



OGRE 3D

Textures

Multitexturing and RenderTextures

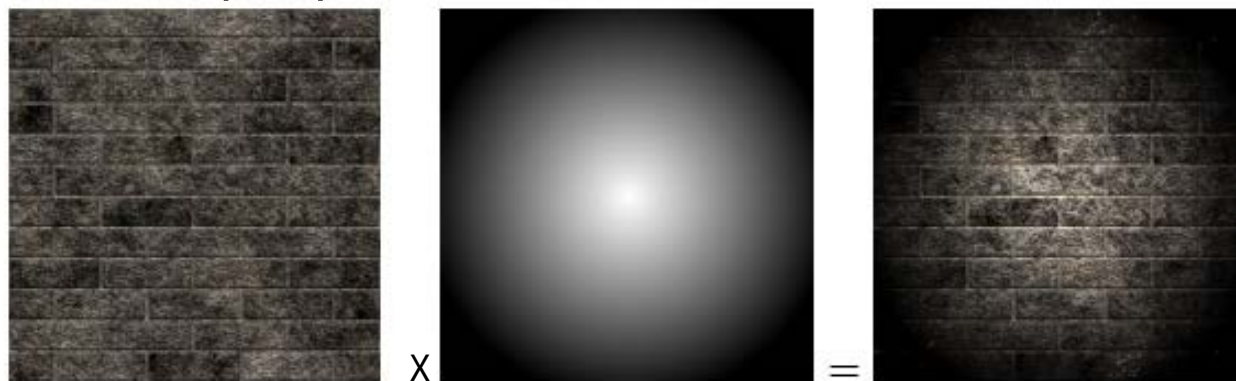
<http://www.ogre3d.org>

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

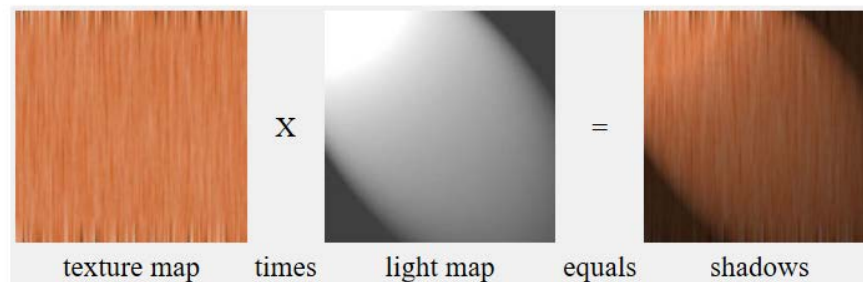
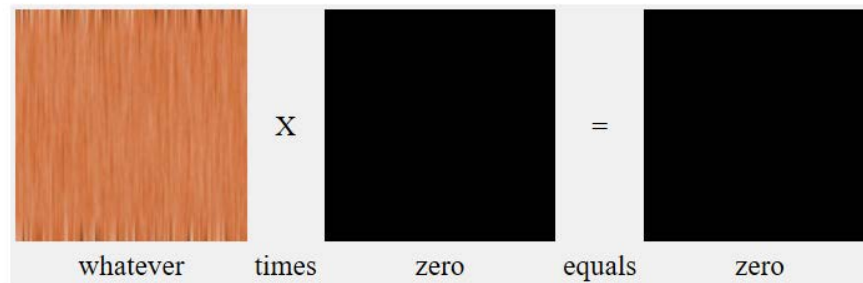
- ❑ **Texture Mapping**: asignar coordenadas de textura (u,v) a una malla para obtener el color a partir de una imagen y las coordenadas (u,v)
- ❑ **Multitexturing**: se pueden utilizar varias unidades de texturas simultáneamente para generar efectos (iluminación estática, sombras, ...) mezclando los colores de las distintas imágenes (y con el color obtenido por la iluminación).

Los colores de las texturas se mezclan entre si: **add**, **modulate**, **blending**

