Texturas en OpenGL

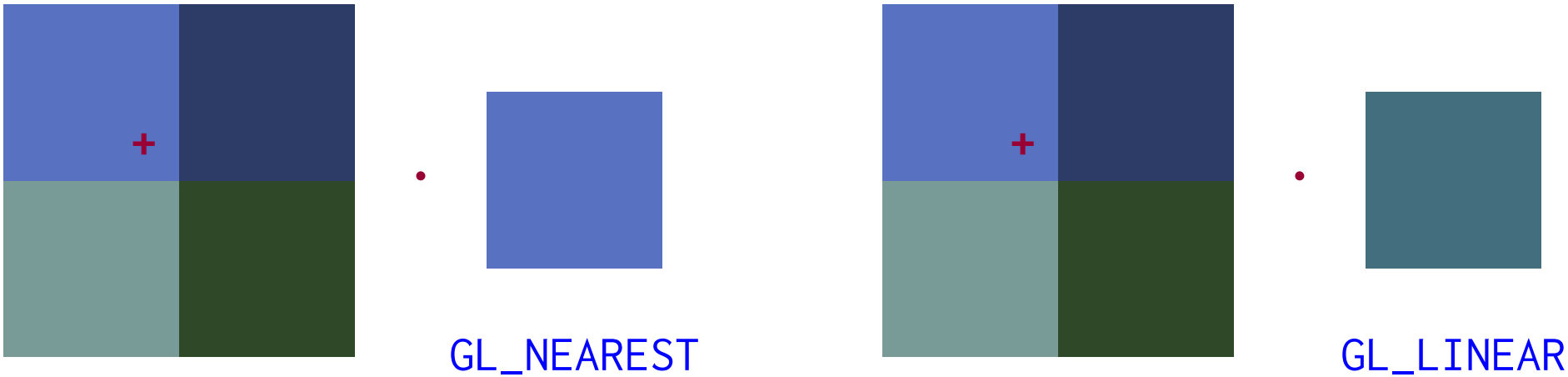
Informática Gráfica I

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

Contenido

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.



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
```

Aplicación de una textura a una malla

- El método `Mesh::load()` tiene que cargar en la GPU el array de coordenadas de textura:

```
glGenBuffers(1, &mTCO);
glBindBuffer(GL_ARRAY_BUFFER, mTCO);
glBufferData(GL_ARRAY_BUFFER,
             vTexCoords.size() * sizeof(vec2),
             vTexCoords.data(), GL_STATIC_DRAW);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
             sizeof(vec2), nullptr);
glEnableVertexAttribArray(2);
```

- El método `Mesh::unload()` tiene que eliminarlo de la GPU:

```
if (mTCO != NONE) glDeleteBuffers(1, &mTCO);
```

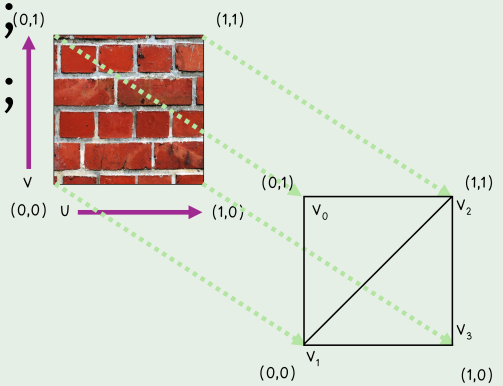
Aplicación de una textura a una malla

- 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 una nueva clase `EntityWithTexture` que extiende `Abs_Entity` con un atributo para la textura (`Texture* mTexture`), que habrá que establecer al crear la entidad (o con el método `setTexture`), y activar/desactivar al renderizarla. Además, selecciona un *shader* compatible como *texture*.

Aplicación de una textura a una malla

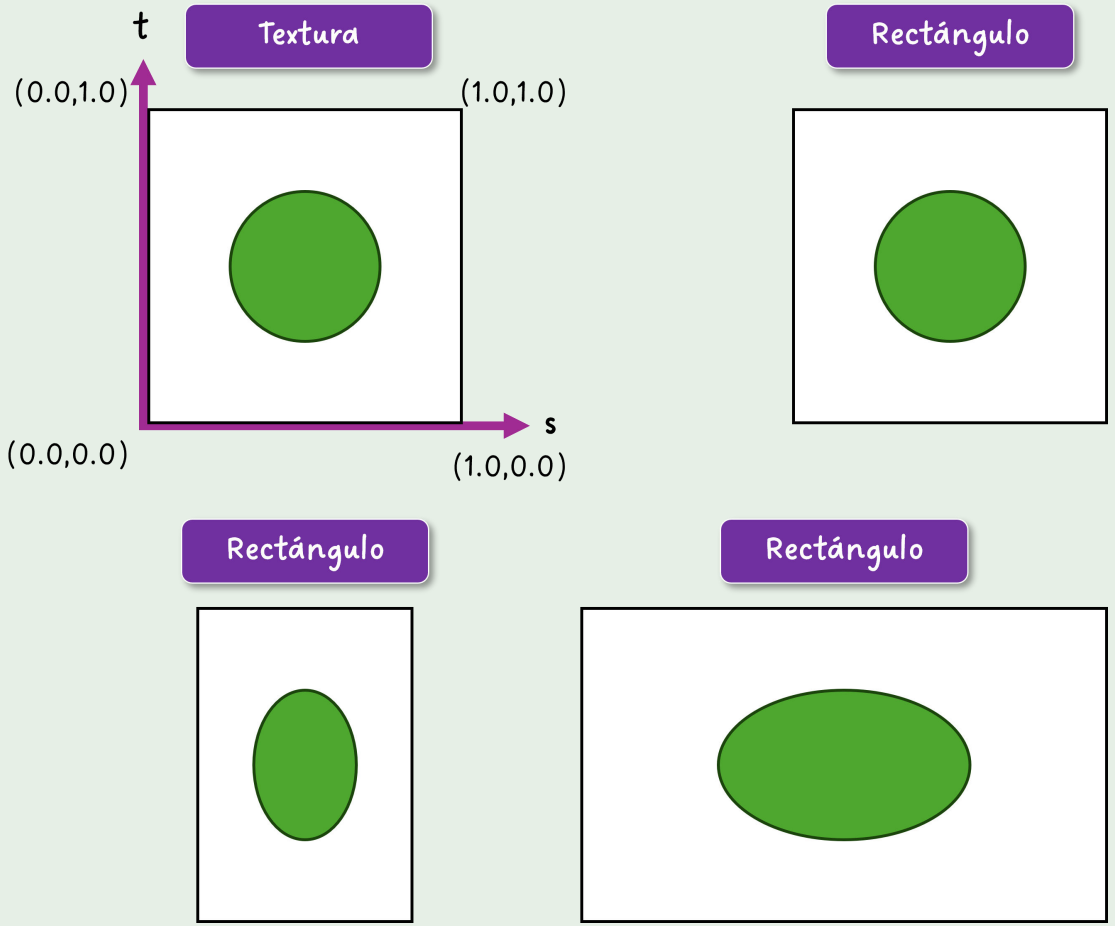
Toda la textura en un triángulo

```
Mesh* Mesh::generateRectangleTexCor(GLdouble w, GLdouble h) {
    Mesh *m = generateRectangle(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

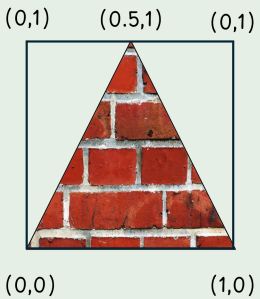
Dependiendo de las dimensiones del rectángulo

