Informática Gráfica II

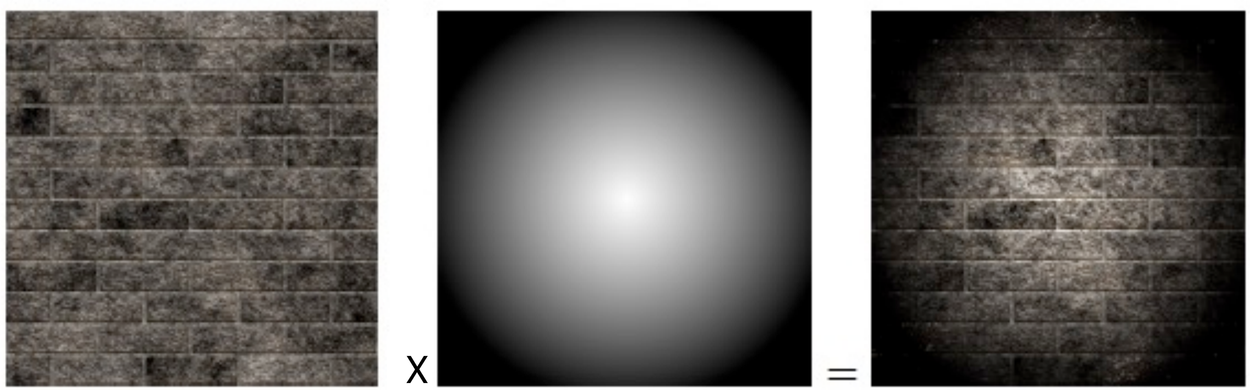
Object-oriented Graphics Rendering Engine

Multitexturing y RenderTextures

Material original: Ana Gil Luezas
Adaptación al curso 24/25: Alberto Núñez
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

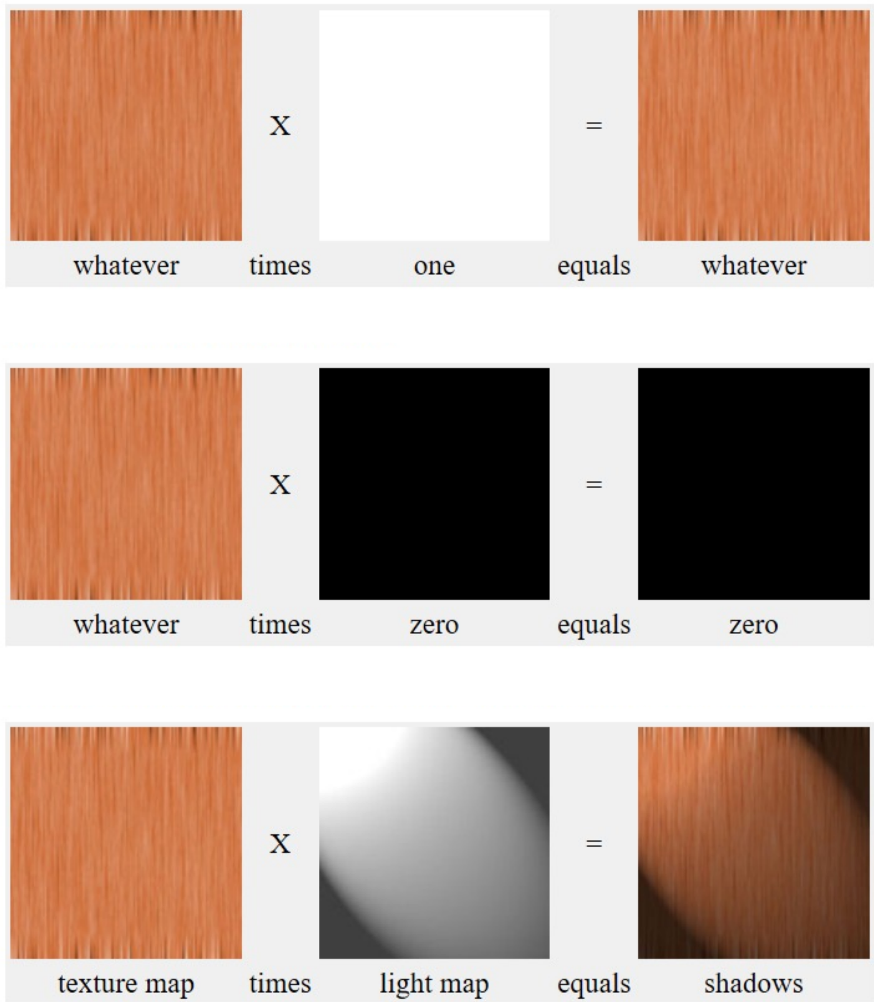
Multitexturing

- **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)