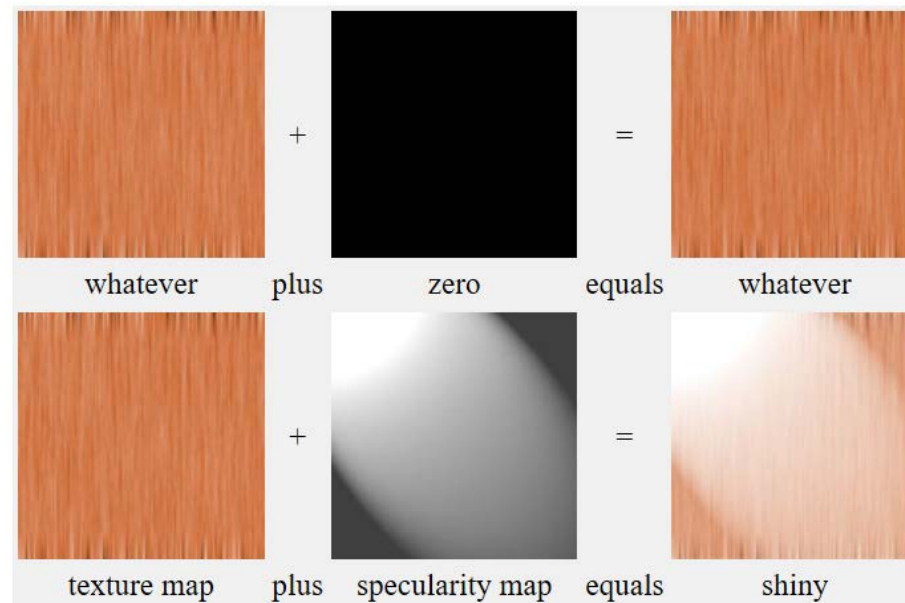
Ejemplo: **Light map** (imagen en grises que se utiliza para simular la intensidad de luz por pixel)



❑ **Modulate:** El producto oscurece la imagen (tiende a negro)



- ❑ **Add:** La suma tiende a blanco (aumenta el brillo)



```
material exampleMT1
```

```
{  
  technique  
  {  
    pass  
    {  
      ambient 0.5 0.5 0.5 // coeficientes de reflexión para Luz ambiente  
      diffuse 1.0 1.0 1.0 // coefs. de reflex. para la componente difusa de la Luz  
      texture_unit // Texture unit 0  
      {  
        texture wibbly.jpg // nombre del archivo de la imagen  
        // color_op modulate // valor por defecto: Los colores se multiplican con ...  
      }  
      texture_unit // Texture unit 1 (this is a multitexture pass)  
      {  
        texture wobbly.png // nombre del archivo de la imagen  
        colour_op add // Los colores de esta imagen se suman con ...  
      }  
    }  
  }  
}
```

Mezcla de colores resultante:

$Luz \times TU0 + TU1$

¿Todas las unidades de textura utilizan las mismas coordenadas?

- ❑ En una **mall**a se pueden especificar diferentes asignaciones de coordenadas de textura para utilizarlas en distintas unidades de textura:

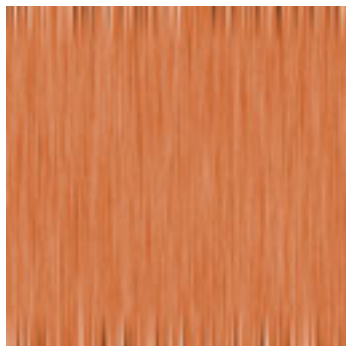
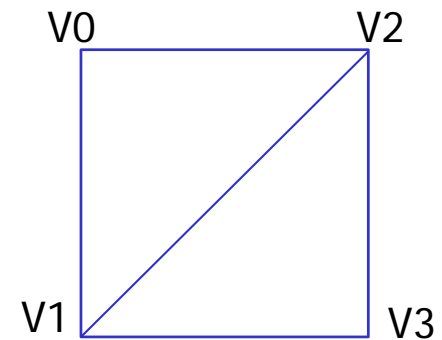
Vertex: $(-1, 1, 0), (-1, -1, 0), (1, 1, 0), (1, -1, 0)$

Normal: ...

TexCoord0: $(0, 1), (0, 0), (1, 1), (1, 0)$

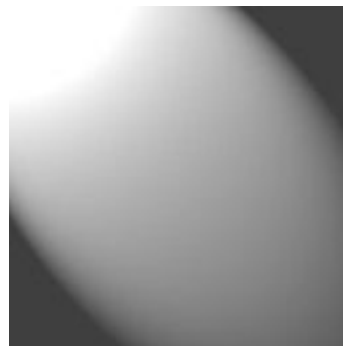
TexCoord1: $(1, 1), (0, 1), (1, 0), (0, 0)$ //90°

TexCoord2: $(0, 2), (0, 0), (2, 2), (2, 0)$



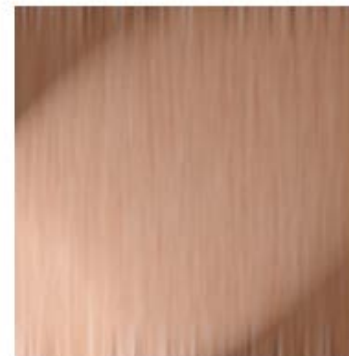
TexCoord0

+



TexCoord1

=



- ❑ **Coordenadas de textura:** ¿Todas las unidades de textura utilizan las mismas coordenadas?
- ❑ Varias texturas pueden utilizar las mismas coordenadas de textura.
- ❑ Cada textura puede utilizar distintas coordenadas de textura:
 - ❑ Incluyendo en la malla varios juegos de coordenadas de textura
 - ❑ Generando coordenadas a partir de los vértices o normales (`env_map`).
 - ❑ Transformando las coordenadas de textura antes de utilizarlas

