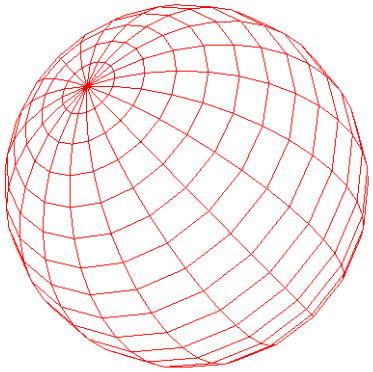
Universidad de Huelva

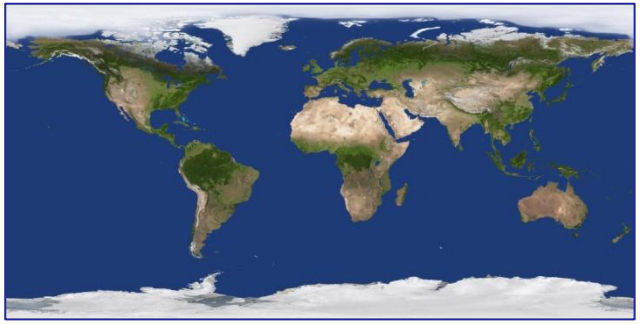
Tema 7

Texturas

Índice

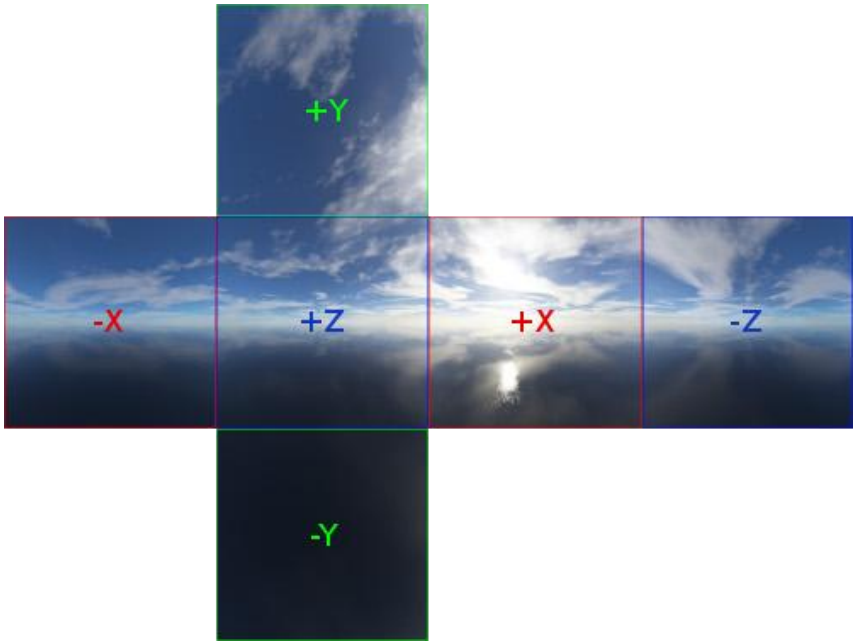
- Se denomina ***textura*** a la aplicación de una imagen a una primitiva geométrica.
- Los puntos de las texturas se denominan ***texels***.

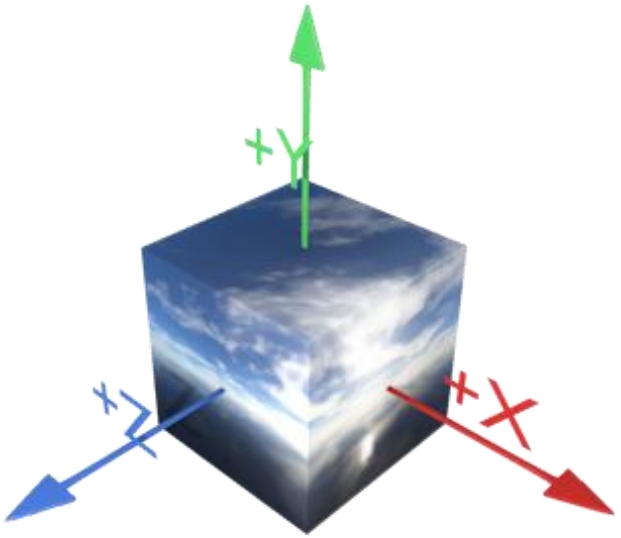




- Tipos de texturas: 1D, 2D, 3D, CUBEMAP
  - Texturas 1D: vector de texels en una dimensión (línea de texels)
  - Texturas 2D: matriz de texels (imagen en 2 dimensiones)
  - Texturas 3D: matriz de tres dimensiones (colección de imágenes o volumen de texels)
  - Texturas Cubemap: conjunto de 6 superficies distribuidas como las caras de un cubo.
- OpenGL identifica el tipo de textura mediante las constantes GL_TEXTURE_1D, GL_TEXTURE_2D , GL_TEXTURE_3D y GL_TEXTURE_CUBE_MAP.

- Textura Cubemap:





- Para trabajar con texturas se definen *objetos textura* que definen buffers de datos que se almacenan en la tarjeta gráfica. Cada *objeto textura* se identifica mediante un número.
- Para reservar identificadores de *objetos textura* se utiliza

glGenTextures(GLsizei n, GLuint* textures);

donde *n* es el número se identificadores a reservar y *textures* el vector que almacena los identificadores.

- Para saber si un identificador está reservado como textura se utiliza

glIsTexture(GLuint texture);

- Para seleccionar el objeto textura sobre el que se va a trabajar se utiliza

glBindTexture(GLenum target, GLuint texture);

donde *target* indica el tipo de textura (GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D o GL_TEXTURE_CUBE_MAP) y *texture* el identificador del objeto textura a seleccionar.

