# CAPITULO 2

# OBJETOS GEOMÉTRICOS Y TRANSFORMACIONES

# CAPITULO 3

# PROPIEDADES Y RENDERING

2.2 Transformaciones Geométricas en 2D

3.1 Color, Luz, materiales y textura

# 2.2.3 Textures

# Textures

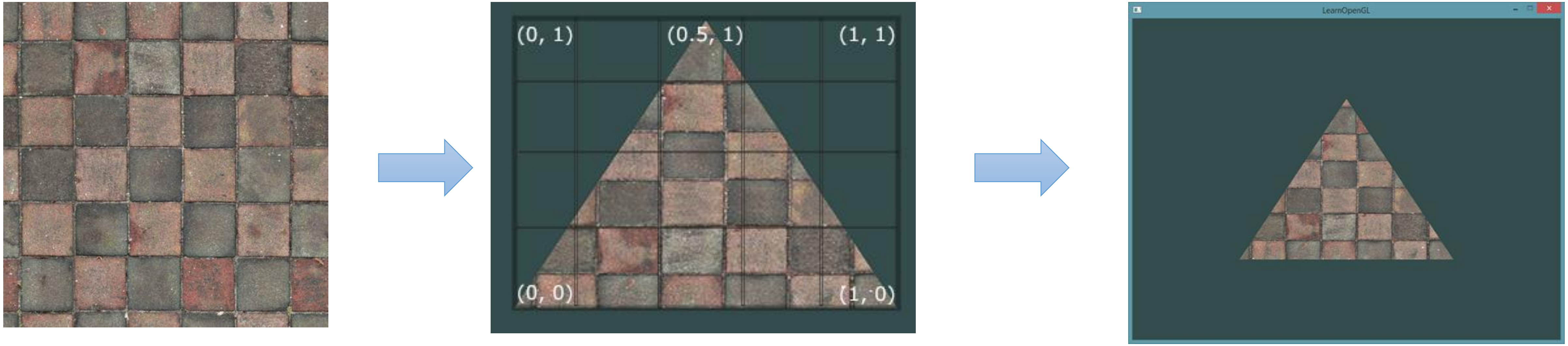
- A texture is a 2D image (even 1D and 3D textures exist) used to add detail to an object.

- Think of a texture as a piece of paper with a nice brick image (for example) on it neatly folded over your 3D house so it looks like your house has a stone exterior.

- Because we can insert a lot of detail in a single image, we can give the illusion the object is extremely detailed **without having to specify extra vertices**.

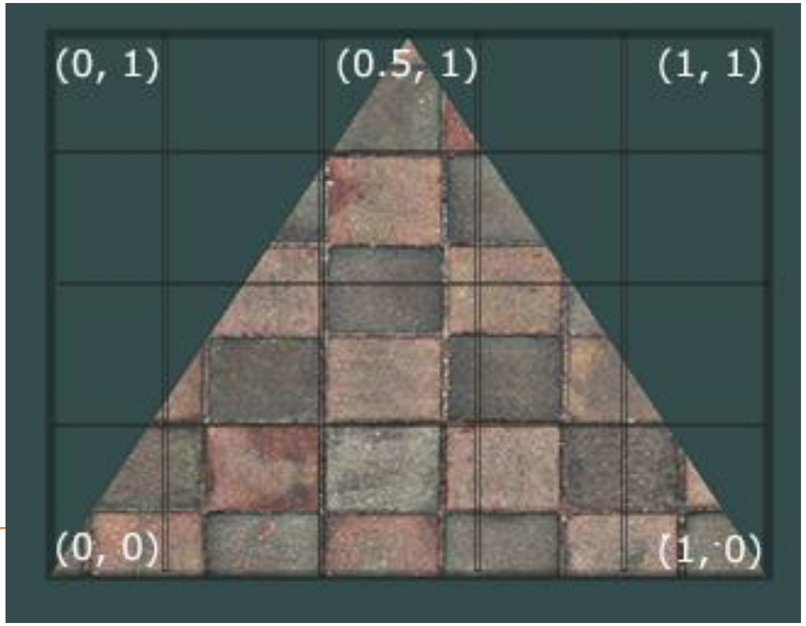
# Textures

- In order to map a texture to the triangle we need to tell each vertex of the triangle which part of the texture it corresponds to.
- Each vertex should thus have a texture coordinate associated with them that specifies what part of the texture image to sample from.
- Fragment interpolation then does the rest for the other fragments.