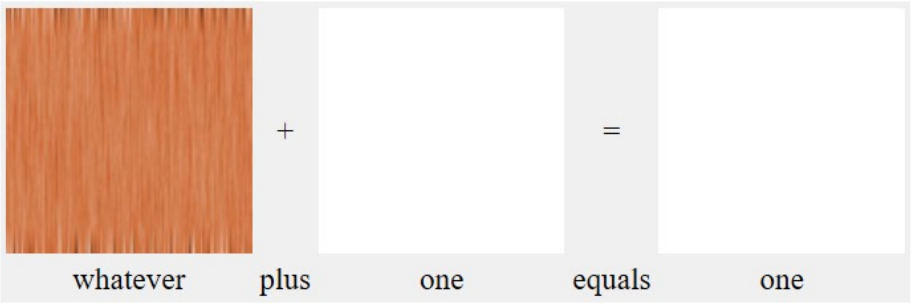


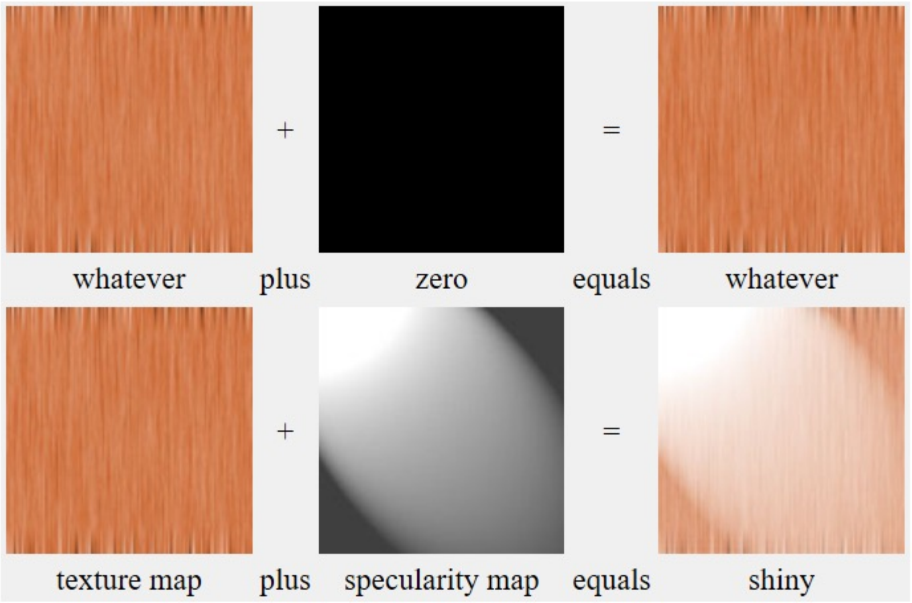
Multitexturing

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

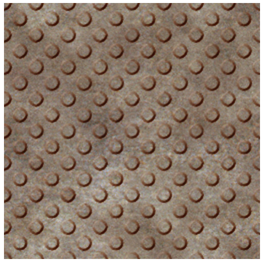


- **Add:** La suma tiende a blanco (aumenta el brillo)

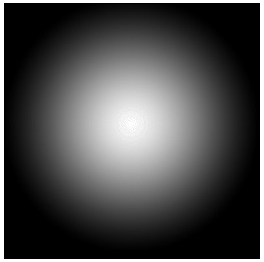




Ejemplo



BumpyMetal.jpg



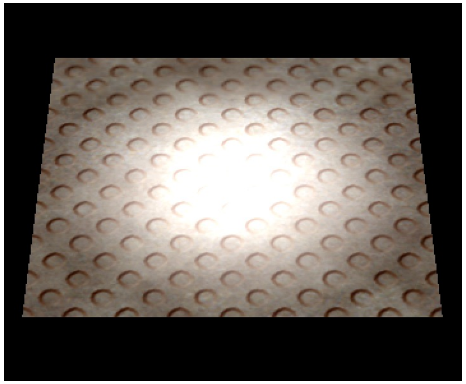
lightMap.jpg

```
material example/textureTest{
    technique {
        pass {

            texture_unit {
                texture BumpyMetal.jpg
                colour_op modulate
            }
             texture_unit {
                texture lightMap.jpg
                colour_op add
            }
        }
    }
}
```

```
material example/textureTest{
    technique {
        pass {

            texture_unit {
                texture BumpyMetal.jpg
                colour_op modulate
            }
             texture_unit {
                texture lightMap.jpg
                colour_op modulate
            }
        }
    }
}
```



Mezcla de colores resultante: Luz x TU_0 + TU_1

```
material exampleMT1{
  technique{
     pass  {
         ambient 0.5 0.5 0.5    // Coeficientes de reflexión para luz ambiente
         diffuse 1.0 1.0 1.0    // Idem para la componente difusa

          // Texture unit 0
          texture_unit {
              texture wibbly.jpg  // Nombre del archivo de la imagen
              color_op modulate   // Los colores se multiplican con …
          }

          // Texture unit 1 (multitexture pass)
          texture_unit {
              texture wobbly.png   // Nombre del archivo de la imagen
              colour_op add        // Los colores de esta imagen se suman con …
          }
     }
  }
}
```