- Para eliminar objetos textura se utiliza

glDeleteTextures(GLsizei n, GLuint* textures);

- OpenGL permite trabajar simultaneamente con varias texturas.
- Las texturas activas se identifican mediante códigos:

GL_TEXTURE0, GL_TEXTURE1, …

- Para activar un determinado código de textura se utiliza el comando

glActiveTexture(GLenum tex);

- Para asociar un objeto textura a un código de textura hay que activar el código y a continuación seleccionar el objeto textura con

glBindTexture(GLenum target, GLuint texture);

- Para cargar una textura 1D en memoria se utiliza la función

```
glTexImage1D(   GLenum target,
                GLint level,
                GLint internalFormat,
                GLsizei width,
                GLint border,
                GLenum format,
                GLenum type,
                const GLvoid * data);
```

- Para cargar una textura 2D en memoria se utiliza la función

```
glTexImage2D(   GLenum target,
                GLint level,
                GLint internalFormat,
                GLsizei width,
                Glsizei height,
                GLint border,
                GLenum format,
                GLenum type,
                const GLvoid * data);
```

- Para cargar una textura 3D en memoria se utiliza la función

```
glTexImage3D( GLenum target,
              GLint level,
              GLint internalFormat,
              GLsizei width,
              GLsizei height,
              GLsizei depth,
              GLint border,
              GLenum format,
              GLenum type,
              const GLvoid * data);
```

- Los valores de *target* indican el tipo de textura que se está cargando. Pueden ser:
  - GL_TEXTURE_1D
  - GL_TEXTURE_2D
  - GL_TEXTURE_3D
  - GL_TEXTURE_CUBE_MAP_POSITIVE_X
  - GL_TEXTURE_CUBE_MAP_NEGATIVE_X
  - GL_TEXTURE_CUBE_MAP_POSITIVE_Y
  - GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
  - GL_TEXTURE_CUBE_MAP_POSITIVE_Z
  - GL_TEXTURE_CUBE_MAP_NEGATIVE_Z

- El parámetro *level* indica el nivel de "mipmap" de la textura cargada (valor 0 por defecto).
- El parámetro *internalFormat* indica el formato en el que se va a almacenar la textura. Los más comunes son GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB y GL_RGBA. También pueden indicarse formatos comprimidos.
- Los parámetros *width*, *height* y *depth* indican el tamaño de la textura. En las versiones anteriores a OpenGL 2.0 se exigía que fueran potencias de 2 (1,2, 4, 8, 16, 32, 64, …). En versiones posteriores no se exige, aunque es aconsejable.

- El parámetro *border* indica el número de texels de borde. El borde se entiende como una extensión de la textura. Se trata de una propiedad eliminada (*deprecated*) que debe asignarse siempre a 0.
- Los parámetros *format*, *type* y *data* describen los datos que se van a almacenar en memoria.

- El parámetro *format* indica que es lo que representa cada elemento de la matriz de píxeles. Se admiten los siguientes valores

GL_COLOR_INDEX	GL_STENCIL_INDEX
GL_DEPTH_COMPONENT	GL_RGB
GL_BGR	GL_RGBA
GL_BGRA	GL_RED
GL_GREEN	GL_BLUE
GL_ALPHA	GL_LUMINANCE
GL_LUMINANCE_ALPHA	

- El campo *type* indica el tipo de dato de cada elemento de la matriz

Tipo	Contenido
GL_UNSIGNED_BYTE	unsigned 8-bit integer
GL_BYTE	signed 8-bit integer
GL_BITMAP	single bits in unsigned 8-bit integers
GL_UNSIGNED_SHORT	unsigned 16-bit integer
GL_SHORT	signed 16-bit integer
GL_UNSIGNED_INT	unsigned 32-bit integer
GL_INT	32-bit integer
GL_FLOAT	single-precision floating-point
GL_UNSIGNED_BYTE_3_3_2	unsigned 8-bit integer
GL_UNSIGNED_BYTE_2_3_3_REV	unsigned 8-bit integer with reversed component ordering
GL_UNSIGNED_SHORT_5_6_5	unsigned 16-bit integer
GL_UNSIGNED_SHORT_5_6_5_REV	unsigned 16-bit integer with reversed component ordering

- El campo *type* indica el tipo de dato de cada elemento (continuación)

Tipo	Contenido
GL_UNSIGNED_SHORT_4_4_4_4	unsigned 16-bit integer
GL_UNSIGNED_SHORT_4_4_4_4_REV	unsigned 16-bit integer with reversed component ordering
GL_UNSIGNED_SHORT_5_5_5_1	unsigned 16-bit integer
GL_UNSIGNED_SHORT_1_5_5_5_REV	unsigned 16-bit integer with reversed component ordering
GL_UNSIGNED_INT_8_8_8_8	unsigned 32-bit integer
GL_UNSIGNED_INT_8_8_8_8_REV	unsigned 32-bit integer with reversed component ordering
GL_UNSIGNED_INT_10_10_10_2	unsigned 32-bit integer
GL_UNSIGNED_INT_2_10_10_10_REV	unsigned 32-bit integer with reversed component ordering

- El campo *data* contiene el puntero a los datos que describen la imagen. Cada elemento del array es del tipo indicado en el campo *type* y contiene la información descrita en el campo *format*.
- OpenGL no contiene funciones para leer imágenes a partir de ficheros en formatos estándar (GIF, PNG, JPEG, …). Es responsabilidad del programador obtener la representación de las imágenes como vectores de píxeles.
- Existen bibliotecas de libre distribución que permiten generar pixmaps a partir de ficheros en formatos estándar, por ejemplo, FreeImage.

- FreeImage es una biblioteca de código abierto que permite cargar imágenes en formatos estándar, de manera que puedan ser incorporadas facilmente a OpenGL.

```
FREE_IMAGE_FORMAT format = FreeImage_GetFileType("stone.jpg", 0);
FIBITMAP* bitmap = FreeImage_Load(format, file_name);
FIBITMAP *pImage = FreeImage_ConvertTo32Bits(bitmap);
int nWidth = FreeImage_GetWidth(pImage);
int nHeight = FreeImage_GetHeight(pImage);
(void*) pixmap = FreeImage_GetBits(pImage);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, nWidth, nHeight,
                        0, GL_BGRA, GL_UNSIGNED_BYTE, pixmap);
FreeImage_Unload(pImage);
```

- Además de almacenarlas directamente, las imágenes que forman las texturas se pueden copiar del color buffer

```
glCopyTexImage1D(GLenum target,  GLint level,
                 GLint internalFormat,
                 GLint x, GLint y,
                 GLsizei width, GLint border);
glCopyTexImage2D(GLenum target,  GLint level,
                 GLint internalFormat,
                 GLint x, GLint y,
                 GLsizei width, GLsizei height,GLint border);
```

- Donde *x* e *y* indican la posición del color buffer desde la que se copia.