# Textures - Sampling

- Texture coordinates range from 0 to 1 in the x and y axis (remember that we use 2D texture images).
- Retrieving the texture color using texture coordinates is called **Sampling**.
- Texture coordinates start at **(0,0)** for the lower left corner of a texture image to **(1,1)** for the upper right corner of a texture image.
- The following image shows how we map texture coordinates to the triangle:



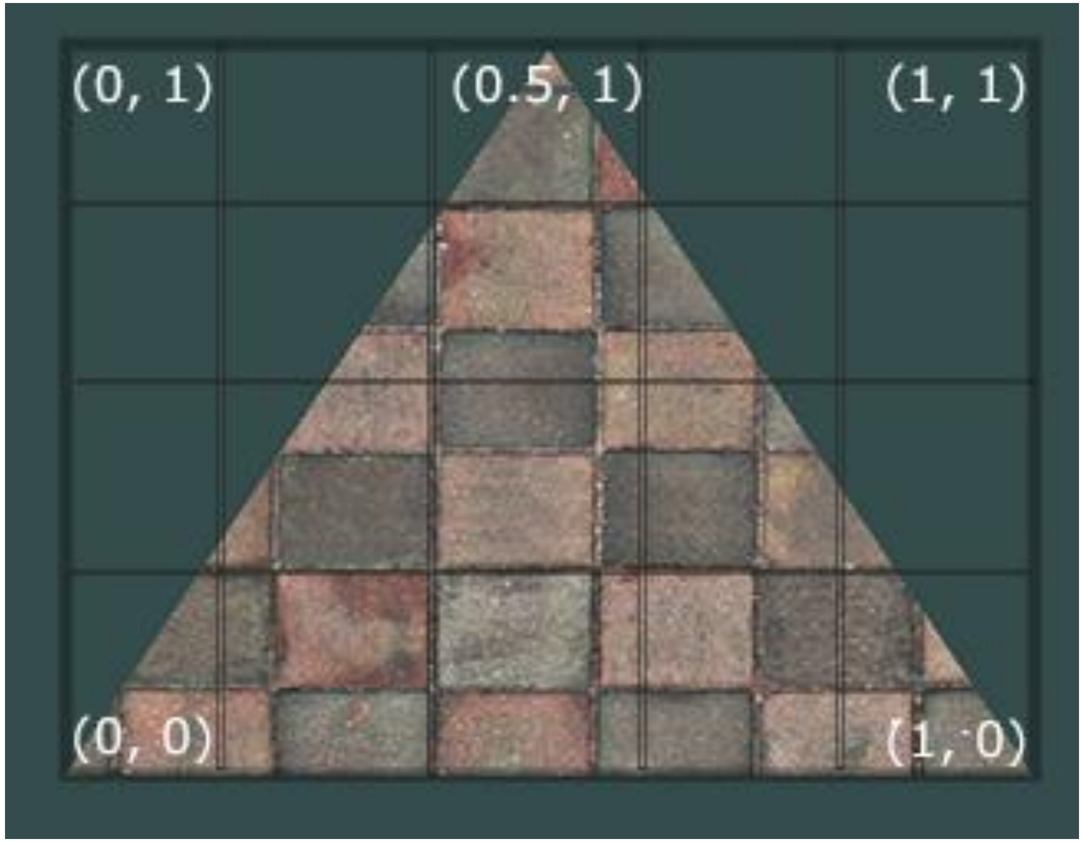
# Textures - Sampling

We specify **3 texture coordinate points** for the triangle.