En **Ogre** ->

En **Ogre** podemos declarar varias unidades de texturay determinar el juego de coordenadas que se quiere utilizar

```
pass { // this is a multitexture pass
texture_unit { // Texture unit 0
    // tex_coord_set 0 // valor por defecto: Primer juego de coordenadas de la malla
    texture wibbly.jpg
}
texture_unit { // Texture unit 1
    // tex_coord_set 1 // si la malla tiene un segundo juego de coordenadas de
    // textura, podríamos usarlo de esta forma.
    rotate 1.57 // también podemos aplicar una transformación a las coordenadas:
    // usamos el primer juego pero girado 90°
    texture wobbly.png
}
texture_unit { // Texture unit 2
    env_map spherical // planar / cubic_reflection / cubic_normal
    texture checker.jpg // Environment maps make an object look reflective
} }
```



```
material exampleMT2 {  
    technique {  
        pass {  
            ambient 0.5 0.5 0.5  
            diffuse 1.0 1.0 1.0  
            texture_unit // Texture unit 0  
            {  
                tex_coord_set 2 // Tercer juego de coordenadas de la malla  
                texture wibbly.jpg  
                // color_op modulate // valor por defecto  
            }  
            texture_unit // Texture unit 1  
            {  
                tex_coord_set 1 // Segundo juego de coordenadas de la malla  
                texture wobbly.png  
                colour_op add  
            }  
        }  
    }  
}
```

En este ejemplo, cada unidad de textura utiliza un juego de coordenadas de la malla.

La malla tendría que tener tres juegos de coordenadas de textura

```
material exampleMT3 {
    technique {
        pass {
            ambient 0.5 0.5 0.5
            diffuse 1.0 1.0 1.0
            texture_unit { // Texture unit 0
                texture wibbly.jpg
                // tex_coord_set 0 // utiliza el juego 0 de coordenadas de tex. de la malla
                scroll 0.5 0.0 // pero antes de utilizarlas trasladamos las coordenadas
                             // 0.5 en horizontal
            }
            texture_unit { // Texture unit 1
                texture wobbly.png
                // tex_coord_set 0 // utiliza el mismo juego de coordenadas de tex.
                rotate 1.57 // pero ahora giramos las coordenadas de tex. 90°
                colour_op add // Los colores de esta imagen se suman con ...
            }
        }
    }
}
```

En este ejemplo, ambas unidades de textura utilizan las mismas coordenadas (text_coord_set 0), pero se transforman antes de utilizarlas

- ❑ **Las coordenadas de textura se pueden transformar** de la misma forma que las coordenadas de los vértices (trasladar, rotar, escalar), pero hay que tener cuidado porque el resultado puede quedar fuera de $[0, 1]$ (-> **tex_address_mode wrap (repeat) | clamp | ...**)

Por ejemplo, para realizar una rotación de 90° se aplica un giro de $\theta=90^\circ$ sobre el eje Z a las coordenadas de textura (u, v):

$$(u, v) \rightarrow \text{Roll}(\theta) (u, v) = (u \cos(\theta) - v \sin(\theta), u \sin(\theta) + v \cos(\theta))$$

-> **tex_address_mode**

- ❑ **Texture animation:** Las coordenadas de textura se transforman en función del tiempo transcurrido desde el último renderizado.

Por ejemplo, para realizar un scroll horizontal se aplica una traslación incrementando (decrementando) la coordenada u

$$(u, v) \rightarrow (u + \text{velocidad} * \text{tiempo transcurrido}, v) \rightarrow (u + \dots, v)$$

-> **tex_address_mode ...**

❑ Texture Units:

`texture_unit [name] { atributos para esta unidad de textura }`

❑ Atributos relativos a cada unidad de textura

https://ogrecave.github.io/ogre/api/latest/_material-_scripts.html

`texture <name> // nombre del archivo de la imagen + ...`

`tex_address_mode wrap`

`// what happens when texture coordinates exceed 1.0:`

`// wrap, clamp, mirror, ...`

`filtering bilinear`

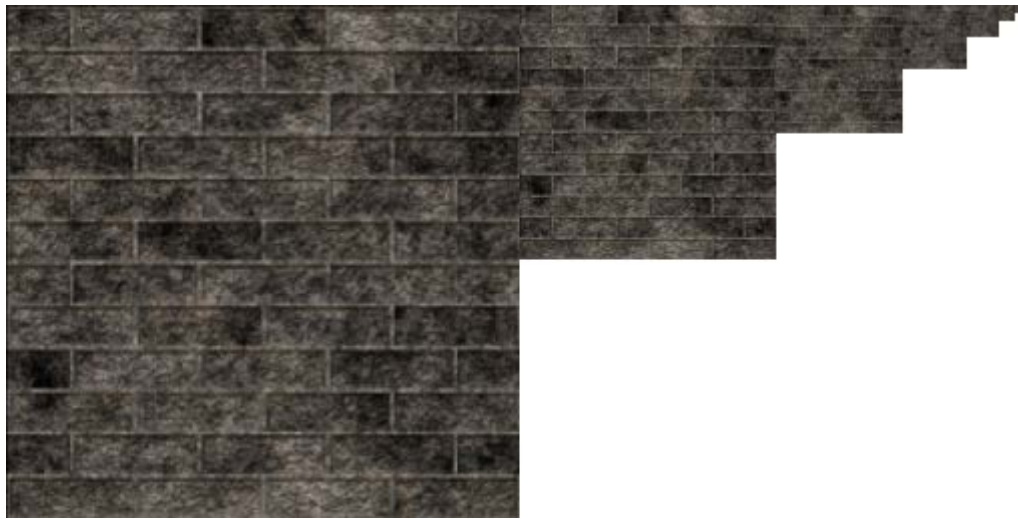
`// the type of texture filtering used when magnifying`

`// or minifying a texture: bilinear,trilinear,anisotropic,none,...`

`... // ->`

sampling state: el formado por **mode+filtering** y utilizado para determinar el color asociado a unas coordenadas de texturas

- ❑ **Mip-Mapping**: Para una imagen se utiliza una serie de imágenes (**mipmaps**), cada una con menor resolución que la anterior (normalmente: $2^N, 2^{N-1}, \dots, 2^1, 1$)



A la hora de determinar el color de un fragmento (x, y, z) , dependiendo de la distancia (z) , se utiliza un **mipmap** u otro.

En OpenGL se pueden generar al carga una imagen con el comando `glGenerateMipmap`. **Ogre** lo hace automáticamente.

❑ Texture Units: Atributos relativos a cada unidad de textura

Only applies to the fixed-function pipeline or the RTSS

tex_coord_set 0 // default 0

colour_op <replace | add | modulate | alpha_blend> // (default modulate)
how the colour of this texture layer is combined with the one below it (or the lighting effect on the geometry if this is the first layer)

scroll <u> <v> // the translation offset of the texture

scroll_anim <uSpeed> <vSpeed> // number of loops per second (+/-)

rotate <angle> // anticlockwise rotation factor

rotate_anim <revs_per_second> // complete anticlockwise revolutions per second

scale <uScale> <vScale>

anim_texture transform <base_name> <num_frames> <duration>

transform <matrix>

wave_xform ...

... // ..._material-_scripts.html

❑ Atributos para cada pase

lighting <on | off> // (default on) No effect if a vertex program is used

shading <mode> // flat, Gouraud, Phong (default gouraud)

polygon_mode <solid | wireframe | points> // (default solid)

ambient 1.0 1.0 1.0 1.0 // default

diffuse 1.0 1.0 1.0 1.0 // default

specular 0.0 0.0 0.0 0.0 0.0 // default

emissive 0.0 0.0 0.0 0.0 // default

depth_check on // depth-buffer checking on or not (default on)

depth_write on // depth-buffer writing on or not (default on)

depth_func less_equal // (default less_equal)

alpha_rejection <function> <value> // (default always_pass)

scene_blend <blend_type> //add, modulate, color_blend, alpha_blend

scene_blend <src_factor> <dest_factor> // control over the blending operation

❑ Atributos para cada pase

```
normalise_normals <on | off> // (default off)
transparent_sorting <on | off | force> // (default on)
cull_hardware <clockwise | anticlockwise | none> // (default OpenGL)
cull_software <back | front | none> // (default back)
colour_write on // default
start_light 0 // default
max_lights 8 // default
point_size 1.0 // default
polygon_mode_overrideable true // da prioridad al definido en la cámara
fog_override <on | off> [<type> <colour> <density> <start> <end>]
```

...

https://ogrecave.github.io/ogre/api/latest/_material-_scripts.html

□ SkyPlane, SkyDome, SkyBox

SkyPlane: Plano curvado a una distancia fija de la cámara.