- Se puede especificar cual es el buffer de color del que van a tomar los datos las funciones *glCopyTexImage1D()* y *glCopyTexImage2D()*.

glReadBuffer(GLenum mode);

- Los valores de *mode* pueden ser los siguientes: GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT y GL_AUXi (donde *i* es un número entre 0 y GL_AUX_BUFFERS-1).
- Cuando se utiliza doble buffer, *front* se refiere al buffer mostrado y *back* se refiere al buffer de trabajo. Las opciones *left* y *right* se refieren a configuraciones de imágenes estereoscópicas.
- Para doble buffer y visión no estereoscópica el valor por defecto es GL_BACK.

- Cargar texturas es un proceso lento. En ocasiones conviene cargar solo un trozo de textura sobre la textura ya existente en vez de realizar una carga completa.

```
glTexSubImage1D(GLenum target, GLint level,
                GLint xOffset, GLsizei width,
                GLenum format, GLenum type, const GLvoid * data);

glTexSubImage2D(GLenum target, GLint level,
                GLint xOffset, GLint yOffset,
                GLsizei width, GLsizei height,
                GLenum format, GLenum type, const GLvoid * data);
```

```
glTexSubImage3D(GLenum target, GLint level,
                GLint xOffset, GLint yOffset, GLint zOffset,
                GLsizei width, GLsizei height, GLsizei depth,
                GLenum format, GLenum type, const GLvoid * data);
```

- Los valores de *xOffset*, *yOffset* y *zOffset* indican la posición en la que comienza la susitución de la textura existente.

- También se puede copiar trozos de texturas con valores tomados desde el color buffer:

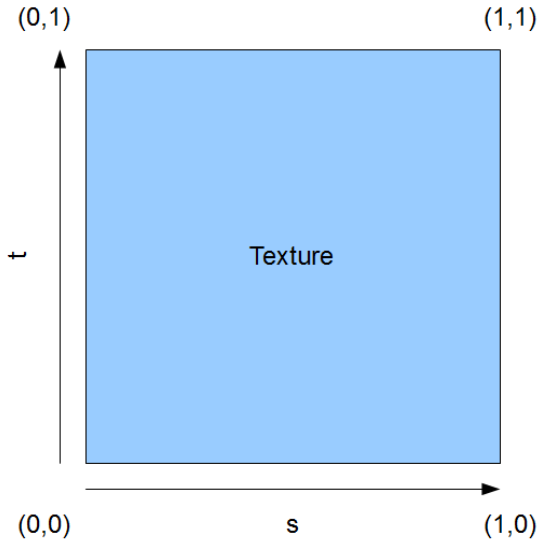
```
glCopyTexSubImage1D(GLenum target,  GLint level,
                    GLint xOffset,
                    GLint x, GLint y,
                    GLsizei width);
glCopyTexSubImage2D(GLenum target,  GLint level,
                    GLint xOffset, GLint yOffset,
                    GLint x, GLint y,
                    GLsizei width, GLsizei height);
```

```
glCopyTexSubImage3D(GLenum target,  GLint level,
                    GLint xOffset, GLint yOffset, GLint zOffset,
                    GLint x, GLint y,
                    GLsizei width, GLsizei height);
```

- En el caso de las texturas 3D, la copia de la imagen del color buffer se hace en un único nivel de profundidad marcado por *zOffset*.

- Para identificar un punto en una textura se utilizan coordenadas. OpenGL las denomina (s,t,r,q) con un significado similar a las coordenadas espaciales (q es un factor de escala). Generalmente se utilizan valores en coma flotante, entre 0.0f y 1.0f.
- La coordenada de la textura es una información asociada a los vértices. Normalmente se incluye como atributo de los vértices y se propaga desde el VertexShader al FragmentShader. De esta forma cada pixel del FragmentShader recibe las coordenadas de textura interpoladas a partir de las coordenadas de los vértices de la primitiva.

- Las texturas 1D utilizan solo la coordenada S.
- Las texturas 2D utilizan las coordenadas S y T.
- Las texturas 3D utilizan las coordenadas S, T y R.
- El valor (0,0) corresponde a la esquina inferior izquierda de la imagen:



- Las texturas Cubemap utilizan las tres coordenadas. Para seleccionar la cara del cubo se busca la coordenada con el valor mayor (*major axis direction*). A partir de ahí se escoge la cara y se determinan las coordenadas 2D a aplicar en esa cara (sc, tc).

Major axis	Target	sc	tc
+rx	GL_TEXTURE_CUBE_MAP_POSITIVE_X	-rz	-ry
-rx	GL_TEXTURE_CUBE_MAP_NEGATIVE_X	+rz	-ry
+ry	GL_TEXTURE_CUBE_MAP_POSITIVE_Y	+rx	+rz
-ry	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	+rx	-rz
+rz	GL_TEXTURE_CUBE_MAP_POSITIVE_Z	+rx	-ry
-rz	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	-rx	-ry

$$s = (sc/|ma| + 1) / 2$$

$$t = (tc/|ma| + 1) / 2$$

- Texturas en el VertexShader

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexNormal;
layout(location = 2) in vec2 VertexTexCoord;

uniform mat4 MVP;
uniform mat4 ModelView;

out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

void main() {
  Normal = vec3(ModelView*vec4(VertexNormal,0.0));
  Position = vec3(ModelView*vec4(VertexPosition,1.0));
  TexCoord = VertexTexCoord;
  gl_Position = MVP * vec4( VertexPosition, 1.0);
}
```

- Texturas en el FragmentShader
  - Se introducen en el FragmentShader como variables uniformes de un tipo especial (*sampler1D, sampler2D, sampler3D, samplerCube*).
  - Se puede acceder al texel correspondiente a unas coordenadas de textura por medio de la función predefinida *texture()*.

```
#version 400

in vec2 TexCoord;
uniform sampler2D Tex1;

out vec4 FragColor;

void main()
{
    FragColor = texture(Tex1, TexCoord);
}
```

- Texturas en el FragmentShader
  - Se puede utilizar más de una textura al mismo tiempo (multi-textura).

