

# Introducción a las texturas

## Informática Gráfica I

Material de: **Ana Gil Luezas**  
Adaptado por: **Elena Gómez y Rubén Rubio**  
[{mariaelena.gomez,rubenrub}@ucm.es](mailto:{mariaelena.gomez,rubenrub}@ucm.es)



# Contenido

## 1 Definición

- Texturas

## 2 Aplicación de texturas

- Filtros
- Mallas

## 3 Combinación de la textura

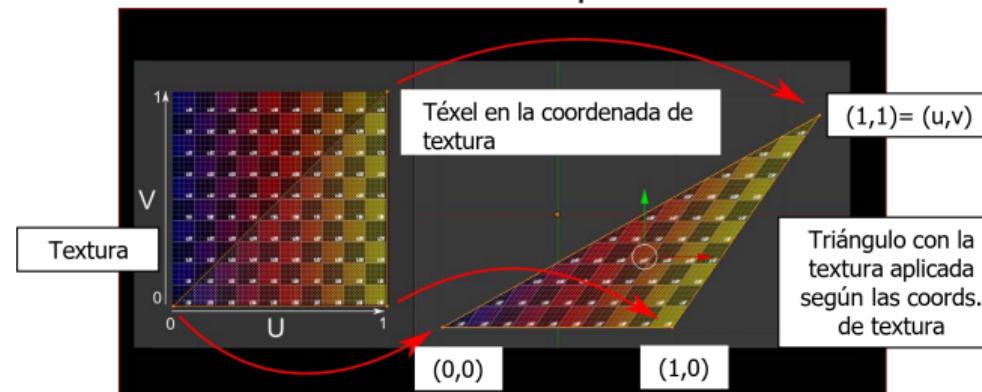
## 4 Clase Texture

- Texturas en OpenGL
- Implementación

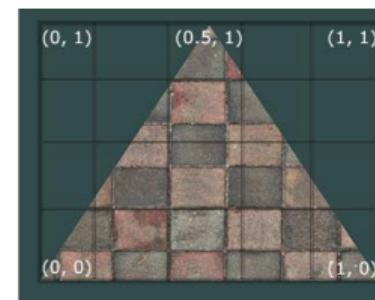
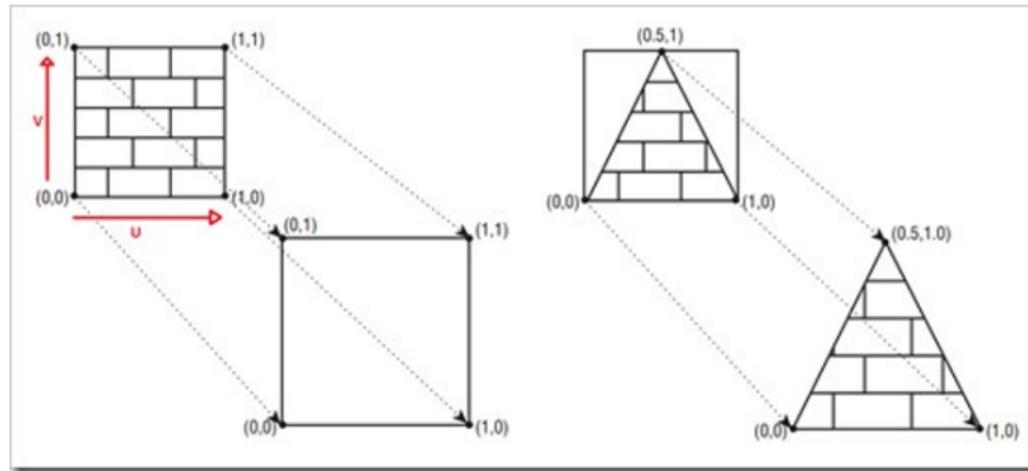
# Texturas

- Una **textura** es una imagen que se aplica a un objeto para darle mayor realismo. Las texturas pueden ser generadas mediante algoritmos (*procedurales*), o a partir de una imagen *rasterizada*.
- Cuando se le añade una textura a una malla, hay que indicar qué parte de la imagen tiene que ser usada para cada triángulo, y para esto se emplean las coordenadas de textura ( $S, T$ ) o ( $U, V$ ).

Cada vértice de la malla tiene que tener, además de su posición, sus coordenadas de textura: un *dvec2*. Estas coordenadas se utilizan para obtener el *téxel* de la imagen:



# Texturas



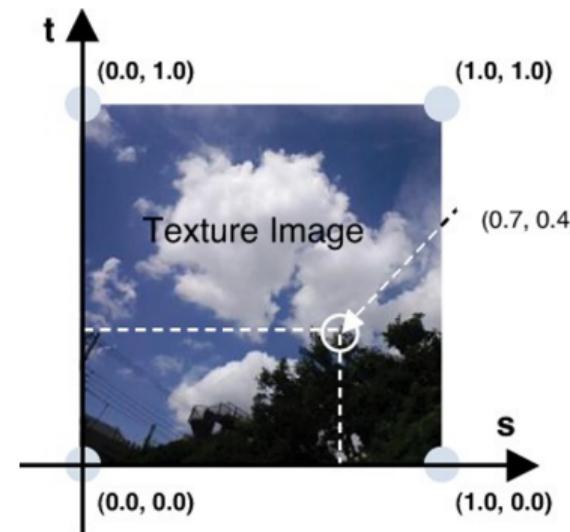
# Definición de texturas

- Una textura 2D es una función de dos parámetros

$T(s, t) : \mathbb{R} \times \mathbb{R} \rightarrow \text{Colores}$  Pero se utiliza la forma normalizada en los intervalos  $[0, 1]$

$T(s, t) : [0, 1] \times [0, 1] \rightarrow \text{Colores}$

Las coordenadas de textura de los píxeles del interior se obtienen por interpolación



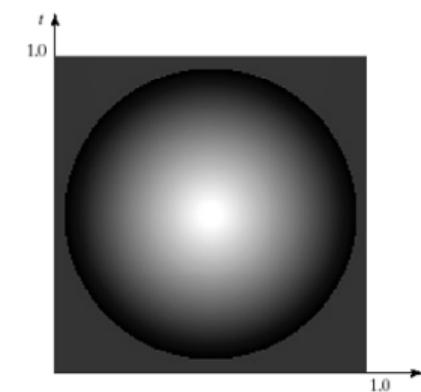
# Definición de texturas

- Definición **procedimental**:

Ejemplo para colores de una sola componente de tipo **double**:

```
double TexturaProc (double s, double t) { // en [0,1]
    double r = 0.4; // radio
    // distancia al centro
    double d = sqrt((s-0.5)*(s-0.5) + (t-0.5)*(t-0.5));
    if (d < r) return 1 - (d / r); // intensidad del círculo
    else return 0.2; // background
}
```

Los puntos dentro del radio del círculo ( $r$ ) son más oscuros en el borde ( $d \approx r$ ) y más claros en el centro ( $d \approx 0.0$ )



# Definición de texturas

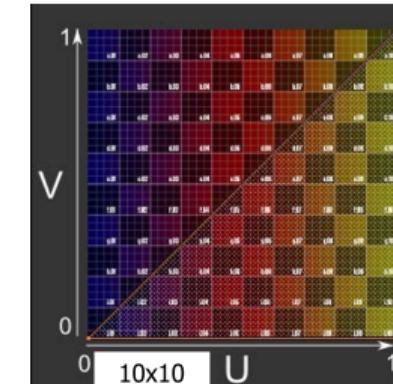
- Definición con imágenes rasterizadas:

Ejemplo para una imagen de  $NC \times NF$  colores RGBA:

```
RGBA TexturaBMP (double s, double t) // en [0,1]
{
    // suponiendo RGBA img[NC][NF]
    return img[trunc(s*NC)][trunc(t*NF)]; // (*)
}
```

Algunos valores se repetirán y otros se saltarán, dependiendo de que sea necesario estirar o encoger la textura al aplicarla sobre una malla.

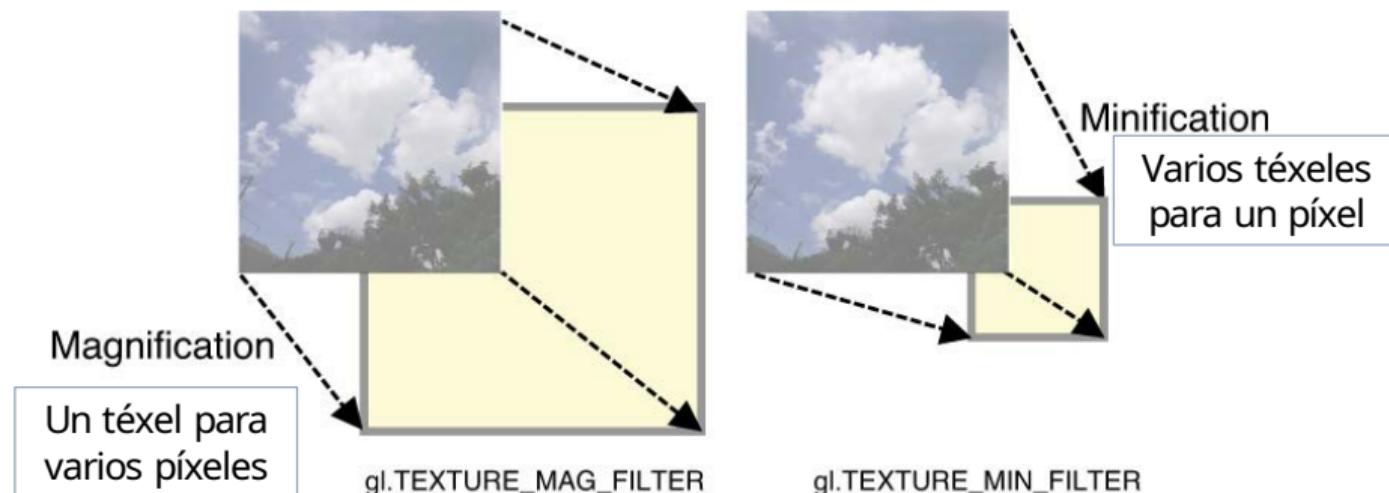
Para evitar estos problemas **se aplican filtros en (\*)**.



# Aplicación de filtros

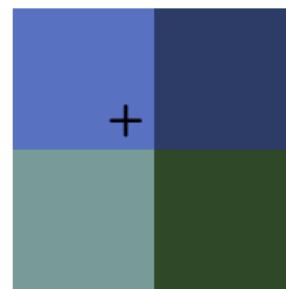
- En caso de tener que aumentar o reducir la imagen durante el renderizado, se pueden **aplicar filtros**.

También se utilizan imágenes de distintas resoluciones ([mipmaps](#)).

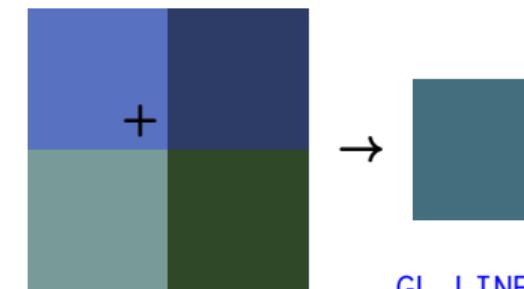


# Aplicación de filtros

- Función para determinar el color correspondiente a las coordenadas de textura de cada píxel:
  - **GL\_NEAREST**: el color del téxel más cercano a las coordenadas de textura.
  - **GL\_LINEAR**: la media ponderada de los colores de los cuatro téxeles más cercanos a las coordenadas de textura.



GL\_NEAREST



GL\_LINEAR

# Aplicación de una textura a una malla

La textura puede aplicarse de varias formas:

- Recubriendo toda la superficie del objeto con la textura.
- Recortando parte de la textura.
- Pegando la textura en una zona concreta de la superficie.