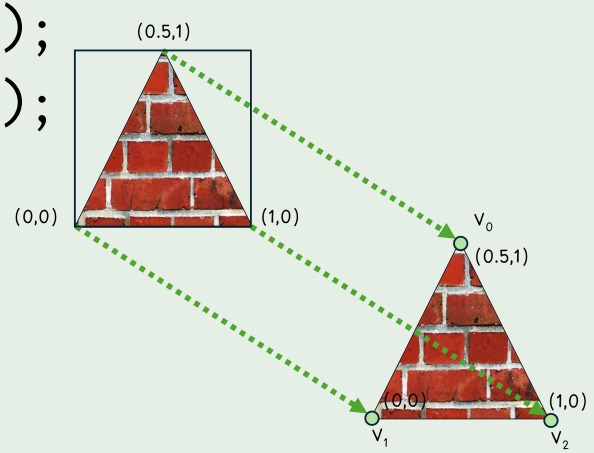


Aplicación de una textura a una malla

Parte de una textura en un triángulo

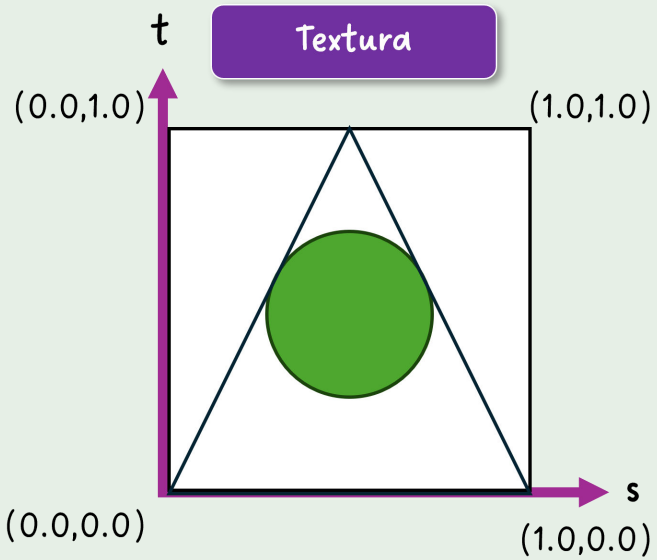
```
Mesh* Mesh::generaTrianguloTexCor(GLdouble rd) {
    Mesh *m = generaPoligono(3, rd);…

    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;
}
```

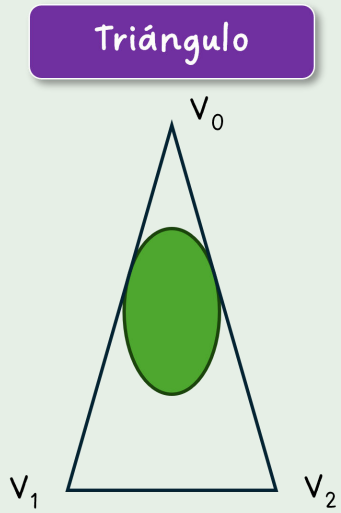




Aplicación de una textura a una malla

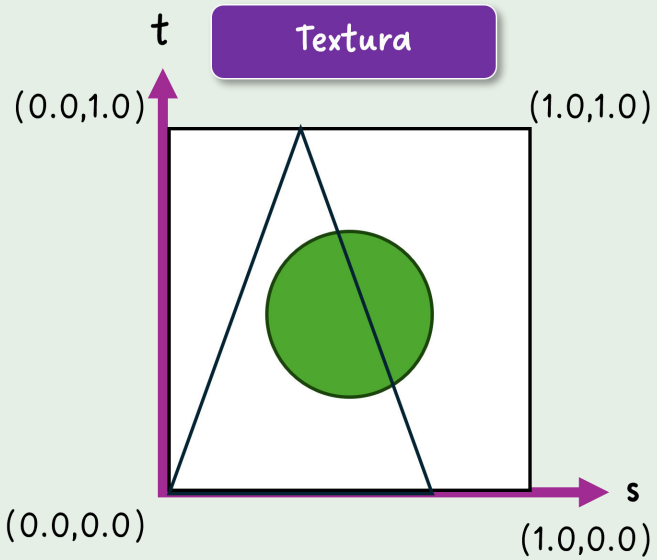
Dependiendo de las dimensiones del triángulo