Combinación de dos texturas con una máscara

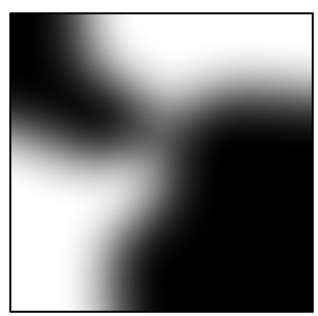
```
material Template/texture_blend{
    technique{
       pass{
          texture_unit{
            texture Material_grass.png
          }
          texture_unit{
             texture Material_alpha_blend.png
             colour_op alpha_blend
          }
          texture_unit{
             texture Material_dirt.jpg
             colour_op_ex blend_current_alpha src_texture src_current
          }
       }
    }
 }
```

Mezcla la hierba con el componente alfa

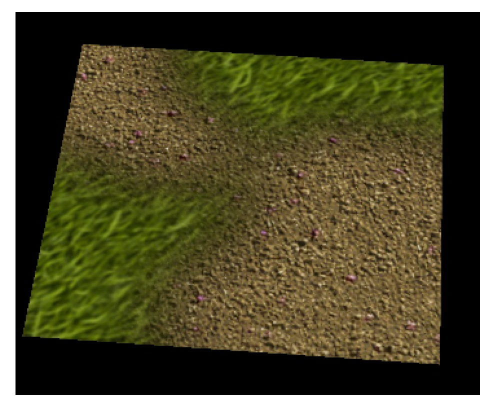
Mezcla la tierra con la máscara de la fase anterior

Material_grass.png

Material_dirt.jpg



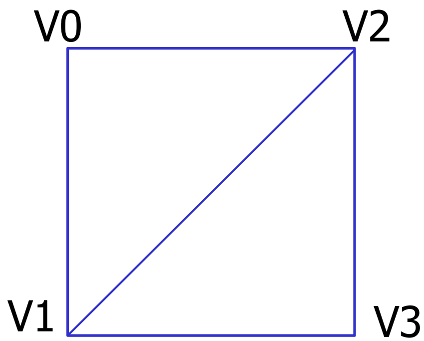
Material_alpha_blend.png



Fuente: https://wiki.ogre3d.org/Materials

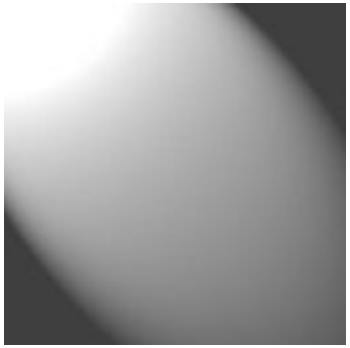
- En una malla se pueden especificar diferentes asignaciones de coordenadas de textura para utilizarlas en distintas unidades de textura:

```
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)  //Rotar 90º
TexCoord2: (0,  2),     (0, 0),      (2,  2),    (2,  0)
```



TexCoord0

+



TexCoord1

=

Coordenadas de textura

- **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

Coordenadas de textura

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

```
// Multitexture pass
pass{

   // Texture unit 0
   texture_unit{
      tex_coord_set 0  // Por defecto: Primer juego de coordenadas de la malla
      texture wibbly.jpg
   }

   // Texture unit 1
   texture_unit{
      tex_coord_set 1  // Segundo juego de coordenadas (si disponible) ó
      rotate 1.57      // también podemos aplicar una transformación a las coordenadas
      texture wobbly.png
    }

    // Texture unit 2
    texture_unit{
      env_map spherical    // Planar / cubic_reflection / cubic_normal
      texture checker.jpg
    }
}
```

Coordenadas de textura

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 exampleMT2{
   technique{
      pass{
         ambient 0.5 0.5 0.5
         diffuse 1.0 1.0 1.0

         // Texture unit 0
         texture_unit{
            tex_coord_set 2          // Tercer juego de coordenadas de la malla
            texture wibbly.jpg
            // colour_op modulate    // Valor por defecto de colour_up -> modulate
         }

         // Texture unit 1
         texture_unit{
            tex_coord_set 1          // Segundo juego de coordenadas de la malla
            texture wobbly.png
            colour_op add
         }
      }
   }
}
```

Coordenadas de textura

```
material exampleMT3{
   technique{
      pass{
          ambient 0.5 0.5 0.5
          diffuse 1.0 1.0 1.0

          // Texture unit 0
          texture_unit{
             texture wibbly.jpg
             // tex_coord_set 0  // Utiliza (por defecto) el juego 0 de coordenadas
             scroll 0.5 0.0      // Trasladamos las coordenadas 0.5 en horizontal
          }

          // Texture unit 1
          texture_unit{
             texture wobbly.png
             // tex_coord_set 0  // Utiliza el mismo juego de coordenadas
             rotate 1.57         // Ahora giramos las coordenadas 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

Transformación de las coordenadas de textura

- **Las coordenadas de textura se pueden transformar** de la misma forma que las coordenadas de los vértices (trasladar, rotar, escalar)
  - 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 90º se aplica un giro de θ=90º sobre el eje Z a las coordenadas de textura (u, v):