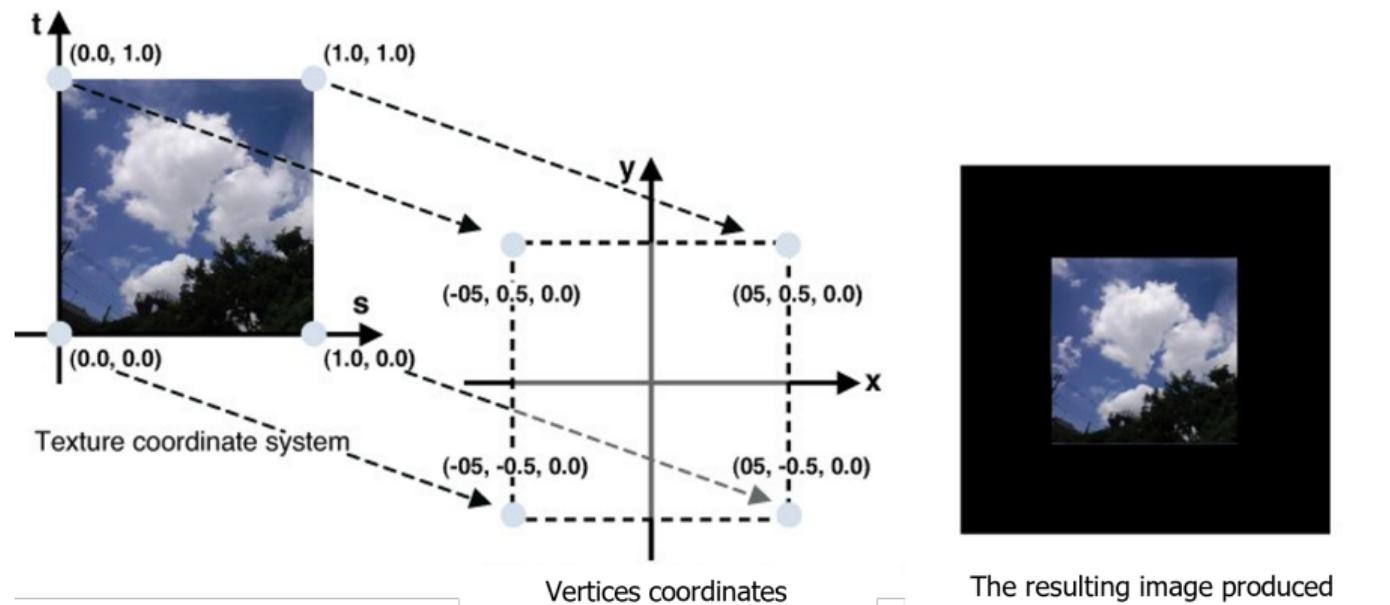
Deben preservarse las proporciones de la textura para evitar distorsiones de la imagen de la textura.

# Aplicación de una textura a una malla

- **Texture mapping:** establecer las coordenadas de textura  $(s, t)$  de cada vértice.

Map: Vértices de la malla  $(\mathbb{R} \times \mathbb{R} \times \mathbb{R}) \rightarrow [0, 1] \times [0, 1]$

OpenGL permite asignar coordenadas fuera del intervalo  $[0, 1]$  ([wrapping](#))



# Aplicación de una textura a una malla

- A cada vértice hay que asignarle sus coordenadas de textura ( $s, t$ ) añadiendo a la clase `Mesh` un vector de coordenadas de textura (análogo al vector de colores pero de 2 coordenadas):

```
std::vector<glm::dvec2> vTexCoords; //vector de coordenadas de textura
```

- El método `Mesh::render()` tiene que activar (y desactivar) el array de coordenadas de textura:

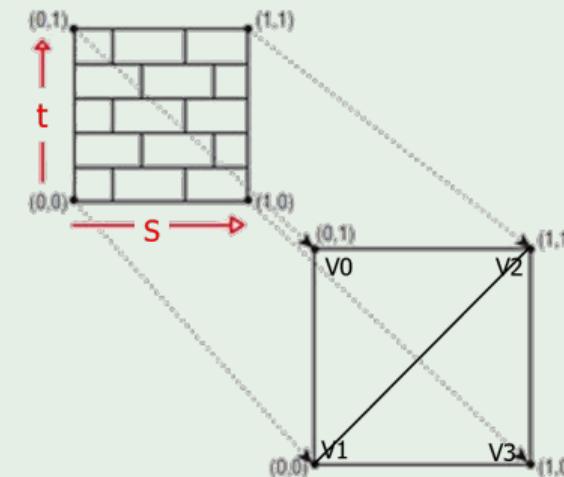
```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
glTexCoordPointer(2, GL_DOUBLE, 0, vTexCoords.data());
```

- Añadimos una nueva clase, `Texture`, con métodos para cargar de archivo una imagen y transferirla a la GPU (`load`), y para activar (`bind`) y desactivar (`unbind`) la textura en la GPU.
- Añadimos a la clase `Abs_Entity` un atributo para la textura (`Texture* mTexture`), que habrá que establecer al crear la entidad (método `setTexture`), y activar/desactivar al renderizarla.

# Aplicación de una textura a una malla

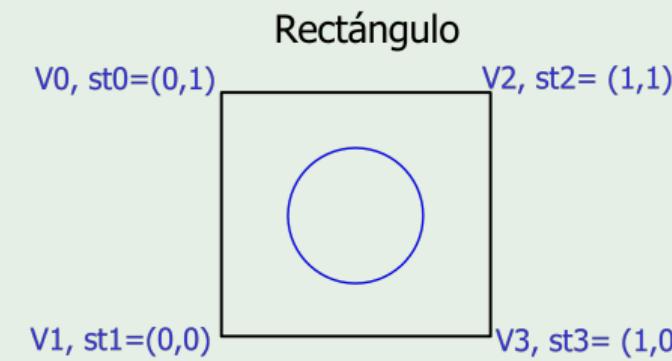
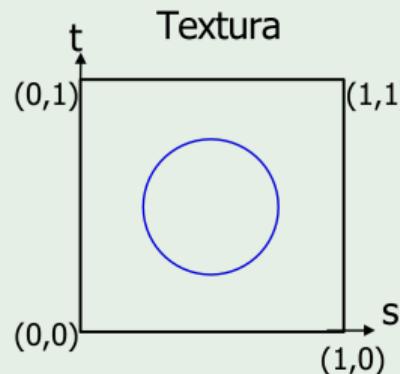
## Ejemplo: Toda la textura en un triángulo

```
Mesh* Mesh::generaRectanguloTexCor(GLdouble w, GLdouble h) {  
    Mesh *m = generaRectangulo(w, h);  
    m->vTexCoords.reserve(m->mNumVertices);  
    m->vTexCoords.emplace_back(0, 1);  
    m->vTexCoords.emplace_back(0, 0);  
    m->vTexCoords.emplace_back(1, 1);  
    m->vTexCoords.emplace_back(1, 0);  
    return m;  
}
```