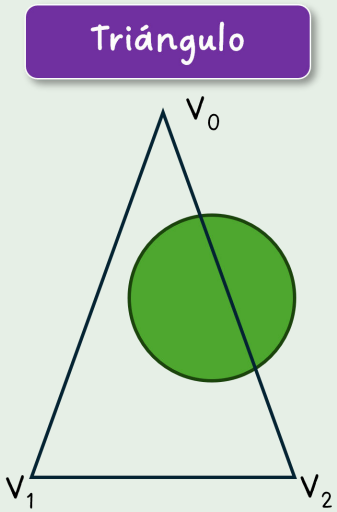




Aplicación de una textura a una malla

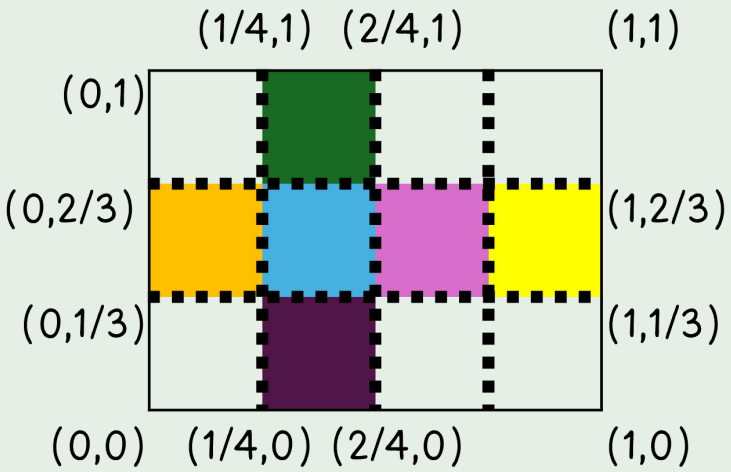
Parte de una textura en un triángulo

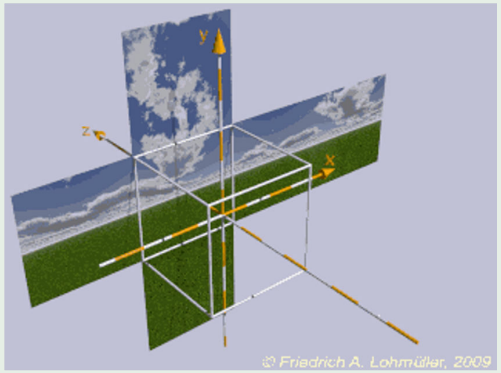




Aplicación de una textura a una malla

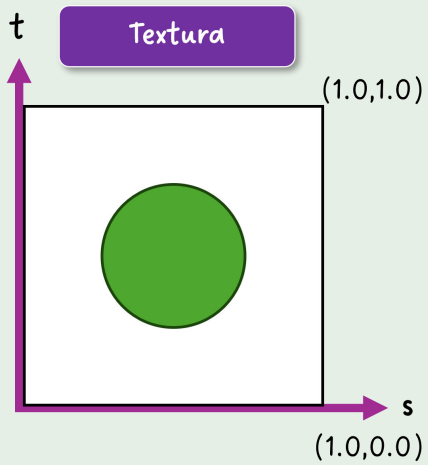
Cubo

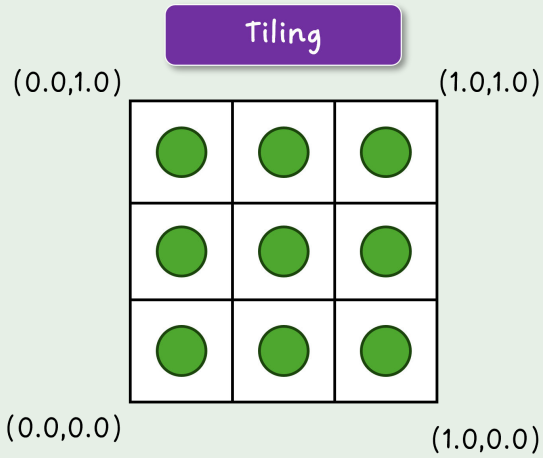




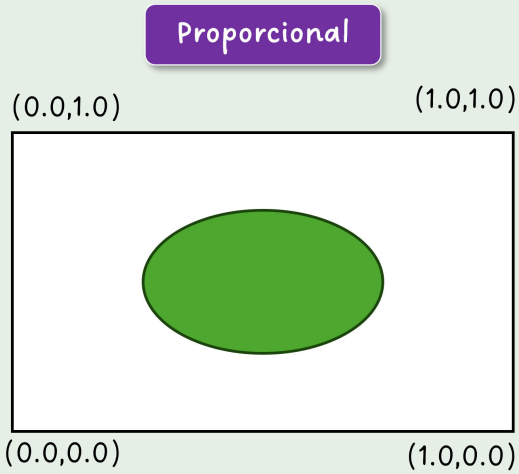
Aplicación de una textura a una malla