- . No se mueve con la cámara, pero se orienta hacia la cámara-> ...
- . ¿Lo posicionamos muy lejos o muy cerca? -> ... cerca
- . ¿Lo renderizamos lo primero o lo último? -> ... lo primero
- . ¿Tapará al resto de la escena? -> configurar depth buffer

Usando **Ogre**

```
mSM -> setSkyPlane(true, Plane(Vector3::UNIT_Z, -20),
    "IG2App/space", 1, 1, true, 1.0, 100, 100);
```

```
// enable, plane, materialName, scale = 1000, tiling = 10, drawFirst,
// bow = 0, Xsegments = 1, Ysegments = 1, ...
```

↑
sin curvatura

Para ciertos efectos, como por ejemplo sombras y reflejos, es necesario capturar el resultado de un renderizado para utilizarlo en el resultado final (en la ventana). En estos casos primero se renderiza directamente en texturas para después utilizar las imágenes así obtenidas (estas texturas no contienen la imagen de un archivo).

En **Ogre** tenemos la clase

□ **RenderTarget**: Se puede renderizar directamente en la ventana, o en texturas (OpenGL FrameBuffer Objects). Subclases:

RenderWindow, **RenderTexture**, **MultiRenderTarget**

Cada **RenderTarget** se puede dividir en varios **Viewports**.

Cada **Viewport**, además de sus dimensiones, tiene asociada una referencia a una cámara y un orden de renderizado (Z-order para posibles superposiciones)

Viewport* vp = **getRenderWindow()->addViewport(puntero a cámara);**

Scene Graph: SceneNode

Ref. a **MovableObject**:

cámara, luz, entidad
(ref. a malla y material)

Modelado: traslación,
giro y escala

Puntero al padre y
a los hijos

Cada cámara se crea con el
gestor de escena y guarda
una referencia ese gestor.

Render Targets:

RenderWindow con
puertos de vista

RenderTexture con
puertos de vista

Cada **Viewport** se añade a
un **RenderTarget**
indicando la cámara con la
que se renderizará

Recursos:

Mallas

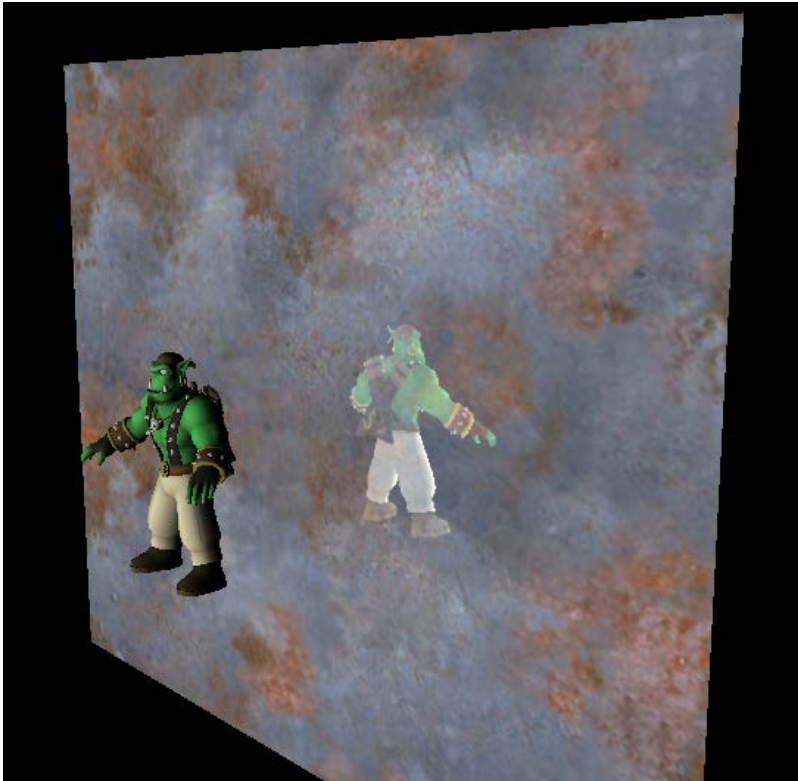
Materiales

❑ Tenemos **dos RenderTargets**:

RenderWindow (mWindow)

cam (vp)