```
#version 400

in vec2 TexCoord;
uniform sampler2D BaseTex;
uniform sampler2D ModTex;

out vec4 FragColor;

void main()
{
    vec4 BaseColor = texture(BaseTex, TexCoord);
    vec4 ModColor = texture(ModTex, TexCoord);
    FragColor = mix(BaseColor, ModColor, Modcolor.a);
}
```

  - La función predefinida *mix()* genera una mezcla de colores utilizando el tercer parámetro como porcentaje de mezcla.

- Asignación de texturas
  - Para asignar los valores a las variables uniformes de textura (por ejemplo, BaseTex y ModTex) se necesita definir las texturas activas, buscar la posición de las variables uniformes y enlazar las texturas activas a las posiciones correspondientes.

- Asignación de texturas

```
Gluint baseTexId, modTexId;
glGenTextures(1, &baseTexId);
glGenTextures(1, &modTexId);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, baseTexId);
glTexImage2D(GL_TEXTURE_2D, …);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, modTexId);
glTexImage2D(GL_TEXTURE_2D, …);

int baseTexLoc = glGetUniformLocation(program, "BaseTex");
if(baseTexLoc >=0) glUniform1i(baseTexLoc, 0);

int modTexLoc = glGetUniformLocation(program, "ModTex");
if(modTexLoc >=0) glUniform1i(modTexLoc, 1);
```

- El proceso de aplicación de texturas se puede configurar por medio del comando *glTexParameter..(),* que permite modificar algunos parámetros del proceso de aplicación de texturas.

glTexParameterf(GLenum target, GLenum pname, GLfloat param);

glTexParameteri(GLenum target, GLenum pname, GLint param);

glTexParameterfv(GLenum target, GLenum pname, Glfloat* param);

glTexParameteriv(GLenum target, GLenum pname, Glint* param);

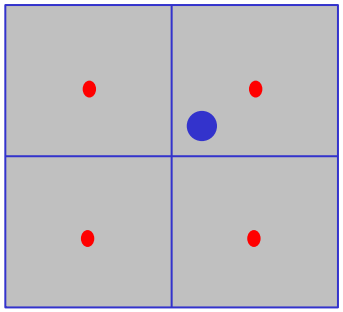
- *target* indica el tipo de textura (GL_TEXTURE_1D, …).
- *pname* indica el parámetro a configurar.
- *param* indica el valor asignado al parámetro.

- *pname* puede ser

GL_TEXTURE_MIN_FILTER	GL_TEXTURE_MAG_FILTER
GL_TEXTURE_BASE_LEVEL	GL_TEXTURE_MAX_LEVEL
GL_TEXTURE_MIN_LOD	GL_TEXTURE_MAX_LOD
GL_TEXTURE_LOD_BIAS	GL_TEXTURE_WRAP_S
GL_TEXTURE_WRAP_T	GL_TEXTURE_WRAP_R
GL_TEXTURE_COMPARE_FUNC	GL_TEXTURE_COMPARE_MODE
GL_TEXTURE_SWIZZLE_R	GL_TEXTURE_SWIZZLE_G
GL_TEXTURE_SWIZZLE_B	GL_TEXTURE_SWIZZLE_A
GL_DEPTH_STENCIL_TEXTURE_MODE	

- Para aplicar una textura sobre una primitiva geométrica se calculan las coordenadas de textura correspondientes a cada pixel de la primitiva y se busca en la textura el color correspondiente a esas coordenadas. Esto se conoce como *texture-filtering*.
- Típicamente, el tamaño de la textura y el tamaño de la primitiva serán diferentes y al filtrar tendremos que agrandar la textura o encogerla.
- OpenGL define formas de filtrado independientes para agrandar o encoger las texturas.

- Para configurar el proceso de filtrado se utiliza la función *glTexParameter…()* con los parámetros GL_TEXTURE_MAG_FILTER y GL_TEXTURE_MIN_FILTER.
- Los valores típicos para los filtros son GL_NEAREST y GL_LINEAR.
- El filtro GL_NEAREST se limita a asignar al pixel el color del texel más cercano a sus coordenadas de textura. Este filtro puede provocar problemas de aliasing. El filtro GL_LINEAR realiza una interpolación lineal entre los cuatro texels vecinos.

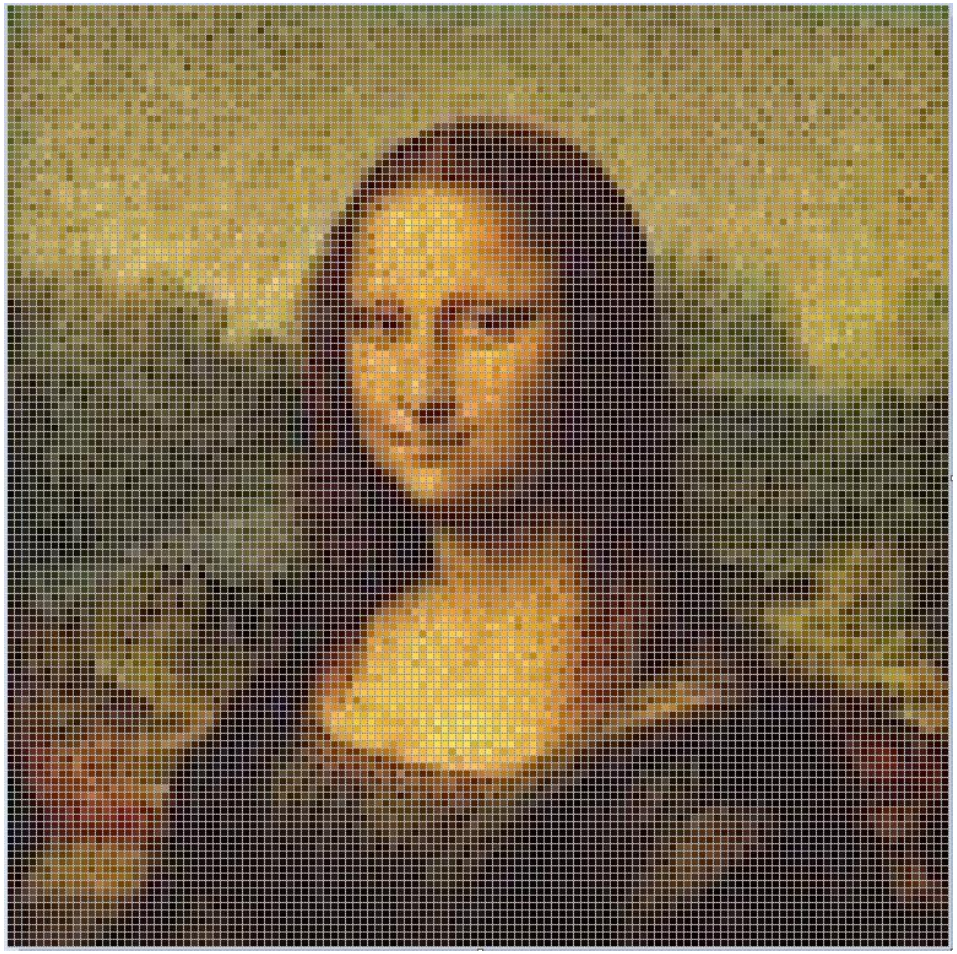


- Las texturas se dimensionan entre 0.0f y 1.0f. Sin embargo, las coordenadas de textura no tienen por qué estar incluidas entre estos límites. Esto puede ocurrir al utilizar la opción GL_LINEAR o al introducir estos valores como atributos de los vértices.
- El comportamiento de los filtros de textura más allá de los límites se denomina *texture wrapping*. OpenGL permite configurar comportamientos diferentes en cada dimensión.
- Para configurarlos se utilizan los parámetros GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T y GL_TEXTURE_WRAP_R.

- Los valores de estos parámetros pueden ser los siguientes.
  - GL_REPEAT: genera una repetición de la textura.
  - GL_CLAMP_TO_BORDER: utiliza el color del borde. El color del borde se puede asignar con el parámetro GL_TEXTURE_BORDER_COLOR.
  - GL_CLAMP_TO_EDGE: utiliza los texels de los extremos, es decir, repite el último texel en cada dimensión.
  - GL_MIRRORED_REPEAT: genera una repetición especular de la textura.
  - GL_MIRROR_CLAMP_TO_EDGE: genera una única repetición especular de la textura y a partir de ahí se comporta como la opción GL_CLAMP_TO_EDGE.

- Técnicas de uso de texturas para algunos efectos avanzados
  - Se puede modificar el FragmentShader para incluir los efectos de la luz y el uso de texturas. En ese caso se suele separar el efecto de la luz ambiental y difusa (que se multiplica a la textura) del efecto de la luz especular (que se suma al total).
  - Se puede utilizar la información de la segunda textura como modificación de los vectores normales para calcular el efecto de la luz. Esto se conoce como *normal maps*.
  - Se puede utilizar la segunda textura describir el entorno (un cubemap) y calcular el reflejo del entorno sobre el objeto. También se pueden introducir efectos de refracción para simular objetos translúcidos.
  - Se pueden utilizar ecuaciones de Fresnel para calcular un factor de reflexión (la luz se refleja más cuando incide en un ángulo grande).

- Cuando la primitiva geométrica resulta mucho más pequeña que el tamaño de la textura se produce un efecto llamado *scintillation*. El problema es que si el fragmento es pequeño, dos píxeles consecutivos pueden corresponder a texels alejados y el filtrado provoca efectos de aliasing. Este defecto resulta más visible cuando los objetos se encuentran en movimiento.
- Para evitar este problema se utilizan *mipmaps*. El término *mip* procede de la expresión latina "*multum in parvo*" (mucho en poco). La técnica consiste en tener versiones de la textura de tamaños inferiores y aplicar en cada caso el nivel de mipmap adecuado al tamaño del fragmento.



- Cada nivel de mipmap se obtiene dividiendo entre dos cada dimensión de la textura del nivel anterior.
- Por ejemplo, si tenemos una textura de 32x16, el nivel 1 tendría un tamaño 16x8, el nivel 2 sería 8x4, el nivel 3 sería 4x2, el nivel 4 sería 2x1 y el nivel 5 sería 1x1. (Esta es la razón por la que los tamaños deben ser potencias de 2)
- El mipmapping supone un gasto adicional en memoria. Para texturas 1D la memoria aumenta un 100%. Para texturas 2D la memoria necesaria es un 33% mayor. Para texturas 3D el aumento es de un 15%.
- Los mipmaps se cargan con la funciones *glTexImage..()* y derivadas, indicando el nivel de mipmap en el argumento *level*.

- Por defecto, para utilizar mipmapping es necesario que todos los niveles de mipmap estén cargados. Se puede indicar un rango de niveles de mipmap obligatorios con los parámetros de textura GL_TEXTURE_BASE_LEVEL y GL_TEXTURE_MAX_LEVEL.
- Por ejemplo, para configurar un mipmapping con los niveles obligatorios de 0 a 4.

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL , 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
```