Tablero formado por $NDC \times NDF$ rectángulos



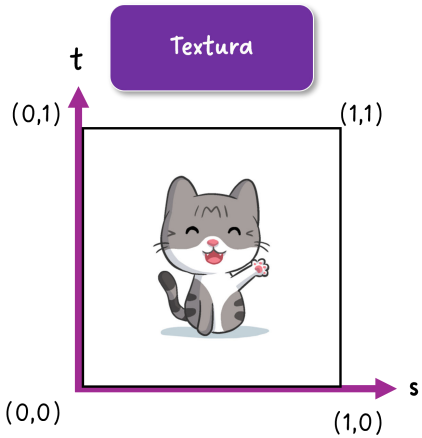


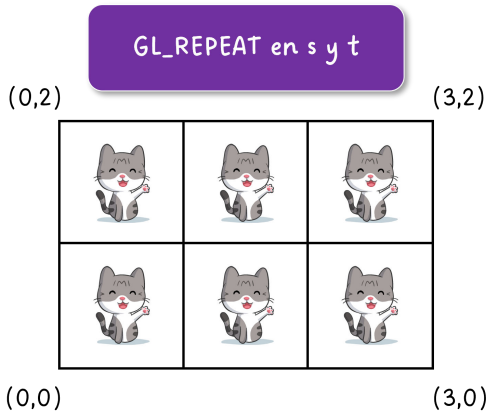
$$coorTextura(i,j) = (\frac{i}{NDC}, \frac{j}{NDF})$$

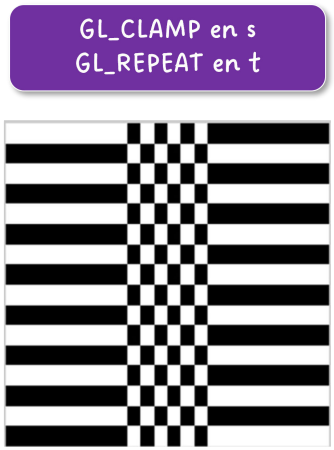


Aplicación de una textura a una malla

- OpenGL permite asignar coordenadas fuera del intervalo $[0, 1]$.
  Texture Wrapping:
  - `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.