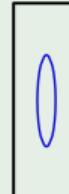


# Aplicación de una textura a una malla

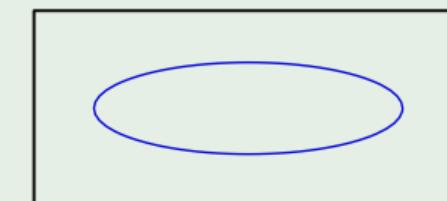
## Ejemplo: Dependiendo de las dimensiones del rectángulo



Rectángulo



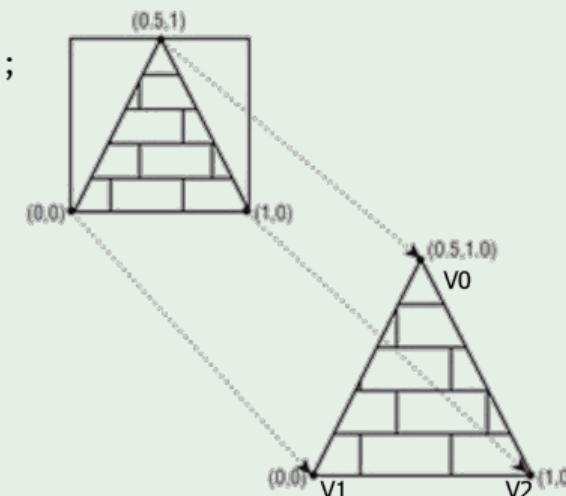
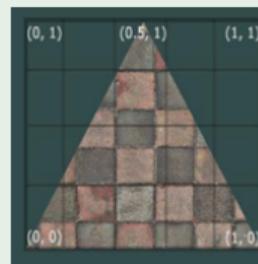
Rectángulo



# Aplicación de una textura a una malla

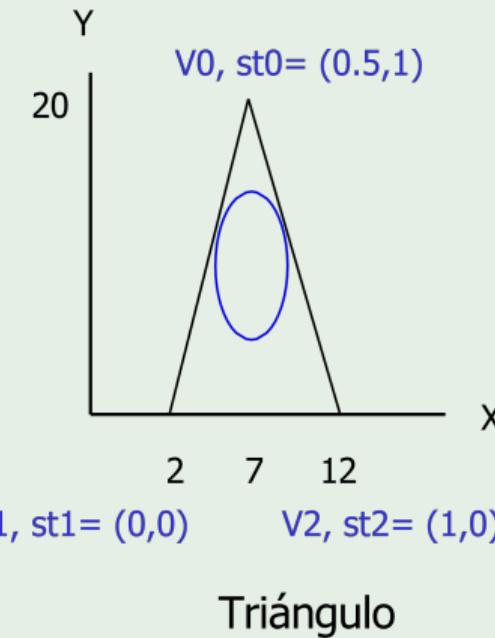
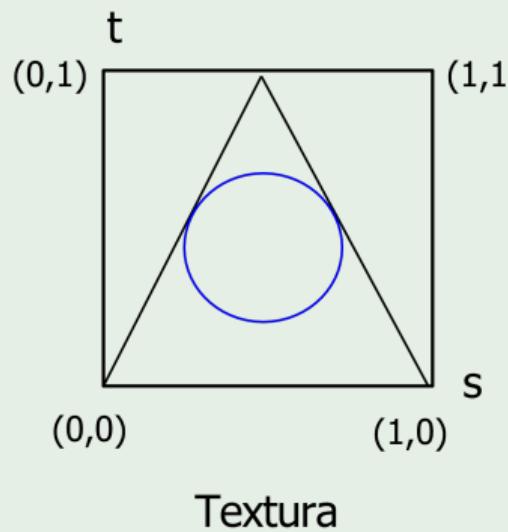
## Ejemplo: Parte de una textura en un triángulo

```
Mesh* Mesh::generaTrianguloTexCor(GLdouble rd) {  
    Mesh *m = generaPoligono(3, rd);...  
  
    m->vTexCoords.reserve(3);  
    m->vTexCoords.emplace_back(0.5, 1);  
    m->vTexCoords.emplace_back(0, 0);  
    m->vTexCoords.emplace_back(1, 0);  
    return m;  
}
```



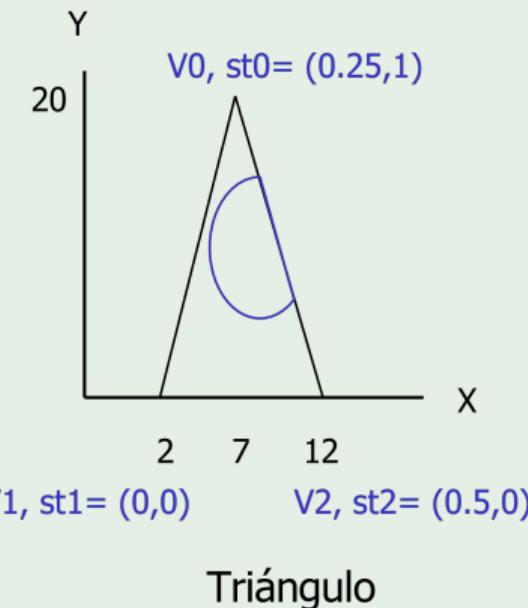
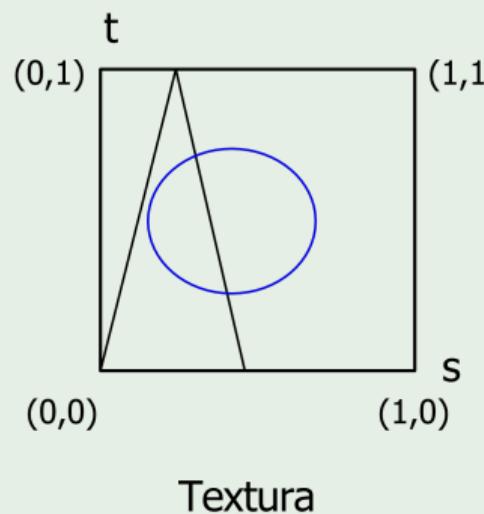
# Aplicación de una textura a una malla