- También se puede configurar los niveles de mipmap (*level of detail*) que van a ser utilizados realmente con los parámetros GL_TEXTURE_MIN_LOD y GL_TEXTURE_MAX_LOD.

level of detail

- El uso de mipmaps añade un nuevo aspecto a las opciones de filtrado. Al considerar las componentes de textura de un pixel podemos encontrarnos entre dos niveles de mipmap. En este caso podemos calcular el color final de la textura con el mipmap más cercano o con una interpolación lineal entre los dos mipmaps.
- Las opciones de filtrado pueden ser entonces

GL_NEAREST	GL_LINEAR
GL_NEAREST_MIPMAP_NEAREST	GL_LINEAR_MIPMAP_NEAREST
GL_NEAREST_MIPMAP_LINEAR	GL_LINEAR_MIPMAP_LINEAR

- Para utilizar mipmaps es necesario generar la imagen correspondiente a cada nivel. Esto puede realizarse con alguna herramienta de edición de imágenes y cargarse con *glTexImage…()* o generarse de forma automática.
- OpenGL permite generar automáticamente los mipmaps utilizando las funciones

glGenerateMipmap(GLenum target);

glGenerateTextureMipmap(Gluint texture);

- Las texturas pueden llegar a ocupar una cantidad de memoria enorme. Los videojuegos modernos pueden ocupar más de un Gigabyte de memoria en cada nivel del juego.
- Una forma de mejorar el almacenamiento en memoria de las texturas es utilizar formatos comprimidos. De esta forma la cantidad de memoria utilizada es menor y se reducen los accesos a memoria.
- La contrapartida es que los datos deben ser descomprimidos para poder utilizarlos. Para que esto no suponga una ralentización del renderizado se utilizan métodos que permiten una descompresión rápida.
- Por esta razón, los formatos de compresión utilizados no son tan potentes como los incluidos en otros formatos de imagen (como JPEG).