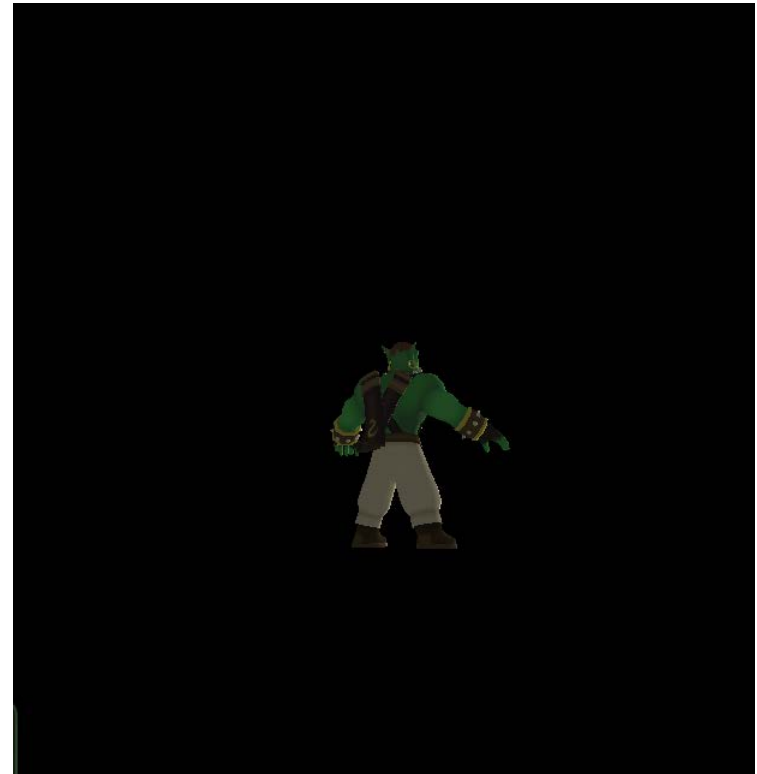
Frame Buffer



RenderTexture (renderTexture)

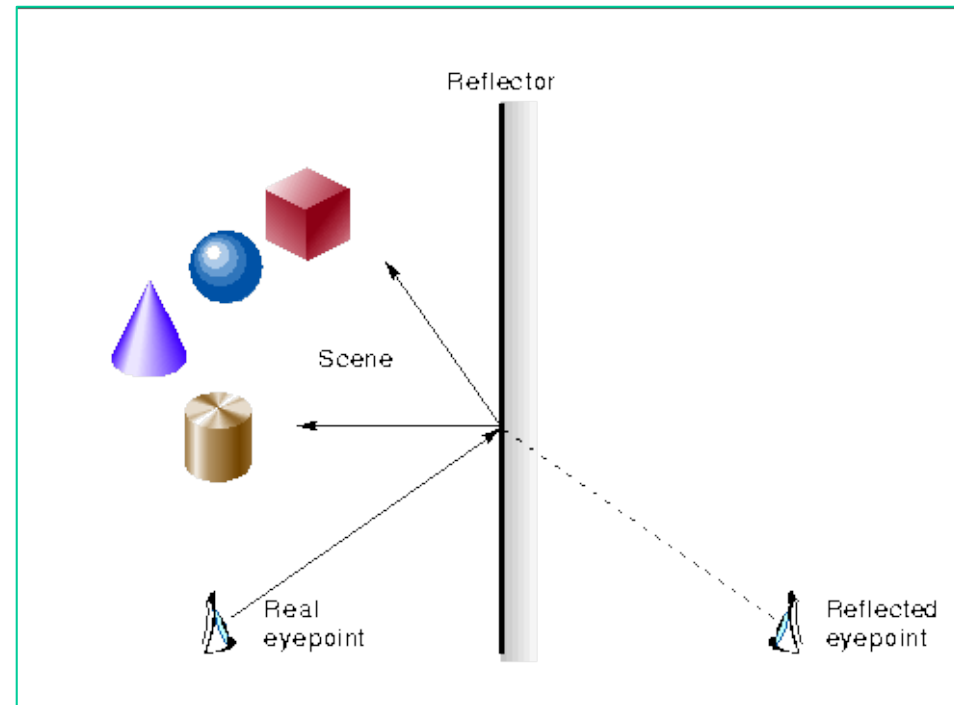
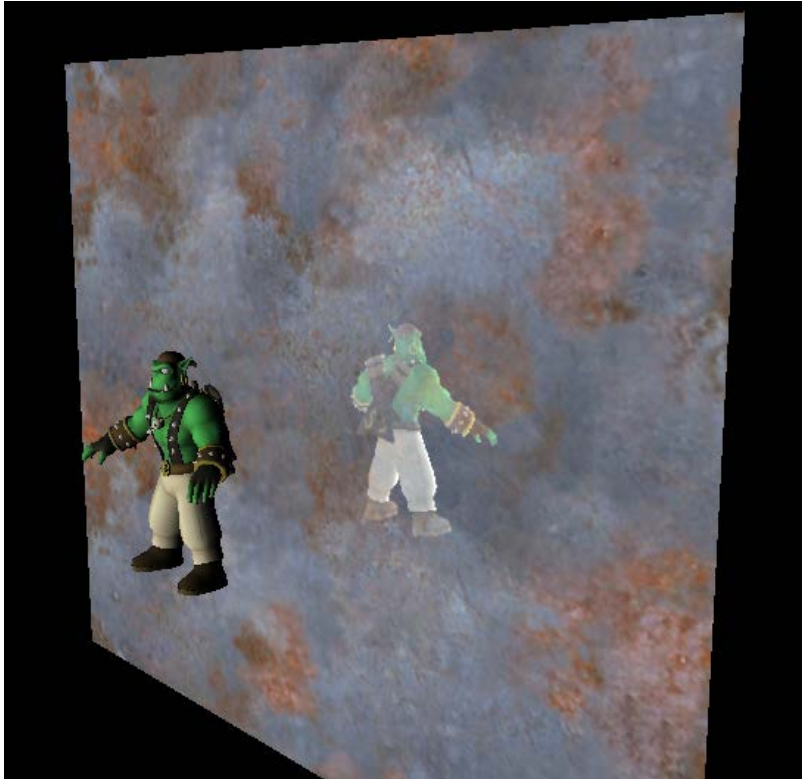
camRef (vpt)