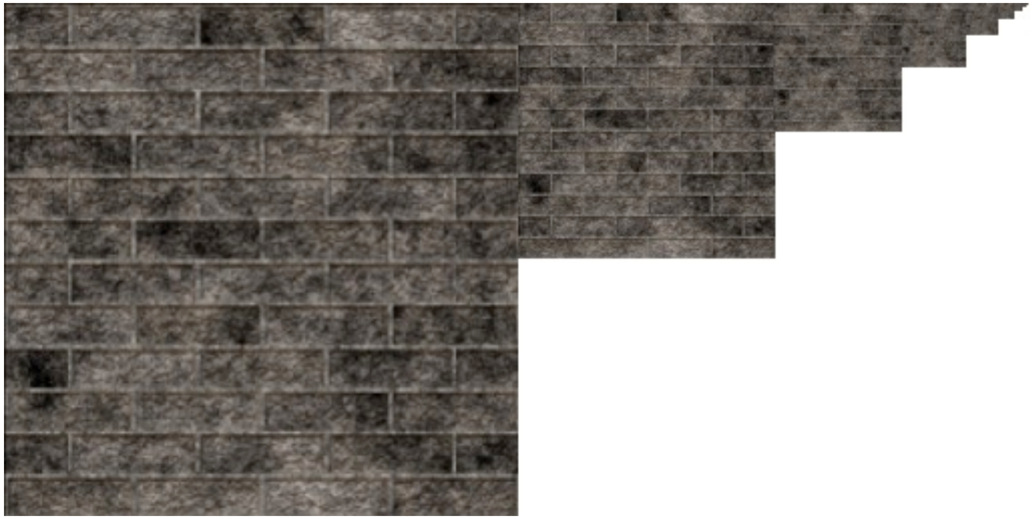
  $(u, v) \rightarrow Roll(\theta)\,(u, v) = (u \cos(\theta) - v \sin(\theta),\ u \sin(\theta) + v \cos(\theta))$ -> tex_address_mode

- **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) -> (u + velocidad * tiempo transcurrido, v) -> (u + …, v) -> tex_address_mode

Mip-Mapping

- **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}, …. 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.

Unidades de textura (Texture Units)

❑ Texture Units

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

❑ Atributos relativos a cada unidad de textura

```
// Nombre del archivo de la imagen
texture <name>

// Si las coordenadas de textura superan 1.0: wrap, clamp, mirror, …

tex_address_mode wrap


// Filtro aplicado a la textura: bilinear, trilinear, anisotropic, none,…

filtering bilinear
```

❑ Sampling state

- ❑ Formado por **`mode+filtering`** y utilizado para determinar el color asociado a unas coordenadas de texturas

Atributos de Texture Unit

tex_coord_set <index> // Default: 0

- Establece qué conjunto de coordenadas de textura se utilizará para esta capa

colour_op <replace | add | modulate | alpha_blend> // Default: modulate

- Determina cómo se combina el color de esta capa de textura con la inferior

scroll <u> <v>

- Establece el desplazamiento de la textura, es decir, desplaza la textura en las direcciones u y v

scroll_anim <uSpeed (loopsPerSec)> <vSpeed (loopsPerSec)>

- Establece un desplazamiento animado para la capa de textura

rotate <angleInDegrees>

- Establece el factor de rotación en antihorario aplicado a las coordenadas de la textura

rotate_anim <revs_per_second>

- Establece una rotación de textura animada para esta capa

Atributos de Texture Unit

scale <uScale> <vScale>

- Establece el factor de escala aplicado a las coordenadas de la textura

anim_texture <base_name> <num_frames> <duration>

- Establece las imágenes que se utilizarán en una capa de textura animada.
- Todas las imágenes deben tener el mismo tamaño, y sus nombres deben tener un número de fotograma añadido antes de la extensión
  - Por ejemplo, si especifica un nombre "flame.jpg" con 3 fotogramas, los nombres de las imágenes deben ser "flame_0.jpg", "flame_1.jpg" y "flame_2.jpg".

wave_xform <ttype> <waveType> <base> <frequency> <phase> <amplitude>

- Establece un efecto general de modificación de la textura en función del tiempo
  - ttype = [scroll_x | scroll_y | rotate | scale_x | scale_y]
  - waveType = [sine | triangle | square | sawtooth | inverse_sawtooth | pwm]

Podéis consultar la lista completa aquí:
https://ogrecave.github.io/ogre/api/latest/_material-_scripts.html#Texture-Units

Atributos de los pases

lighting <on | off> // Default: on. No tiene efecto si se utiliza un shader de vértices

- Establece si se activa o no la iluminación dinámica para este pase

shading <mode> // Default: default Gouraud. mode = [flat |Gouraud | Phong]

- Establece el tipo de sombreado aplicado para representar la iluminación dinámica de este pase

polygon_mode <solid | wireframe | points> // Default: solid

- Indica cómo deben rasterizarse los polígonos, es decir, si deben rellenarse o sólo dibujarse como líneas o puntos.

ambient (<red> <green> <blue> [<alpha>]| vertexcolour) // Default: 1.0 1.0 1.0 1.0

- Establece las propiedades de reflectancia del color ambiente de este pase

diffuse (<red> <green> <blue> [<alpha>]| vertexcolour) // Default: 1.0 1.0 1.0 1.0

- Establece las propiedades de reflectancia del color difuso de este pase

specular (<red> <green> <blue> [<alpha>]| vertexcolour) <shininess> // Default: 0.0 0.0 0.0 0.0 0.0