- OpenGL permite almacenar las texturas en formatos comprimidos. Los valores de *internalFormat* que indican formatos comprimidos son (comenzando con el prefijo (GL_COMPRESSED_)

RED	RG
RGB	RGBA
SRGB	SRGB_ALPHA
RED_RGCT1	SIGNED_RED_RGTC1
RG_RGTC2	SIGNED_RG_RGTC2
RGBA_BPTC_UNORM	SRGB_ALPHA_BPTC_UNORM
RGB_BPTC_SIGNED_FLOAT	RGB_BPTC_UNSIGNED_FLOAT
RGB8_ETC2	SRGB8_ETC2
RGB8_PUNCHTHROUGH_ALPHA1_ETC2	SRGB8_PUNCHTHROUGH_ALPHA1_ETC2
RGBA8_ETC2_EAC	SRGB8_ALPHA8_ETC2_EAC
R11_EAC	SIGNED_R11_EAC
RG11_EAC	SIGNED_RG11_EAC

- Los formatos RED, RG, RGB, RGBA, SRGB y SRGB_ALPHA son genéricos. Esto quiere decir que cada fabricante puede desarrollarlos como quiera.

- Si al cargar una textura se indica que el formato interno es comprimido, OpenGL realiza automáticamente la compresión de los datos. Sin embargo esto supone ralentizar el proceso de carga de las texturas.

- Se puede obtener la versión comprimida de la imagen almacenada con la función

```
glGetCompressedTexImage(GLenum  target,
                        GLint  lod,
                        GLvoid *  img);
```

donde *target* corresponde a la textura que se pretende leer, *lod* es el nivel de mipmap e *img* el puntero a los datos.

- Estos datos se pueden almacenar externamente y cargarlos directamente. En ocasiones esta carga puede crear problemas si el fabricante de la tarjeta gráfica utiliza una versión no estándar del algoritmo de compresión.

- Para cargar directamente los datos ya comprimidos se utilizan las funciones:

```
glCompressedTexImage1D(
                        GLenum target,
                        GLint level,
                        GLenum internalFormat,
                        GLsizei width,
                        GLint border,
                        GLsizei imageSize,
                        void*  data);
```

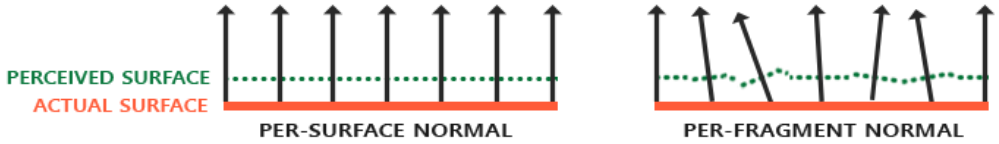
algoritmo de compresión

```
glCompressedTexImage2D(
                        GLenum target,
                        GLint level,
                        GLenum internalFormat,
                        GLsizei width,
                        GLsizei height,
                        GLint border,
                        GLsizei imageSize,
                        void* data);
```

```
glCompressedTexImage3D(
                        GLenum target,
                        GLint level,
                        GLenum internalFormat,
                        GLsizei width,
                        GLsizei height,
                        GLsizei depth,
                        GLint border,
                        GLsizei imageSize,
                        void*  data);
```

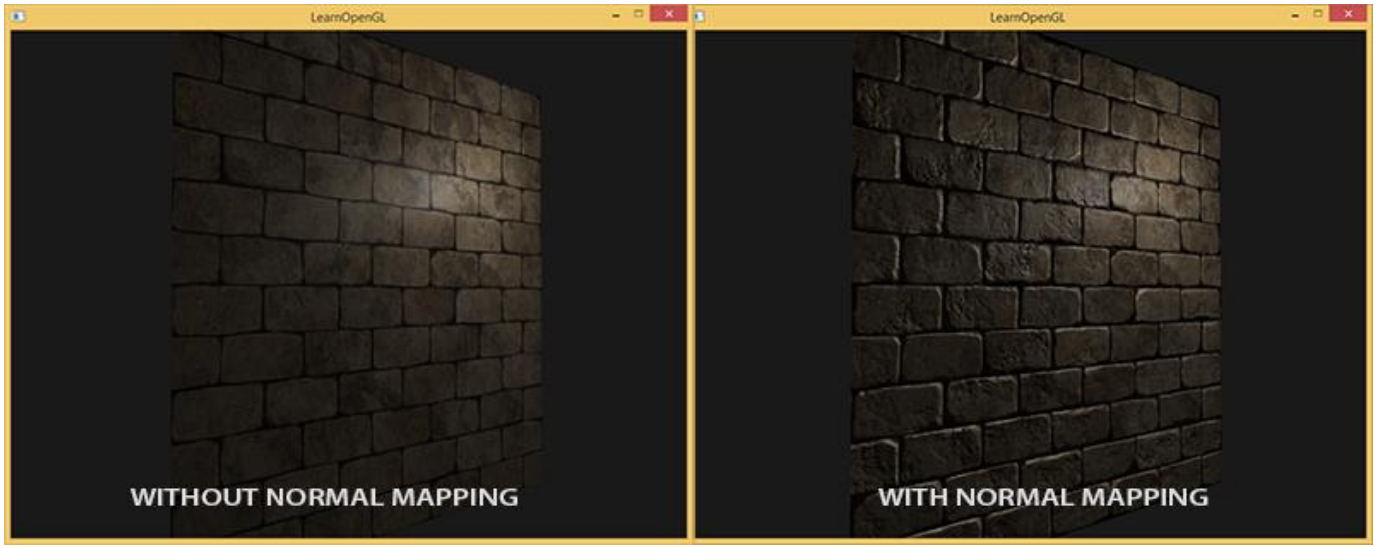
- Las texturas permiten proyectar una imagen sobre una primitiva, pero esa proyección es plana. (Es como colocar una pegatina sobre una superficie plana).
- Si queremos aumentar el nivel de detalle de la superficie para incorporar rugosidades o grietas habría que dividir la superficie en primitivas más pequeñas que consiguieran modelar esas rugosidades. Esto supone aumentar considerablemente el número de vértices y primitivas necesarias para modelar las superficies.
- Si observamos las superficies tangencialmente podemos apreciar la rugosidad sobre el perfil de la superficie, pero si las observamos de forma perpendicular la rugosidad se nota básicamente en la forma en que el cuerpo es iluminado.