Frame Buffer Object



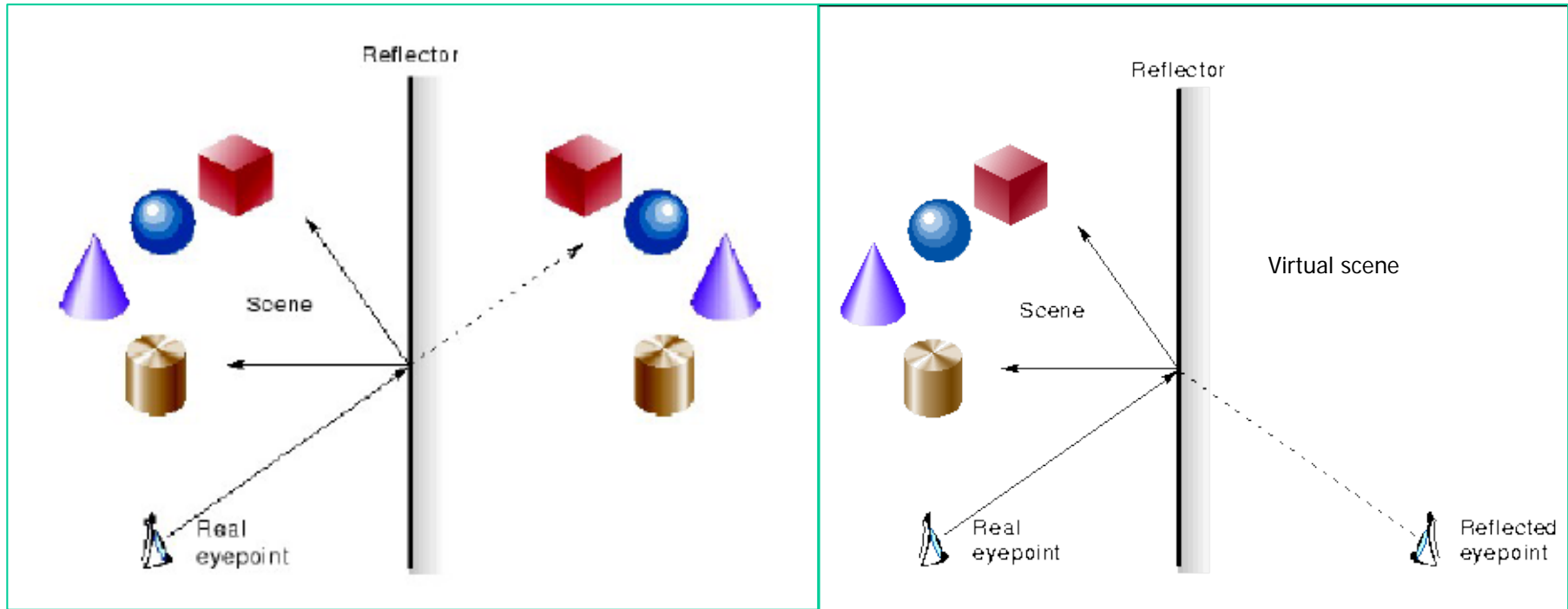
Ejemplo: Reflejo

- ❑ Vamos a añadir un RenderTarget de la clase **RenderTexture** y una cámara para obtener el reflejo de la escena en un plano.



- ❑ Para posicionar la cámara del reflejo: se traza una recta paralela al vector normal que pasa por la posición de la cámara y se calcula la intersección con el plano.