- Establece las propiedades de reflectancia del color especular de este pase

Atributos de los pases

emissive (<red> <green> <blue> [<alpha>]| vertexcolour) // Default 0.0 0.0 0.0 0.0

- Establece la cantidad de auto-iluminación que tiene un objeto

depth_check <on | off> // Default: on

- Establece si este pase renderiza con la *comprobación* del búfer de profundidad activada o no.
- Si ON, cada vez que un píxel está a punto de escribirse en el buffer, se comprueba si está delante de todos los demás píxeles escritos en ese punto. Si no es así, el píxel no se escribe.
- Si OFF, los píxeles se escriben sin importar lo que se haya renderizado antes

depth_write <on | off> // Default: on

- Establece si este pase renderiza o no con la *escritura* en el buffer de profundidad activada.
- Si OFF los píxeles se escriben sin actualizar el buffer de profundidad. Se utiliza cuando se renderiza una colección de objetos transparentes al final de una escena para que se solapen entre sí correctamente.

depth_func <function> // Default: less_equal

- Establece la función utilizada para comparar los valores de profundidad si `depth_check = ON`
- function **=** [always_fail | always_pass | less | less_equal | equal | not_equal | greater_equal | greater]

Atributos de los pases

alpha_rejection <function> <value> // Default: always_pass

- Establece la forma en la que el pase aplicará el componente alfa para rechazar totalmente los píxeles del pipeline
- function = [always_fail | always_pass | less | less_equal | equal | not_equal | greater_equal | greater]

scene_blend <blend_type>

- Establece el tipo de mezcla que este pase tiene con el contenido existente de la escena.
- Mientras que las operaciones de mezcla de texturas vistas en las entradas **texture_unit** tienen que ver con la mezcla entre capas de texturas, esta mezcla trata de combinar la salida de este pase como un todo con los contenidos existentes del objetivo de renderizado. Esta mezcla permite la transparencia de los objetos y otros efectos especiales.
- blend_type = add | modulate | colour_blend | alpha_blend

scene_blend <src_factor> <dest_factor> // Default: one zero (material opaco)

- Esta versión permite un control completo sobre la operación de mezcla, especificando los factores de mezcla de origen y destino.
- final=(passOutput * sourceFactor) + (framebuffer * destFactor)

  factor = [one | zero | dest_colour | source_colour | one_minus_dest_colour | one_minus_dest_colour | dest_alpha | source_alpha | one_minus_dest_alpha | one_minus_source_alpha]

Atributos de los pases

transparent_sorting <on |off | force> // Default: on

- Establece si las texturas transparentes deben ordenarse por profundidad o no

cull_hardware <clockwise | anticlockwise | none> // Default: clockwise

- Establece el modo de selección por hardware para este pase.

desactivar culling para la esfera de la práctica para poder ver su interior

cull_software <back | front | none> // Default: back

- Establece el modo de selección por software para este pase

max_lights <max_lights> // Default: 8

- Establece el número máximo de luces que se considerarán para su uso con este pase

Podéis consultar la lista completa aquí:
https://ogrecave.github.io/ogre/api/latest/_material-_scripts.html#Passes

Skyplane

- Podemos simular el cielo con un plano con **`setSkyPlane`**
  - Realmente es un 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?
    - Podemos configurar el parámetro `depth buffer`

- Es posible crear "manualmente" un plano para simular el cielo
  - Este método crea un plano del que la cámara nunca puede acercarse o alejarse
    - Se mueve con la cámara

Skyplane

- En OGRE tenemos el siguiente método (de la clase SceneManager)

```
virtual void Ogre::SceneManager::_setSkyPlane (bool          enable,
                                               const Plane&   plane,
                                               const String&  materialName,
                                               Real           scale = 1000,
                                               Real           tiling = 10,
                                               bool           drawFirst = true,
                                               Real           bow = 0,
                                               int            xsegments = 1,
                                               int            ysegments = 1,
                                               const String & groupName =
                                               ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME
)
```

- **enable:** True para activar el plano, false para desactivarlo.
- **plane:** Plano que representa su normalidad y su distancia a la cámara.
- **materialName:** Material utilizado en el plano.
- **scale:** La escala aplicada al plano del cielo.
- **tiling:** El número de veces que se coloca la textura en el cielo.
- **drawFirst:** Si es *true*, el plano se dibuja antes que el resto de geometría de la escena, sin actualizar el buffer de profundidad. Todos los demás objetos aparecerán siempre delante del cielo.
- **bow:** Si es cero, el plano será completamente plano. Si está por encima de cero, el plano será curvo, permitiendo que el cielo aparezca por debajo del nivel de la cámara.
- **xsegments**, **ysegments:** Determinan el número de segmentos que tendrá el plano. Esto es más importante cuando se está arqueando el plano, pero también puede ser útil si necesita teselación en el plano para realizar efectos por vértice.
- **groupName:** El nombre del grupo de recursos al que asignar la malla.