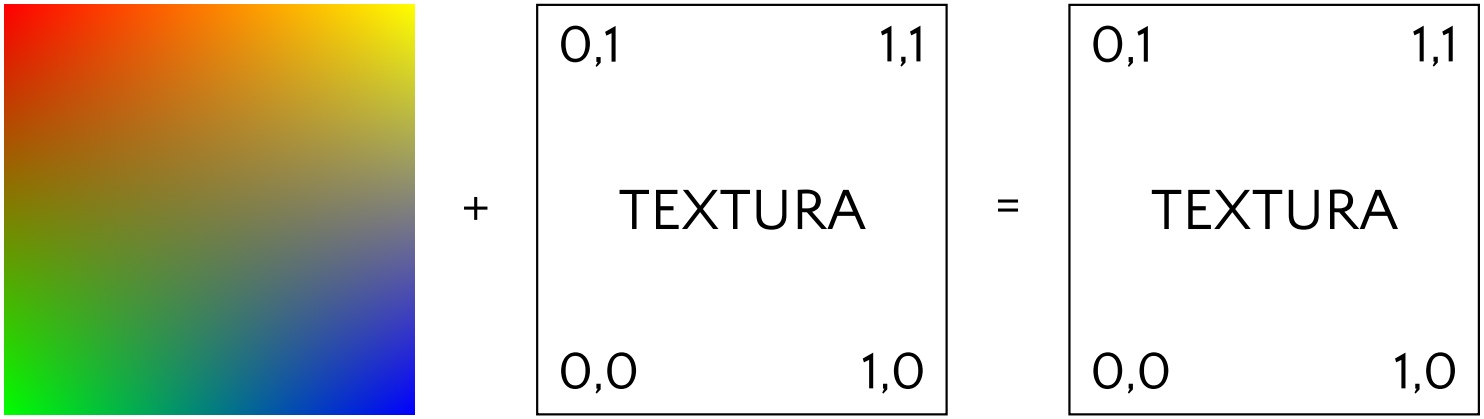
Mezcla de la textura con el color

- 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: $C = C + T(s,t)$.

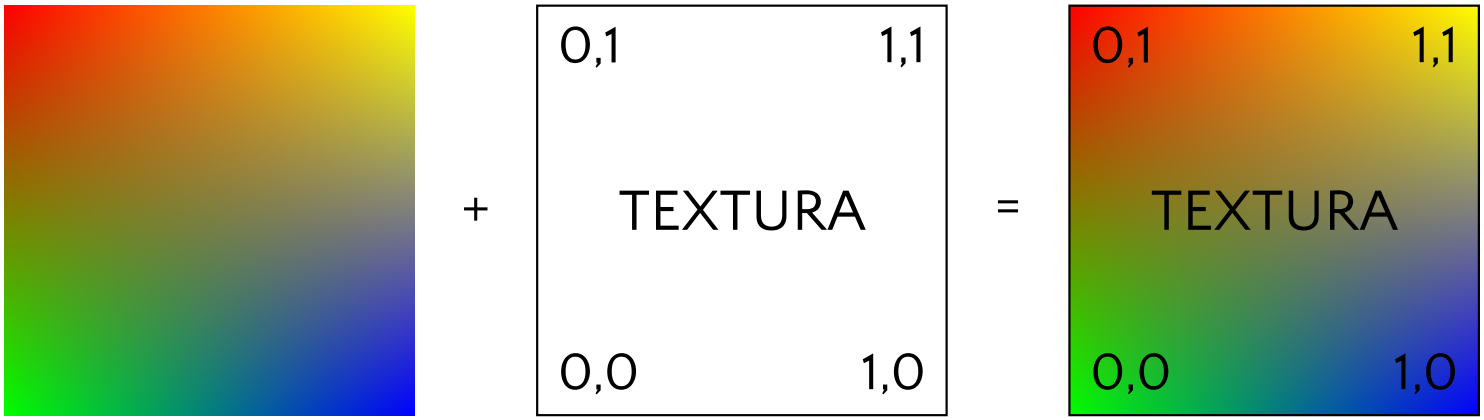
  Los nombres como `GL_REPLACE` se utilizaban en el perfil de compatibilidad, pero ahora es algo que ha de manejar el *shader* de fragmentos.
- El color resultante se escribirá en el `Color Buffer`.

Mezcla de la textura con el color

- GL_REPLACE: Utiliza exclusivamente la textura: $C = T(s,t)$.

