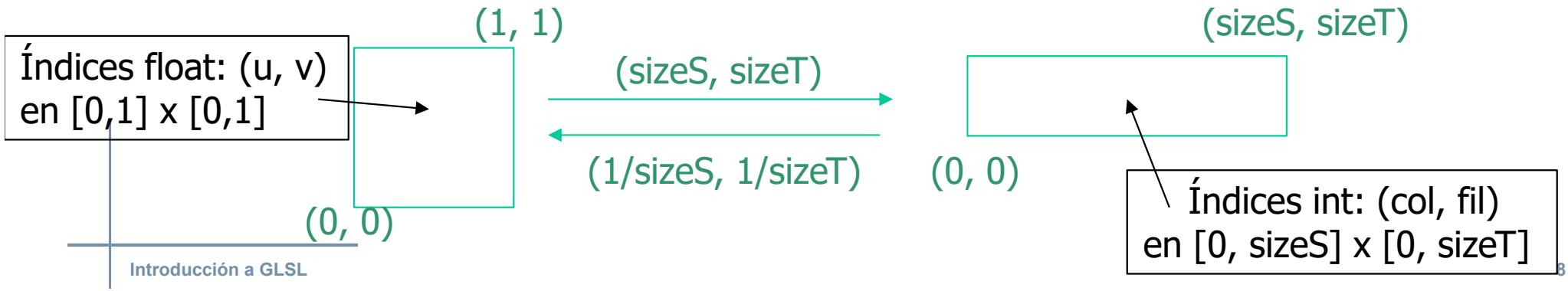
```
CompositorManager::getSingleton().addCompositor(vp,
                                    "Night Vision");

CompositorManager::getSingleton().setCompositorEnabled(vp,
                                    "Night Vision", true);
```

En `Luminance` (el primero) `input previous` es la escena
En `Night Vision` (el segundo) `input previous` es Luminance

## Operaciones con los texels vecinos (textureSize)

Para acceder a los vecinos de un téxel tenemos que calcular el desplazamiento en horizontal y en vertical:

```
vec2 uv; // coordenadas de textura en [0, 1]
vec4 texelUV = texture(colores, uv); // téxel en uv
     ¿vecinos al téxel de coordenadas de textura uv?
ivec2 texSize = textureSize(RTT0, 0); // dimensiones
float sizeS = texSize.s; // ancho
float sizeT = texSize.t; // alto
float incS = 1/sizeS;  // inc. horizontal
float incT = 1/sizeT;  // inc. vertical
```



## Operaciones con los texels vecinos (textureSize)

Para acceder a los vecinos de un téxel tenemos que calcular el desplazamiento en horizontal y en vertical: 1./sizeS y 1./sizeT

```
vec2 uv; // coordenadas de textura en [0, 1]
vec4 texelUV = texture(colores, uv); // téxel en uv
```

¿vecinos al téxel de coordenadas de textura uv?

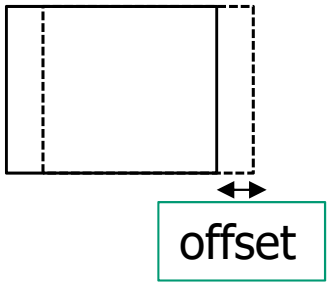
```
ivec2 texSize = textureSize(RTT0, 0); // dimensiones
float incS = 1./ texSize.s;  // inc. horizontal
float incT = 1./ texSize.t;  // inc. vertical
vec2 uvS = vec2(uv.s+incS, uv.t);  // inc. horizontal
vec2 uvT = vec2(uv.s, uv.t+incT);  // inc. vertical
vec2 uvD = vec2(uv.s+incS, uv.t+incT); // inc. Diagonal
vec4 texelN = texture(RTT0, uvN); // téxel vecino … !!
…
```

¿en [0,1]x[0,1]?

tex_address_mode:
wrap / clamp / …

- Fragment shader -> ejemplo visión doble (mezcla de una textura con ella misma desplazada)

```
#version 330 core
in vec2 vUv0; // en [0, 1]
uniform sampler2D RTT0;
out vec4 fFragColor;
const float OFFSET = 10.; // desplazamiento
void main() {
 vec4 texel = texture(RTT0, vUv0);  // téxel en vUv0
 ivec2 texSize = textureSize(RTT0, 0); // dimensiones
 vec2 incOf = OFFSET / float(texSize.s); // desplazamiento
 vec2 uvOf = vUv0 + vec2(incOf, 0);  // uvOf.s en [0, 1+...] !?
 vec4 texelD = texture(RTT0, uvOf); // téxel en uvOf !? ->
 if(uvOf.s > 1) texelD = texel;  // -> borde sin mezcla
 fFragColor = mix(texel, texelD, 0.5);
}
```