## Ejemplo: Dependiendo de las dimensiones del triángulo



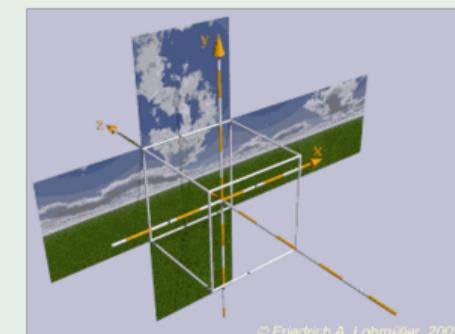
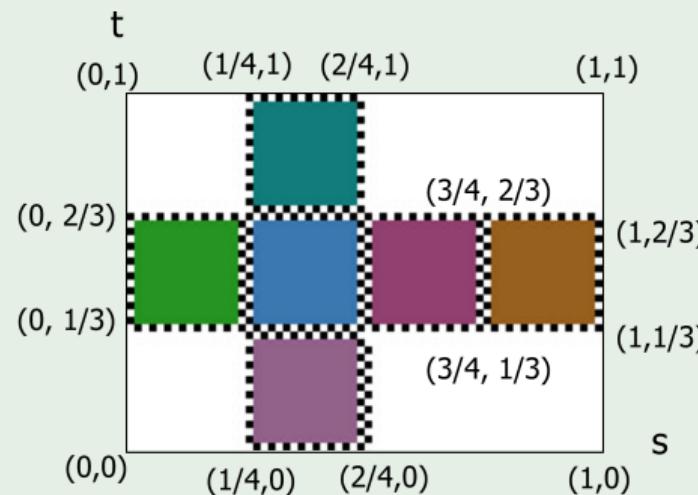
# Aplicación de una textura a una malla

## Ejemplo: Parte de una textura en un triángulo



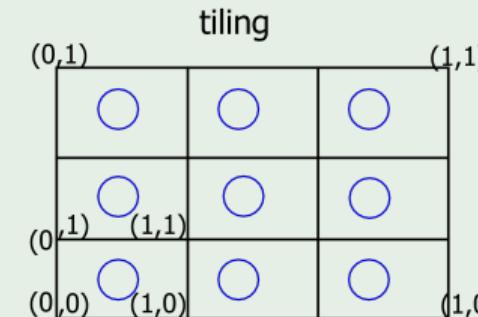
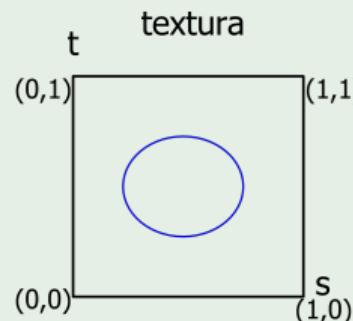
# Aplicación de una textura a una malla

## Ejemplo: Cubo

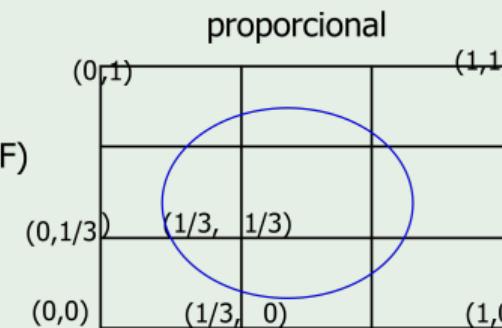


# Aplicación de una textura a una malla

## Ejemplo: Tablero formado por $NDC \times NDF$ rectángulos



$\text{coordText}(i,j) = (i/NDC, j/NDF)$

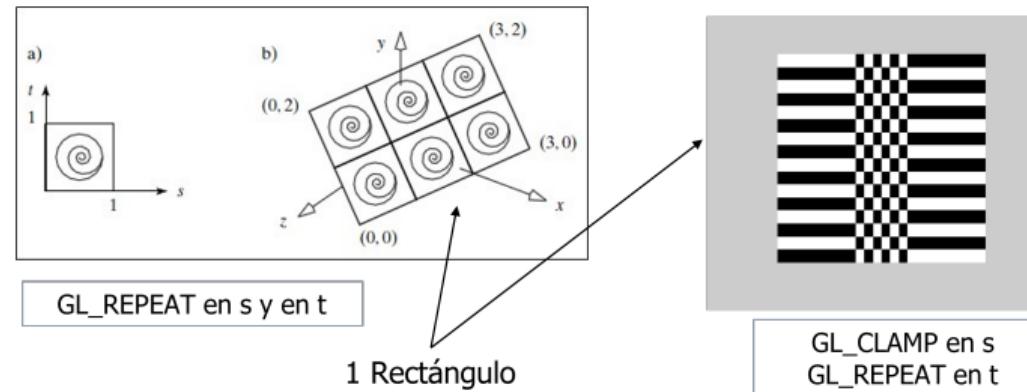


# Aplicación de una textura a una malla

- OpenGL permite asignar coordenadas fuera del intervalo  $[0, 1]$ .

**Texture Wrapping:** ¿Qué se hace cuando las coordenadas de textura caen fuera del rango  $[0, 1]$ ?

- **GL\_REPEAT:** la textura se repite (**tiling**). Se ignora la parte entera de las coordenadas de textura.
- **GL\_CLAMP:** coordenadas de textura superiores a 1 se ajustan a 1, y las coordenadas inferiores a 0 se ajustan a 0.