Ejemplo: Reflejo



1- Entidad Espejo Plano

Creamos la malla de un plano con nombre (ojo, único) y con la orientación que habrá de tener el reflejo-espejo

```
MeshManager::getSingleton().createPlane(  
    meshName, ...,  
    Plane(normal, distance), ...);
```

Creamos la entidad a partir de la malla:

```
Entity* entEP = mSM-> createEntity(meshName);
```

Adjuntamos la entidad a un nodo **espejoNode** que se crea de la escena (**attachObject(entEP)**) y le ponemos un material (el material por defecto es blanco)

```
entEP -> setMaterialName(materialName);
```

Este material se modifica con el reflejo-espejo. Parte de estas operaciones se pueden hacer en la misma clase **Plano**

2- Añadimos una nueva cámara para el reflejo

```
Camera* camRef = mSM->createCamera("RefCam"); //ojo nombre único
```

Configuramos su frustum igual que el de la cámara que usamos para la escena (con `setNearClipDistance()`, `setFarClipDistance()`) y la adjuntamos al nodo de la cámara de la escena (que se obtiene a partir de la cámara con `getParentSceneNode()`)

Configuramos el plano sobre el que se quiere el reflejo-espejo con la misma orientación que la malla de la entidad

```
MovablePlane* mpRef = new MovablePlane(Vector3::ZERO, distancia);
```

Adjuntamos este plano móvil al nodo del reflejo-espejo

```
espejoNode->attachObject(mpRef);
```

Configuramos la cámara para el reflejo-espejo sobre el plano

```
camRef->enableReflection(mpRef);
```

```
camRef->enableCustomNearClipPlane(mpRef);
```


Ejemplo: Reflejo-Espejo

- 3- Añadimos una textura, en el mismo grupo de recursos que la malla del reflejo-espejo, para usarla de **RenderTarget** y de textura del reflejo-espejo

```
TexturePtr rttRef= TextureManager::getSingleton().createManual(
    "rttReflejo", // ojo, nombre único -> (*)
    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
    TEX_TYPE_2D,
    (Real)mWindow->getViewport(0)->getActualWidth(), // ejemplo
    (Real)cam->getViewport()->getActualHeight(), // ejemplo
    0, PF_R8G8B8, TU_RENDERTARGET);
```

Añadimos un puerto de vista al **RenderTarget** con la nueva cámara

```
RenderTexture* renderTexture= rttRef->getBuffer()->getRenderTarget();
Viewport * vpt = renderTexture-> addViewport(camRef);
vpt->setClearEveryFrame(true);
vpt->setBackgroundColour(ColourValue::Black);
```

4- Añadimos la nueva unidad de textura al material del reflejo-espejo

```
TextureUnitState* tu = entEP -> getSubEntity(0) -> getMaterial()->  
    getTechnique(0) -> getPass(0)->  
    createTextureUnitState("rttReflejo"); // <- (*)  
  
tu-> setColourOperation(LBO_MODULATE); // black background  
    // LBO_ADD / LBO_ALPHA_BLEND / LBO_REPLACE  
tu->setTextureAddressingMode(TextureUnitState::TAM_CLAMP);  
...
```

Queremos que la imagen se proyecte sobre el plano del espejo-reflejo conforme a la cámara (frustum): hay que ajustar las coordenadas de textura con el plano cercano

```
tu-> setProjectiveTexturing(true, camRef);  
// la clase Camera hereda de Frustum
```

5- Si queremos realizar algún cambio a la escena antes de renderizar el reflejo, entonces necesitamos ser observadores del nuevo **RenderTarget** (la nueva textura), para que nos avise antes y después del renderizado.

Para eso, tenemos que implementar la clase **RenderTargetListener**, con respuestas a los eventos:

```
virtual void preRenderTargetUpdate(const Ogre::RenderTargetEvent& evt);
```

```
// modificar luz ambiente, ...
```

```
virtual void postRenderTargetUpdate(const Ogre::RenderTargetEvent& evt);
```

```
// restablecer los cambios
```

Y añadir al objeto de observador del RenderTarget:

```
renderTexture->addListener(...);
```

- ❑ Clases que deben ser incluidas para hacer el reflejo
- ❑ La clase **Plano** debe heredar de **Viewport::Listener** y de **RendertargetListener** y necesita incluir
 - ❑ **OgreMovablePlane.h**
 - ❑ **OgreRenderTargetListener.h**
- ❑ Para definir el reflejo necesita incluir además
 - ❑ **OgreRenderTexture.h**
 - ❑ **OgreTextureManager.h**
 - ❑ **OgreHardwarePixelBuffer.h**
 - ❑ **OgreSubEntity.h**
 - ❑ **OgreTechnique.h**

- ❑ La aplicación lanza el bucle de renderizado automático con **root->startRendering()**; dentro del cual se llama a **renderOneFrame()** :
- ❑ Llama a **RenderSystem::_updateAllRenderTargets()**
 - ❑ Para todos los **RenderTarget** activos llama a **update()**
 - ❑ Para todos los **Viewport** activos llama a **update()**
 - ❑ Para su **Cámara**, llama a **_renderScene()**
 - ❑ Para su **SceneManager**, llama a **_renderScene()**
- ❑ Avisa a los **FrameListener** suscritos, antes y después de **renderOneFrame()**
- ❑ Intercambia el buffer trasero y delantero

Rendering