Ejemplo de Skyplane

```
Ogre::Plane plane;
plane.d = 1000;
plane.normal = Ogre::Vector3::NEGATIVE_UNIT_Y;
mSM->setSkyPlane(true, plane, "example/sky", 1500, 50, true, 1.5, 50, 50);
```

spaceSky.jpg

Multipass Rendering

no entra en el examen pero mola

- 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:

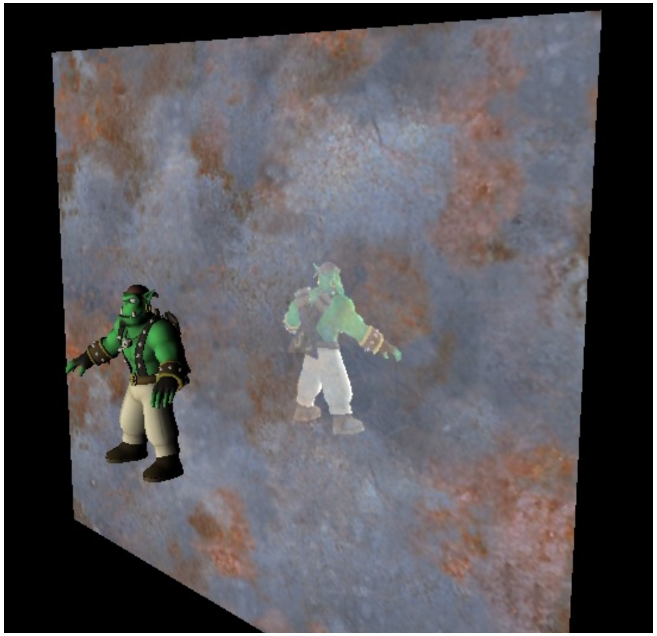
Mallas: . . .

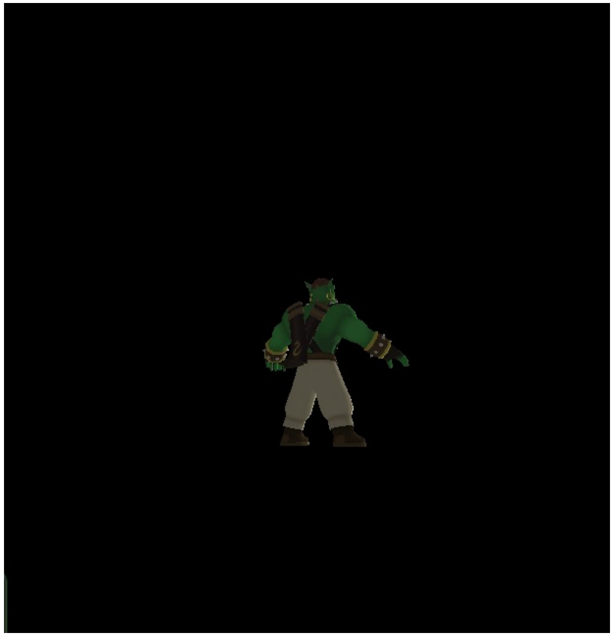
Materiales: . . .

Ejemplo: Reflejo

- Tenemos **dos RenderTargets**

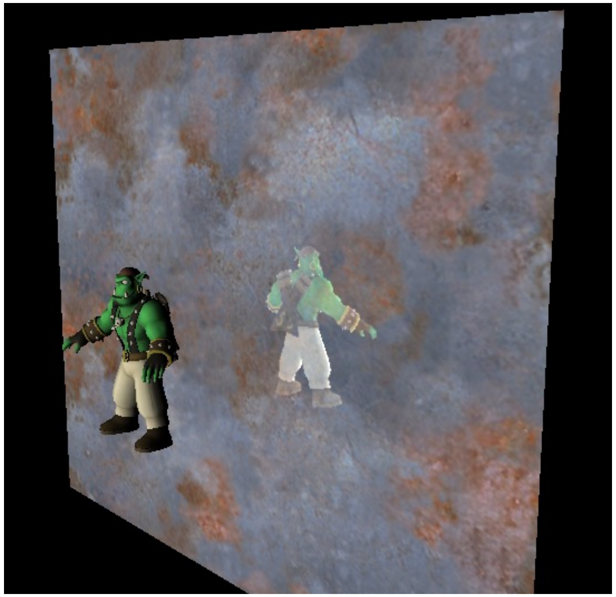
RenderWindow (mWindow) cam (vp) Frame Buffer	**RenderTexture** (renderTexture) camRef (vpt) Frame Buffer Object