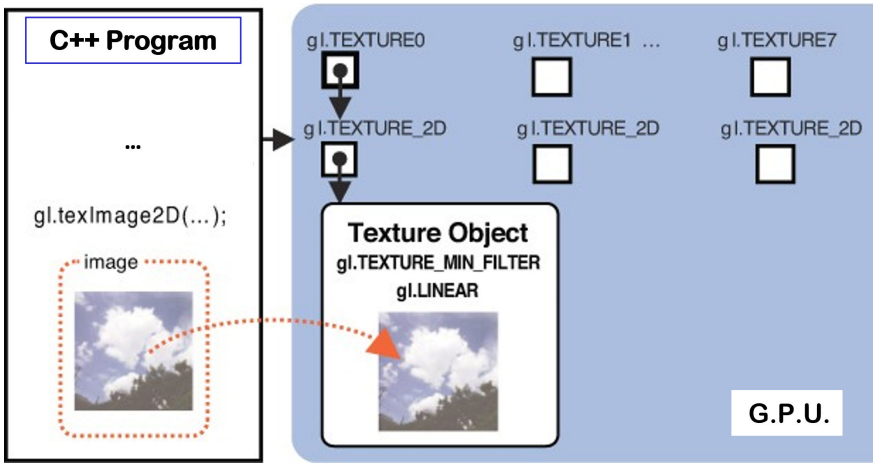


- GL_MODULATE: Modula ambos colores: $C = C \times T(s,t)$.



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

1. Crearlos y destruirlos: `glGenTextures(...)` y `glDeleteTextures(...)`
2. Configurarlos (filtros y wrapping): `glTexParameter*(...)`
3. Activarlos para que tengan efecto: `glBindTexture(...)`, `glTexEnvi(...)`

Texturas 2D en OpenGL

Gestión de objetos de texturas

4. 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);
  // mixMode: GL_REPLACE | MODULATE | ADD
  void bind(GLuint mixMode);
  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();

  // variable para cargar la imagen del archivo
  PixMap32RGBA pixMap;
  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);
  // transferir a GPU
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, mWidth, mHeight, 0,
      GL_RGBA, GL_UNSIGNED_BYTE, pixMap.data());
  // la textura queda desactivada ?
  glBindTexture(GL_TEXTURE_2D, 0);
}
```

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:

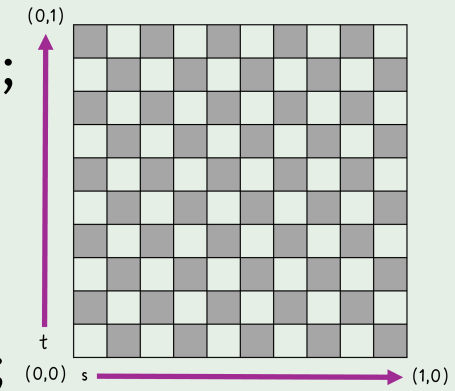
```
glGetTexImage(GL_TEXTURE_2D, level(0),
  imgFormat, imgType, pixels);
// pixels: array donde guardar los
// datos (de tipo y tamaño adecuado)
```

Definición de texturas

- También podemos definir de forma procedimental matrices.

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

```
glTexParameteri(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

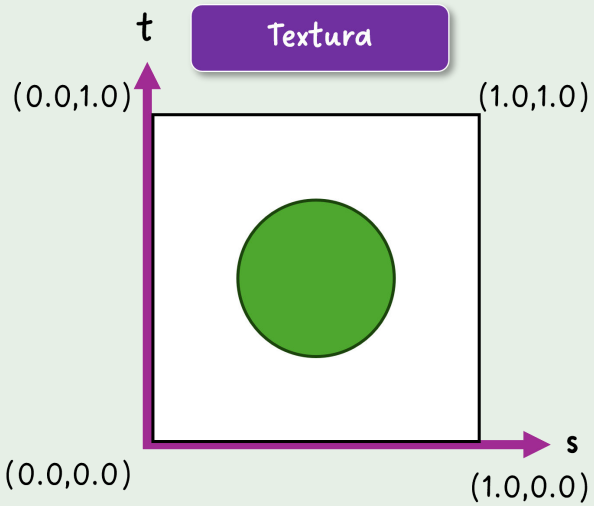
```
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER,
  GL_LINEAR);
```