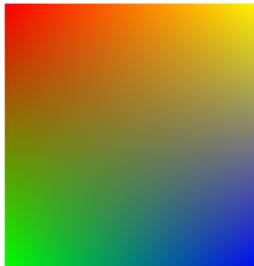


# Mezcla de la textura con el color

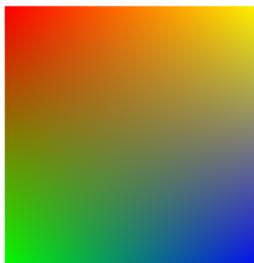
- Cada fragmento, del interior de un triángulo, consta de las coordenadas ( $x, y$ ) del píxel, un color  $C$  y las coordenadas de textura ( $s, t$ ).
- El color  $C$  se **mezcla** con el color de la textura  $T(s, t)$ :  $C = \text{mix}(C, T(s, t))$ .
- Las formas más habituales de combinar estos colores son:
  - **GL\_REPLACE**: Utiliza exclusivamente la textura:  $C = T(s, t)$ .
  - **GL\_MODULATE**: Modula ambos colores:  $C = C \times T(s, t)$ .
  - **GL\_ADD**: Suma ambos colores:  $C = C + T(s, t)$ .
- El color resultante se escribirá en el **Color Buffer**.

# Mezcla de la textura con el color

- **GL\_REPLACE**: Utiliza exclusivamente la textura:  $C = T(s, t)$ .

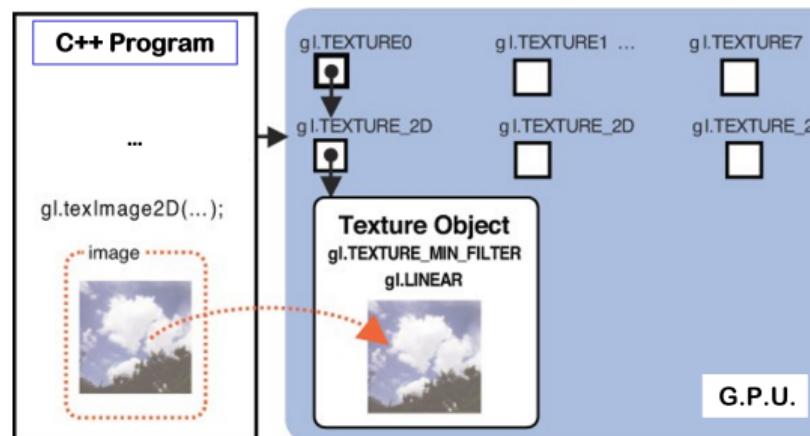

$$\begin{array}{c} + \\ \begin{matrix} & & \\ 0,1 & & 1,1 \\ & \text{TEXTURA} & \\ & & 0,0 & 1,0 \end{matrix} \\ = \end{array} \begin{matrix} & & \\ 0,1 & & 1,1 \\ & \text{TEXTURA} & \\ & & 0,0 & 1,0 \end{matrix}$$

- **GL\_MODULATE**: Modula ambos colores:  $C = C \times T(s, t)$ .


$$\begin{array}{c} + \\ \begin{matrix} & & \\ 0,1 & & 1,1 \\ & \text{TEXTURA} & \\ & & 0,0 & 1,0 \end{matrix} \\ = \end{array} \begin{matrix} & & \\ 0,1 & & 1,1 \\ & \text{TEXTURA} & \\ & & 0,0 & 1,0 \end{matrix}$$

# Texturas 2D en OpenGL

- En OpenGL las texturas se gestionan mediante **objetos de textura**: estructuras GPU que contienen la imagen y la configuración de la textura (filtros y wrapping, pero no el modo de mezcla).



- Hay que activar y desactivar el uso de texturas con:
  - `glEnable(GL_TEXTURE_2D); //en scene::setGL`
  - `glDisable(GL_TEXTURE_2D); //en resetGL`

# Texturas 2D en OpenGL

- Gestión de objetos de texturas:

- Crearlos y destruirlos: `glGenTextures(...)`, `glDeleteTextures(...)`
- Configurarlos (filtros y wrapping): `glTexParameter*(...)`
- Activarlos para que tengan efecto: `glBindTexture(...)`, `glTexEnvi(...)`
- Transferir la imagen (**de CPU a GPU**):

```
glTexImage2D (
    GL_TEXTURE_2D, // 1D ó 3D
    0, // mipmap level
    GL_RGBA, // Formato interno (GPU) de los datos de la textura
    width, height, // Potencias de 2?
    0, // -> border
    GL_RGBA, // Formato de los datos de la imagen (data)
    GL_UNSIGNED_BYTE, // Tipo de datos de los datos de data
    data // puntero a la variable CPU con la imagen
)
```

# Escena con texturas

- Añadimos una nueva clase, `Texture`, con métodos para cargar de archivo una imagen, transferirla a la GPU (`load`) y para activar (`bind`) y desactivar (`unbind`) la textura en la GPU cuando la queramos usar.
- Añadimos a la clase `Scene` un atributo para las texturas:

```
vector<Texture*> gTextures;
```

- La entidad necesita una malla con coordenadas de textura y la textura que queremos usar. Añadimos a la clase `Abs_Entity` un atributo para la textura (`Texture* mTexture`), que habrá que establecer al crear la entidad (método `setTexture`), y activar/desactivar al renderizarla.
- Generamos mallas con coordenadas de textura.

# Implementación

```
class Texture // utiliza la clase PixMap32RGBA para el método load
{
public:
    Texture() = default;
    ~Texture() {if (mId !=0 ) glDeleteTextures(1, &mId); }

    // cargar y transferir a GPU
    void load(const std::string & BMP_Name, GLubyte alpha = 255);
    void bind(GLuint mixMode); // mixMode: GL_REPLACE | MODULATE | ADD
    void unbind() { glBindTexture(GL_TEXTURE_2D, 0); }

protected:
    void init();
    GLuint mWidth =0, mHeight =0; // dimensiones de la imagen
    GLuint mId=0; // identificador interno (GPU) de la textura
                    // 0 significa NULL, no es un identificador válido
};
```

# Implementación

```
void Texture::init() {
    // genera un identificador para una nueva textura
    glGenTextures(1, &mId);

    glBindTexture(GL_TEXTURE_2D, mId); // filters and wrapping
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}
```

# Implementación

```
void Texture::bind(GLuint mixMode) {
    // mixMode: modo para la mezcla los colores

    glBindTexture(GL_TEXTURE_2D, mId); // activa la textura

    // el modo de mezcla de colores no queda
    // guardado en el objeto de textura
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mixMode);
        // mixMode: GL_REPLACE, GL_MODULATE, GL_ADD ...
}
```