- Por tanto, se puede modelar el efecto de la rugosidad si consideramos que las normales en cada punto de la primitiva no son iguales sino que cada punto tiene una normal diferente.



- Para almacenar la información sobre la normal asociada a cada pixel se utiliza una textura, pero en este caso la información almacenada no se interpreta como un color RGB sino como las coordenadas del vector normal.

- La utilización de texturas como forma de almacenar los vectores normales para simular rugosidad se conoce como ***normal mapping*** o ***bump mapping***.



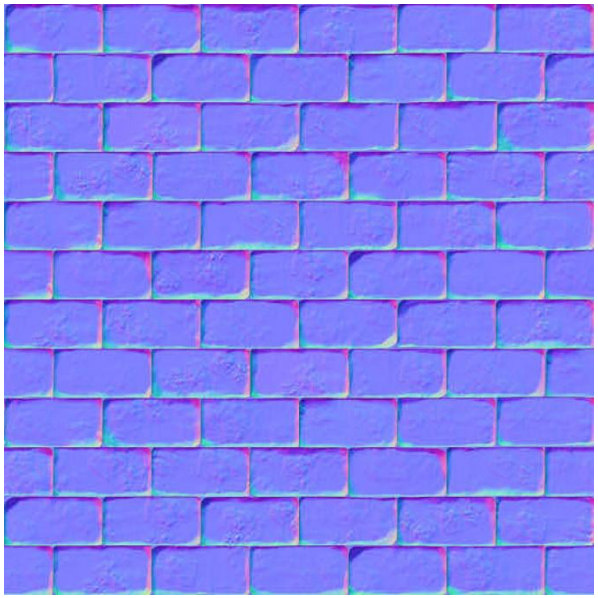
- Para representar el vector normal es necesario fijar un sistema de coordenadas. Habitualmente se utiliza un sistema de coordenadas local a cada punto y tangente a la superficie. De esta forma, el eje X se considera paralelo a la coordenada U de la textura; el eje Y se considera paralelo al eje V de la textura; y el eje Z se considera perpendicular a la primitiva.
- Las coordenadas *xyz* del vector normal se almacenan como coordenadas *rgb* de la textura. La mayor parte de los puntos suele tener una dirección normal perpendicular a la primitiva (es decir, sobre el eje Z en el plano tangente) por lo que la interpretación de las texturas como colores suele dar un tono azul en la mayoría de los puntos.

$(coord/2)+0.5$

- Ejemplo de la textura utilizada en el muro

imagen de la textura

Normal Map.



- VertexShader

```
#version 400

layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;
layout (location = 3) in vec3 VertexTangent;

struct LightInfo {
    vec3 Ldir;
    vec3 La;
    vec3 Ld;
    vec3 Ls;
};
uniform LightInfo Light;

uniform mat4 ModelViewMatrix;
uniform mat4 ViewMatrix;
uniform mat4 MVP;
```

- VertexShader

```
out vec3 LightDir;
out vec2 TexCoord;
out vec3 ViewDir;

void main(void)
{
  vec4 n4 = ModelViewMatrix*vec4(VertexNormal,0.0);
  vec4 t4 = ModelViewMatrix*vec4(VertexTangent,0.0);
  vec4 v4 = ModelViewMatrix*vec4(VertexPosition,1.0);

  // Vectores unitarios del espacio tangente
  vec3 N = normalize( vec3(n4) );
  vec3 T = normalize( vec3(t4) );
  vec3 B = normalize( cross(N,T) );
  …
```

- VertexShader

```
    // Matriz de transformación al espacio tangente
    mat3 TBN = mat3(
                T.x, B.x, N.x,
                T.y, B.y, N.y,
                T.z, B.z, N.z);

    LightDir = normalize(TBN * ViewMatrix * Light.Ldir);
    ViewDir = TBN * normalize( vec3( -v4 ) );
    TexCoord = VertexTexCoord;
    gl_Position = MVP * vec4( VertexPosition, 1.0);
}
```

- FragmentShader

```
#version 400

in vec3 LightDir;
in vec2 TexCoord;
in vec3 ViewDir;

layout(binding=0) uniform sampler2D ColorTex;
layout(binding=1) uniform sampler2D NormalMapTex;
layout(binding=2) uniform sampler2D SpecMapTex;

struct LightInfo {
    vec3 Ldir;
    vec3 La;
    vec3 Ld;
    vec3 Ls;
};
uniform LightInfo Light;

…
```

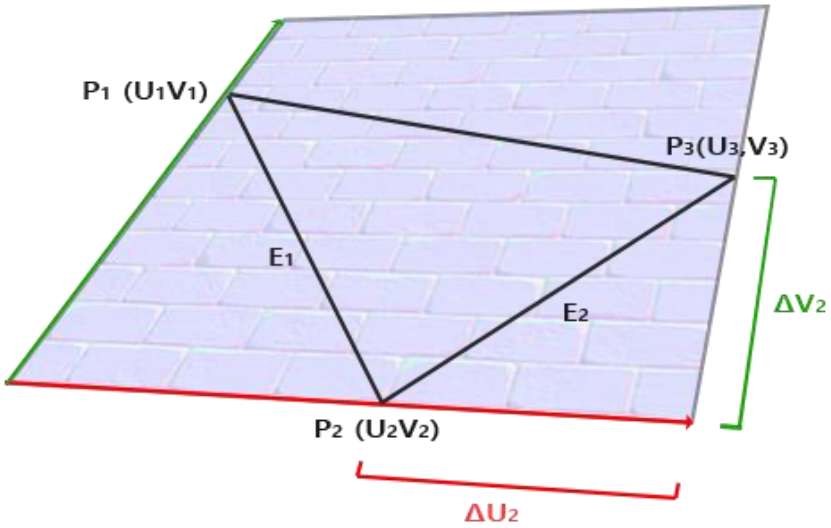
- FragmentShader

```
…

struct MaterialInfo{
    vec3 Ka;
    vec3 Kd;
    vec3 Ks;
    float Shininess;
};
uniform MaterialInfo Material;

out vec4 FragColor;

…
```