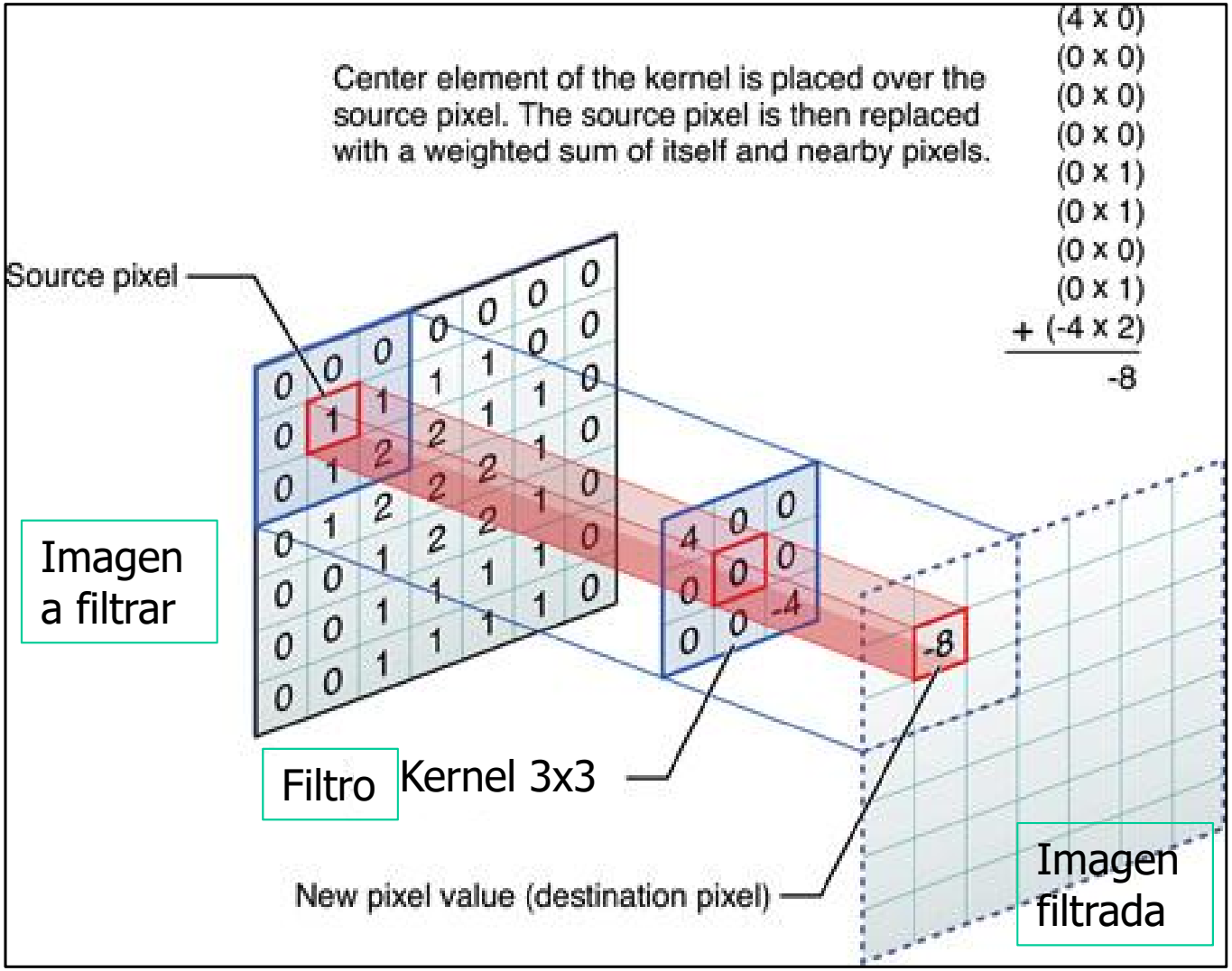



tex_address_mode:
wrap / clamp / ...

## ❑ Aplicación de un filtro basados en un kernel o matriz

Un kernel es una matriz de pesos que determina el efecto en función del entorno de cada pixel.