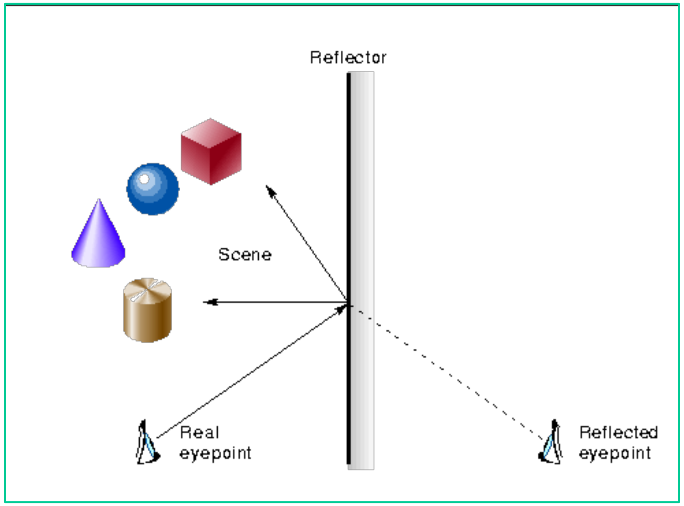




Ejemplo: Reflejo

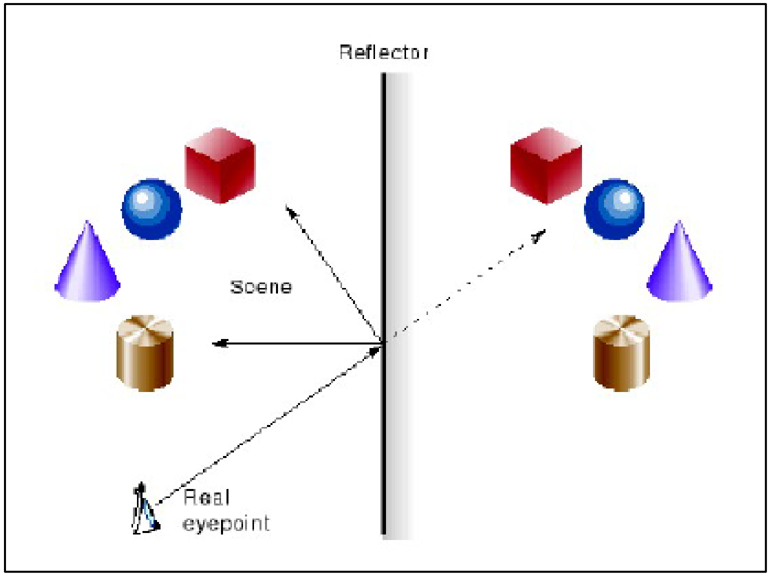
- Vamos a añadir un **RenderTarget** de la clase **RenderTexture** y una cámara para obtener el reflejo de la escena en un plano.

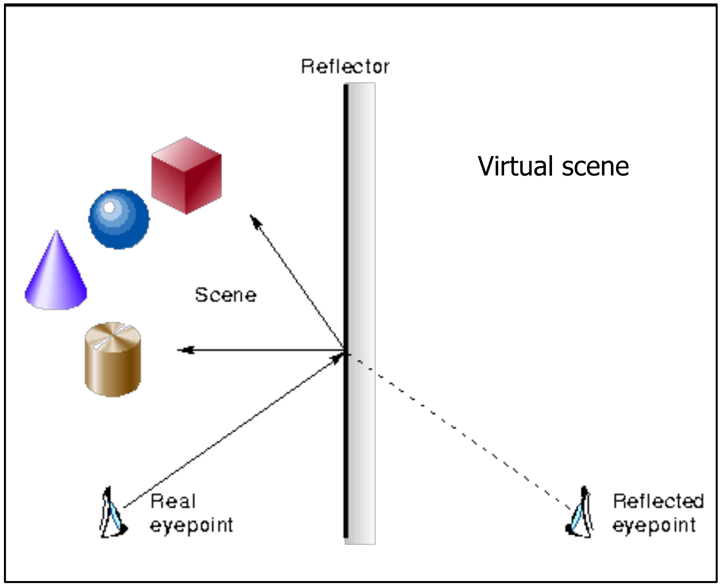




- Para posicionar la cámara del reflejo
  - Se traza una recta paralela al vector normal que pasa por la posición de la cámara y se calcula la intersección con el plano.

Ejemplo: Reflejo





Ejemplo: Reflejo-Espejo

1- Entidad Espejo Plano

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

```
MeshManager::getSingleton().createPlane("mirror",ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
                                        Plane(Vector3::UNIT_Z, 0),
                                        500, 500, 200, 200, true, 1, 5, 5,
                                        Vector3::UNIT_Y);
```

- Creamos la entidad a partir de la malla:

```
Entity* entMirror = mSM->createEntity ("exampleMirror", "mirror");
```

- Adjuntamos la entidad a un nodo `mirrorNode` que se crea de la escena y le ponemos un material

```
entMirror->setMaterialName("example/mirror");
SceneNode* mirrorNode = mSM->getRootSceneNode()->createChildSceneNode();
mirrorNode->attachObject(entMirror);
```

- Este material se modifica con el reflejo-espejo

Ejemplo: Reflejo-Espejo

2- Añadimos una nueva cámara para el reflejo

```
Camera* camRef = mSM->createCamera("RefCam");
```

- Configuramos su frustum igual que el de la cámara que usamos para la escena
  - con `setNearClipDistance()` y `setFarClipDistance()`
  - la adjuntamos al nodo **de la cámara de la escena**

- 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::UNIT_Z, 0);
```

- Adjuntamos este plano móvil al nodo del reflejo-espejo

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

- Configuramos la cámara para el reflejo-espejo sobre el plano

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

Ejemplo: Reflejo-Espejo

3- Añadimos la textura

- En el mismo grupo de recursos que la malla del reflejo-espejo
- Así, podremos usarla de **RenderTarget** y de textura del reflejo-espejo