- FragmentShader

```
vec3 ads(vec3 norm, vec3 tex, vec3 spec) {
    vec3 r = reflect(LightDir,norm);
    float dRate = max(dot(-LightDir,norm), 0.0);
    float sRate = pow(max(dot(r,ViewDir),0.0),
                      Material.Shininess);
    vec3 ambient = Light.La * Material.Ka;
    vec3 difusse = Light.Ld * Material.Kd * dRate;
    vec3 specular = Light.Ls * Material.Ks * sRate;
    return (ambient + difusse)*tex + spec*specular;
}

void main() {
  vec4 normal = 2.0*texture(NormalMapTex,TexCoord)-1.0;
  vec4 tex = texture( ColorTex, TexCoord );
  vec4 spec = texture( SpecMapTex, TexCoord );
  vec3 ads = ads(normal.xyz, tex.rgb, spec.rgb);
  FragColor = vec4(ads, 1.0);
}
```

- El programa gráfico anterior necesita un nuevo atributo asocado a cada vértice que corresponde al vector tangente al vértice. Este vector tangente corresponde a la dirección del eje U (coordenada de textura horizontal) expresado en coordenadas locales.
- Para calcular la dirección del eje U en un vértice es necesario conocer la posición y coordenadas de textura de los otros dos vértices del triángulo.
- Dados las posiciones de los tres vértices de un triángulo (en coordenadas locales) y las coordenadas de textura de dichos vértices se puede calcular los vectores tangente y bitangente.



- El vector E_1 corresponde a la diferencia entre la posición de los vértices P_2 y P_1. El vector E_2 corresponde a la diferencia entre la posición de los vértices P_3 y P_2.

- Los vectores E1 y E2 se pueden expresar en términos de los vectores tangente T y bitangente B.

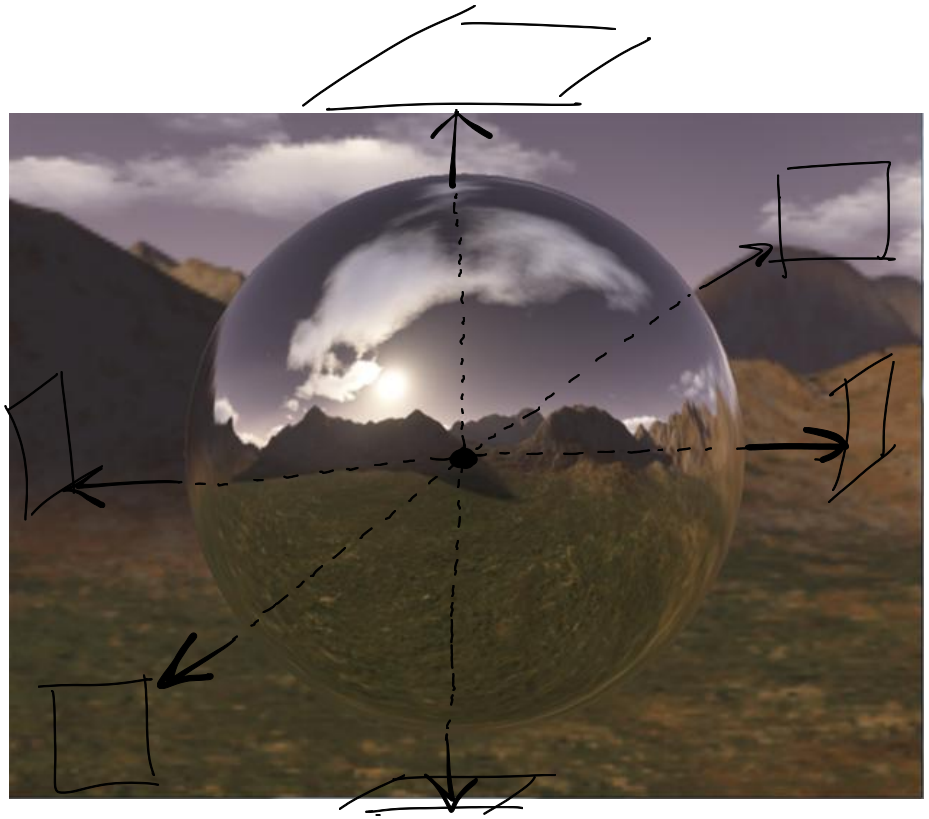
$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$
$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

- A partir de esta ecuación se pueden calcular los valores de T y B.

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

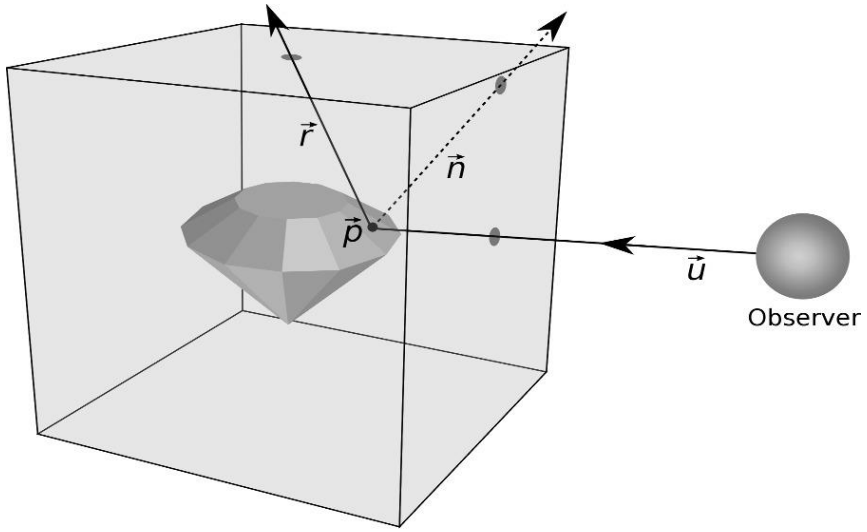
- Esto permite calcular los valores de T y B para cada triángulo. En realidad necesitamos los valores de T y B para cada vértice. Esto se obtiene realizando una media entre los valores de los triángulos a los que pertenece cada vértice.

- Las texturas de tipo Cubemap suelen utilizarse también para generar el efecto de materiales altamente reflectantes (como el cromo).



Para generar el reflejo del entorno tenemos que calcular un nuevo cubemap desde el centro de la esfera y reflejar ese cubemap

- Para ello se utiliza el vector de posición de cada pixel y el vector normal para calcular la dirección de reflejo. El color del pixel se toma de la textura cubemap utilizando la dirección de reflejo como coordenada de textura.



- De forma similar se pueden generar efectos de refracción calculando la dirección de refracción de la luz a partir del vector de posición, el vector normal y el índice de refracción del material.