# Implementación

```
void Texture::load(const std::string& BMP_Name, GLubyte alpha) {
    if (mId == 0) init();

    PixMap32RGBA pixMap; // var. local para cargar la imagen del archivo
    pixMap.load_bmp24BGR(BMP_Name); // carga y añade alpha=255
    // carga correcta ? -> exception
    if (alpha != 255) pixMap.set_alpha(alpha);

    mWidth = pixMap.width();
    mHeight = pixMap.height();

    glBindTexture(GL_TEXTURE_2D, mId);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, mWidth, mHeight, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, pixMap.data()); // transferir a GPU
    glBindTexture(GL_TEXTURE_2D, 0); // la textura queda desactivada ?
}
```

# Entidad con textura

- La entidad necesita una malla con coordenadas de textura y la textura que queremos usar:

```
void Ground::Ground(...) {  
    mesh = Mesh::generateRectangleTexCor(...);  
    // rectángulo con coordenadas de textura  
    ...  
}
```

- Añadimos a la clase `Abs_Entity` un atributo para la textura:

```
Texture* mTexture = nullptr;
```

- Y un método para establecerla:

```
void setTexture(Texture* tex) { mTexture = tex; };
```

- En el método render hay que activar (y desactivar) la textura antes (después) de renderizar la malla.

# Escenas con texturas

- Añadimos a la clase Scene un atributo para las texturas:

```
vector<Texture*> gTextures;
```

- En `init` creamos y cargamos (con el método `load()`) las texturas de los objetos de la escena. Y en `free` las liberamos.
- Al crear los objetos establecemos sus texturas.
- Adaptamos los métodos `setGL` y `resetGL` para activar/desactivar las texturas en OpenGL.
- Para activar las texturas (en `setGL`): `glEnable(GL_TEXTURE_2D);`
- Para desactivarlas (en `resetGL`): `glDisable(GL_TEXTURE_2D);`

# Guardar la escena como una textura

- Copiar en la textura activa parte de la imagen del Color Buffer:

```
glCopyTexImage2D(GL_TEXTURE_2D, level(0), internalFormat,  
xLeft, yBottom, width, height, border(0));  
// en coordenadas de pantalla (como el puerto de vista)
```

- Los datos se copian del buffer de lectura activo: `GL_FRONT` o `GL_BACK`.
- Para modificar el buffer de lectura activo:

```
glReadBuffer(GL_FRONT / GL_BACK); // por defecto GL_BACK
```

- Obtener (de GPU a CPU) la imagen de la textura activa:

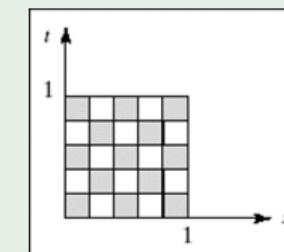
```
glGetTexImage(GL_TEXTURE_2D, level(0), imgFormat, imgType, pixels);  
// pixels-> array donde guardar los datos (de tipo y tamaño adecuado)
```

# Definición de texturas

- También podemos definir de forma procedimental matrices.

## Ejemplo: para colores de 4 componentes Glubyte

```
void TexturaProc (GLubyte mat[NC][NF][4]) {  
    for (int i = 0; i < NC; i++)  
        for (int j = 0; j < NF; j++) {  
            int c = ((i + j) % 2) * 255;  
            mat[i][j][0] = Glubyte(c);  
            mat[i][j][1] = Glubyte(c);  
            mat[i][j][2] = Glubyte(c);  
            mat[i][j][3] = Glubyte(255);  
        }  
}
```



# Referencia de la API de texturas

# Gestión de objetos de textura

- **Generar nombres para los objetos de textura:**

```
GLuint Name; GLuint Names[3];  
glGenTextures(1, &Name);  
glGenTextures(3, Names);
```

- Los nombres que se generan se devuelven en el segundo parámetro. **No son consecutivos.**
- El 0 nunca se devuelve como nombre, pero podemos utilizarlo para que no esté activo ningún objeto de textura.

# Activación y desactivación de objetos de textura

- **Crear objetos de textura y activarlos:**

El comando para crear un objeto de textura es el mismo que para activarlo. Si se activa un objeto que no existe, se crea y queda activo. Los demás comandos sobre texturas se ejecutan sobre el objeto activo.

```
glBindTexture(GL_TEXTURE_2D, Name);  
glBindTexture(GL_TEXTURE_2D, Names[i]);
```

- Para desactivar la textura activa:

```
glBindTexture(GL_TEXTURE_2D, 0);
```

# Liberación de objetos de textura

- **Liberar objetos de texturas:**

```
glDeleteTextures(1, &Name);  
glDeleteTextures(3, Names);
```

# Carga de los píxeles de una textura

- Pasar la imagen a la textura activa:

```
glTexImage2D(GL_TEXTURE_2D, Level, GL_RGB[A],  
             Ncols, NFils, Border, GL_RGB[A], GL_UNSIGNED_BYTE, Data);
```

- Level: El nivel de detalle (multirresolución).  
Para un único nivel debemos poner 0.
- Border: Es un booleano (0 ó 1) que indica si la imagen tiene borde.
- Ncols, NFils: Tamaño de la imagen (Data).
- Data: El array con la imagen que se quiere usar como textura.  
El formato del array debe ser el especificado y a continuación  
puede liberarse. Por ejemplo: `GLubyte data[Ncols * NFils * 3];`

# Ampliación o reducción de una textura

- **Configurar los filtros para el objeto de textura activo**

- Debemos especificar que proceso seguir en caso de tener que **aumentar o reducir la imagen** durante su aplicación.
- Debe evitarse que sea necesario aumentar (en una dirección) y disminuir (en la otra) simultáneamente en la aplicación de la textura.

```
glTexParameterI(GL_TEXTURE_2D, TipoProceso, Proceso);
```

# Ampliación o reducción de una textura

```
glTexParameter(GL_TEXTURE_2D, TipoProceso, Proceso);
```

- Para **aumentar**:

TipoProceso: GL\_TEXTURE\_MAG\_FILTER

Proceso: GL\_NEAREST (el más cercano) |

GL\_LINEAR (valor por defecto, una combinación de los 4 más cercanos)

- Para **reducir**:

TipoProceso: GL\_TEXTURE\_MIN\_FILTER

Proceso: GL\_NEAREST | GL\_LINEAR | GL\_NEAREST\_MIPMAP\_LINEAR

## CUIDADO!!!

GL\_NEAREST\_MIPMAP\_LINEAR es el valor por defecto, pero sólo funciona en el caso de multirresolución.