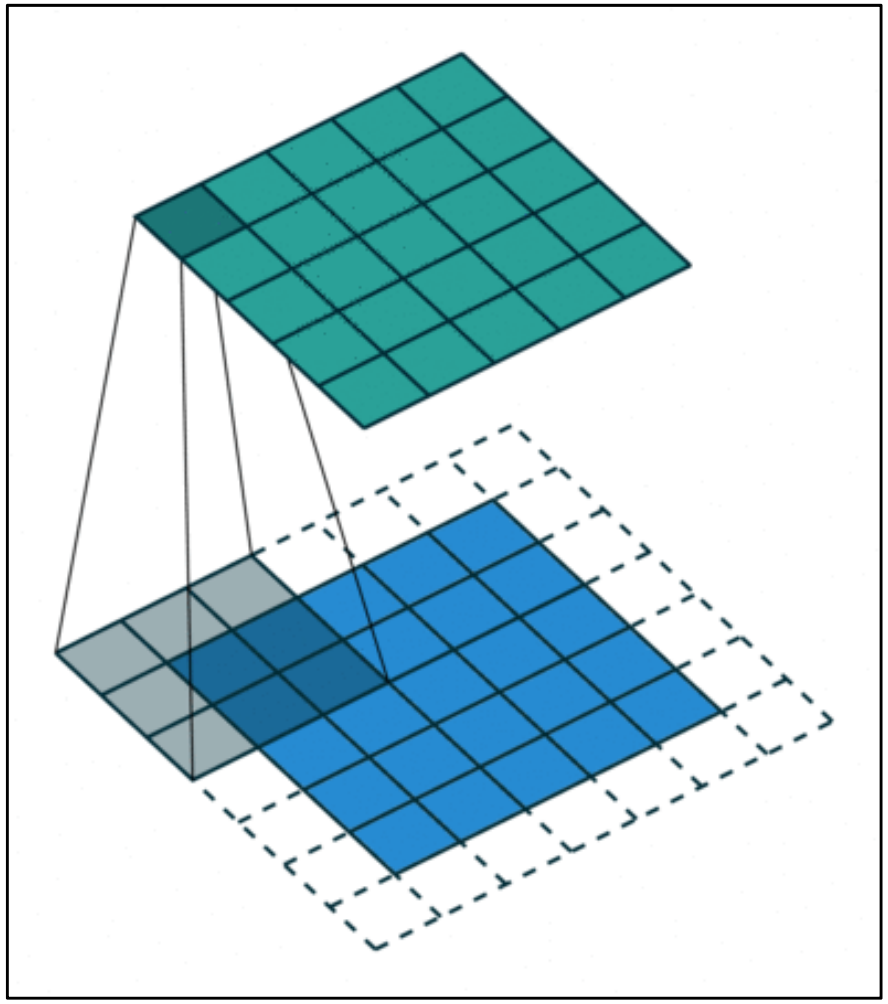
Para aplicar un filtro a una imagen se realiza un barrido del kernel sobre la imagen.



## Aplicación de un filtro

En los límites de la imagen se aplica un tratamiento especial: se asume un marco exterior de ceros o se repiten los valores del borde.

Configurar tex_address_mod de la nidad de textura



Ejemplos: Blur, Sharpen, Edge enhance, Edge detect, Emboss

- Fragment shader -> Blur kernel (difuminar)

```
void main() {
  ivec2 texSize = textureSize(RTT0, 0);
  float incS = 1. / float(texSize.s);
  float incT = 1. / float(texSize.t);
  vec2 incUV[9] = vec2[]( // incrementos para acceder a
                  vec2(-incS,  incT), // top-left
                             …      , // top-center
                             …      , // top-right
                             …      , // center-left
                             …      , // center-center
                             …      , // center-right
                             …      , // bottom-left
                             …      , // bottom-center
                  vec2( incS, -incT)  // bottom-right
                  ); // ->
```

- Fragment shader -> Blur kernel (difuminar)

```
// ->
float kernel[9] = float[](
                    1./16, 2./16, 1./16,
                    2./16, 4./16, 2./16,
                    1./16, 2./16, 1./16 );

vec3 color = vec3(0.0);
for(int i = 0; i < 9; i++) {
    color += … ;
}
fFragColor = vec4(color, 1.0);
}
```

kernel = $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16$

Los kernel pueden tener distintas dimensiones, y si tienen muchos ceros no se utilizan arrays

- vec4 gl_FragCoord coordenadas del fragmento en Screen space (origin at lower-left)

  (x, y, z, 1/w):

  z es la profundidad del fragmento, para el depth buffer

  (x, y) son las coordenadas del píxel

  Podemos hacer un casting para trabajar en valores int (filas y columnas):

```
int col = int(gl_FragCoord.x);
int row = int(gl_FragCoord.y);
if (row < 5 && col < 25) color = (1, 0, 0, 1);
else if (col < 5 && row < 25) color = (0, 1, 0, 1);
else color = (0, 0, 1, 1);
```