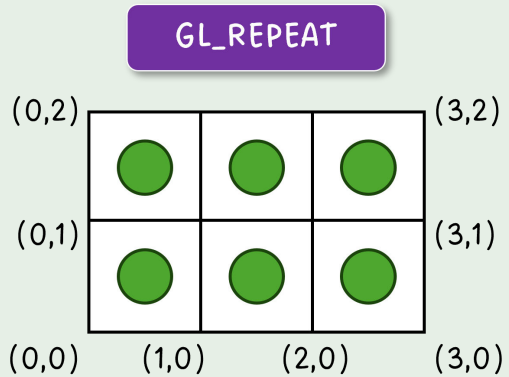
Wrapping

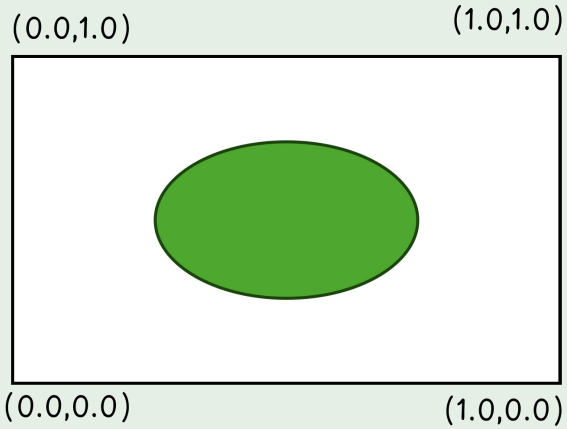
- Una textura 2D normalizada es una función de dos parámetros:
  $T(s,t) : [0,1] \times [0,1] \rightarrow$ Colores
- En caso de utilizar coordenadas de textura (s,t) fuera de los intervalos $[0,1]$, podemos indicar como transformarlas a $[0,1]$.
- Las formas más simples de pasar (**wrap**) $\mathbb{R} \rightarrow [0,1]$ 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***).

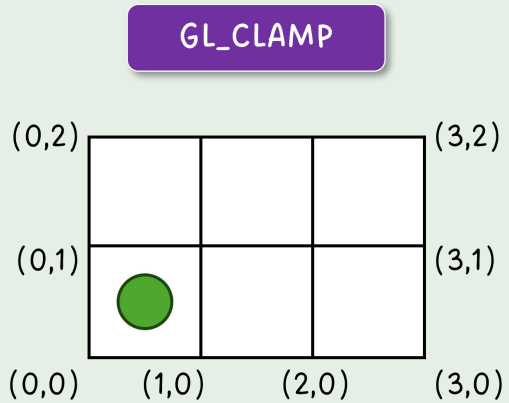
Wrapping

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 tamañ
```

Shader *texture*

El *shader* de vértices solo transmite la coordenada de textura (en la posición 2 del VAO) al *shader* de fragmentos a través de un parámetro `texCoord`:

```
layout (location = 2) in vec2 aTexCoord; // texture coordinates
out vec2 texCoord;
```

que simplemente se asigna

```
void main() {
        gl_Position = projection * modelView * vec4(aPos, 1.0);
        vertexColor = aColor;
        texCoord = aTexCoord;
}
```

Shader *texture*

El *shader* de fragmentos recibe el color y la coordenada de textura del *shader* de vértices, así como la textura y la opción de modulación como uniformes.

```
in vec4 vertexColor; // color assigned to this vertex
in vec2 texCoord;
uniform sampler2D ourTexture; // chosen texture
uniform bool modulate; // whether to modulate with the vertex c
```

Usa la función `texture` de GLSL para consultar el color de la textura en la posición recibida y lo mezcla con el color del vértice si procede.

```
void main()
{
        if (modulate) FragColor = texture(ourTexture, texCoord)
        else          FragColor = texture(ourTexture, texCoord);
}
```