## Ejemplo

```
glTexParameter(GL_TEXTURE_2D,  
               GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

# Combinación de color e imagen

- Configurar la combinación con el color:

```
glTexEnvf(GL_TEXTURE_ENV,  
          GL_TEXTURE_ENV_MODE, FunMix);
```

## CUIDADO!!!

Hay que establecer la mezcla cada vez que se usa la textura.

FunMix: Puede tomar los valores (entre otros):

`GL_DECAL` ( $C = (1 - At).Cf + At.Ct$  y  $A = Af$ )

`GL_REPLACE` ( $C = Ct$  y  $A = At$ )

`GL_MODULATE` (defecto,  $C = Ct.Cf$  y  $A = At.Af$ )

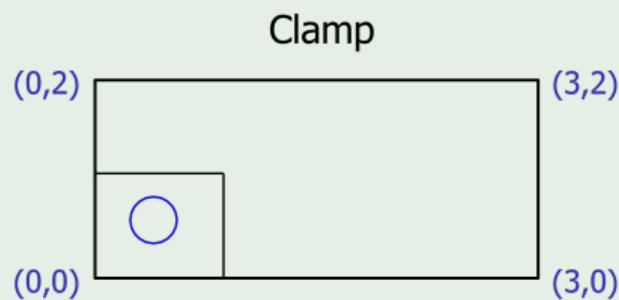
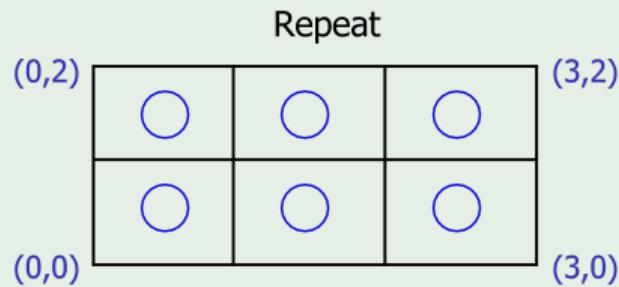
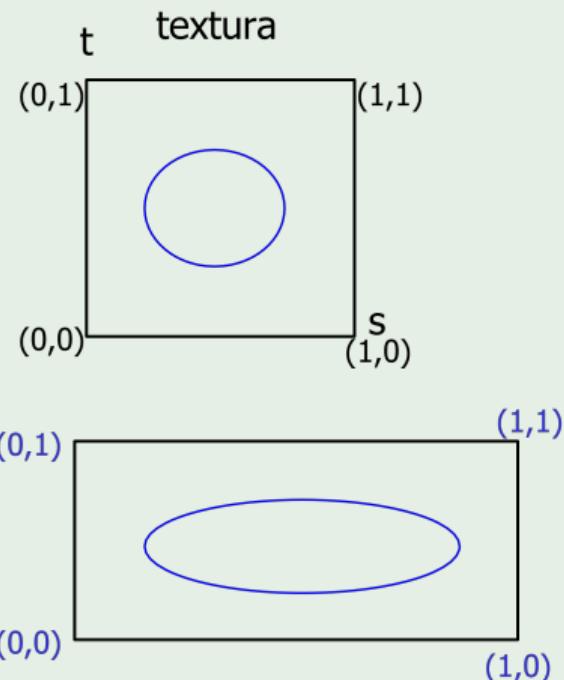
Donde ( $C, A$ ) es el `RGBA` resultado de la combinación, ( $Ct, At$ ) es el `RGBA` de la textura y ( $Cf, Af$ ) es el `RGBA` del `color del fragmento`.

# Wrapping

- Una textura 2D normalizada es una función de dos parámetros:  
 $T(s, t) : [0, 1] \times [0, 1] \rightarrow \text{Colores}$
- En caso de utilizar coordenadas de textura  $(s, t)$  fuera de los intervalos  $[0, 1]$ , podemos indicar como transformarlas a  $[0, 1]$ .
- Las formas más simples de pasar (**wrap**)  $\mathbb{R} \rightarrow [0, 1]$  son:
  - Quedarse con la parte decimal del valor dado, generando una repetición de la imagen (**REPEAT**).
  - Llevar los valores mayores que 1 a 1 y los menores que 0 a 0 (**CLAMP**).

# Wrapping

## Ejemplo



# Wrapping

```
glTexParameterI(GL_TEXTURE_2D, TipoProceso, Proceso);
```

- En caso de utilizar coordenadas de textura  $(s, t)$  fuera de los intervalos  $[0, 1]$ , podemos indicar como transformarlas a  $[0, 1]$ . Se especifica independientemente para cada una de las coordenadas  $s$  y  $t$ .

TipoProceso:    `GL_TEXTURE_WRAP_S` (para la coordenada s)  
                    `GL_TEXTURE_WRAP_T` (para la coordenada t)

Proceso :        `GL_CLAMP` | `GL_REPEAT`

- El valor por defecto, en ambos casos, es `GL_REPEAT`, que se queda con la parte decimal del valor dado, generando una repetición de la imagen (*tiling*).
- `GL_CLAMP` lleva los valores mayores que 1 a 1 y los menores que 0 a 0.

# Lectura de los píxeles de la imagen o de una textura

- **Copiar en la textura activa parte de la imagen del Color Buffer:**

```
glCopyTexImage2D(GL_TEXTURE_2D, level, internalFormat,  
    xleft, ybottom, w, h, border);  
    // en coordenadas de pantalla (como el puerto de vista)
```

- Los datos se copian del buffer de lectura activo: GL\_FRONT o GL\_BACK.
- Para modificar el buffer de lectura activo:

```
glReadBuffer(GL_FRONT | GL_BACK); //por defecto GL_BACK.
```

- **Obtener la imagen de la textura activa:**

```
glGetTexImage(GL_TEXTURE_2D, level, format, type, pixels);  
// pixels-> array donde guardar los datos (de tipo y tamaño adecuado)
```