```
TexturePtr rttRef= TextureManager::getSingleton().createManual("rttReflejo",
                                    ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
                                    TEX_TYPE_2D,
                                    (Real)getRenderWindow()->getViewport(0)->getActualWidth(),
                                    (Real)cam->getViewport()->getActualHeight(),
                                     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);
```

Ejemplo: Reflejo-Espejo

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

```
TextureUnitState* tu = entMirror->getSubEntity(0)->getMaterial()->
                     getTechnique(0)->getPass(0)->createTextureUnitState("rttReflejo");
tu->setColourOperation(LBO_MODULATE);
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);
```

Ejemplo: Reflejo-Espejo

5- Cambios en la escena antes de renderizar el reflejo

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

- Y añadir al objeto de observador del **RenderTarget**:

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

Ejemplo: Reflejo

- Clases que deben ser incluidas para hacer el reflejo
- La clase que implemente el **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`

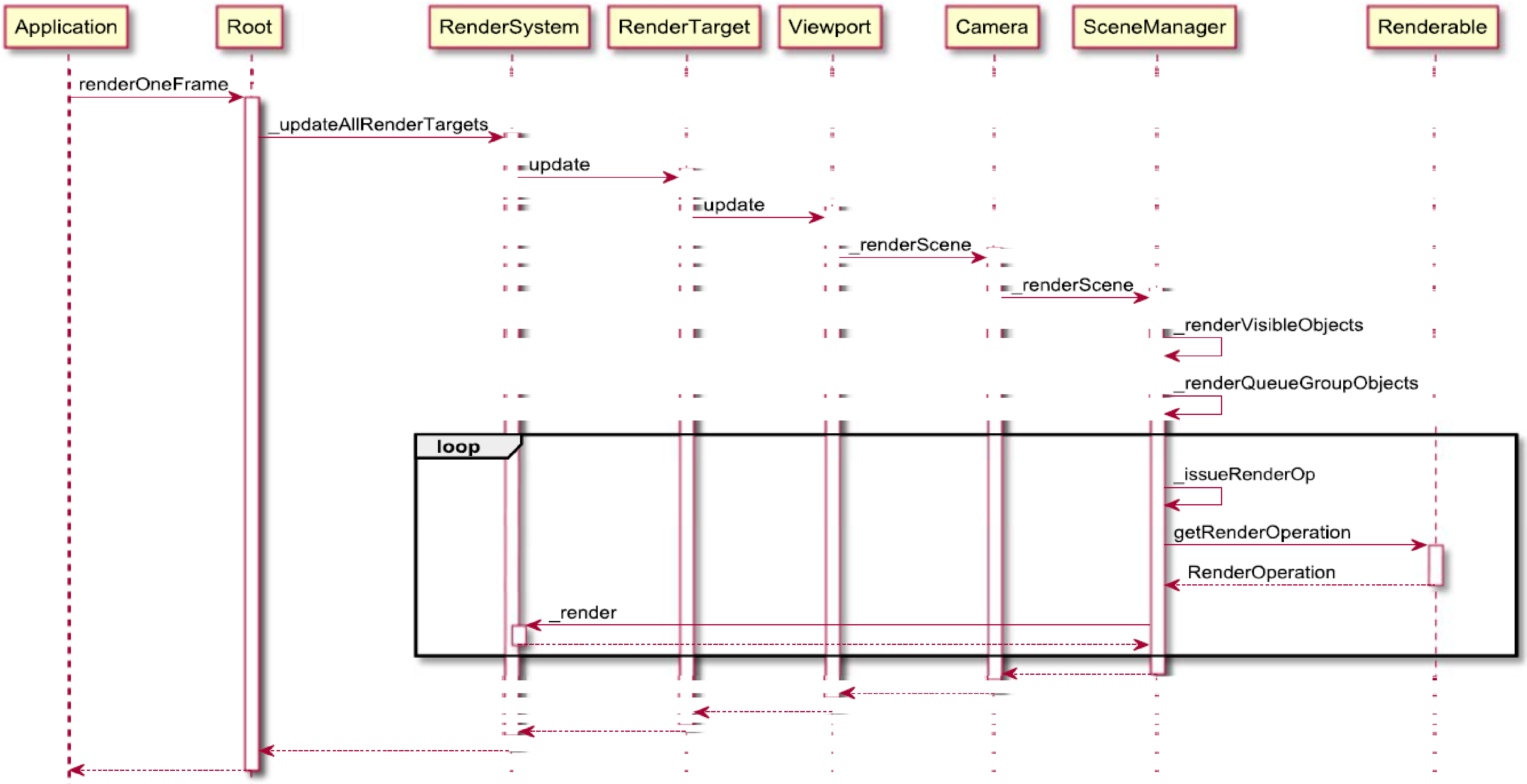
Rendering

- 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


Application
Root
RenderSystem
RenderTarget
Viewport
Camera
SceneManager
Renderable
renderOneFrame
_updateAllRenderTargets
update
update
_renderScene
_renderScene
_renderVisibleObjects
_renderQueueGroupObjects
loop
_issueRenderOp
getRenderOperation
RenderOperation
_render