- We want the bottom-left side of the triangle to correspond with the bottom-left side of the texture so we use the (0,0) texture coordinate for the triangle's bottom-left vertex.

- The same applies to the bottom-right side with a (1,0) texture coordinate.

- The top of the triangle should correspond with the top-center of the texture image so we take (0.5,1.0) as its texture coordinate.

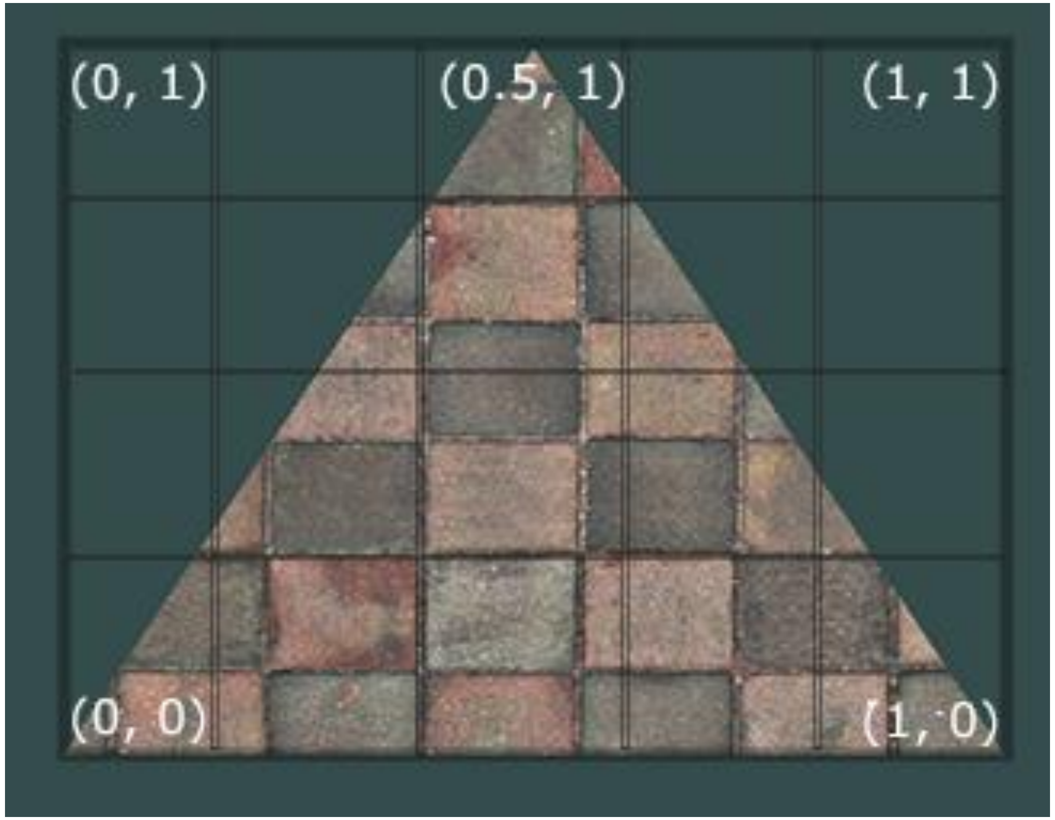



We only have to **pass 3 texture coordinates to the vertex shader**, which then passes those to the **fragment shader** that neatly interpolates all the texture coordinates for each fragment.

# Textures - Sampling

The resulting texture coordinates would then look like this:

```
float texCoords[] = {
   0.0f, 0.0f,  // lower-left corner
   1.0f, 0.0f,  // lower-right corner
   0.5f, 1.0f   // top-center corner
};
```




- Texture sampling has a loose interpretation and can be done in many different ways.

- It is thus our job to tell OpenGL how it should sample its textures.

Texture coordinates usually range from (0,0) to (1,1) but what happens if we specify coordinates outside this range?

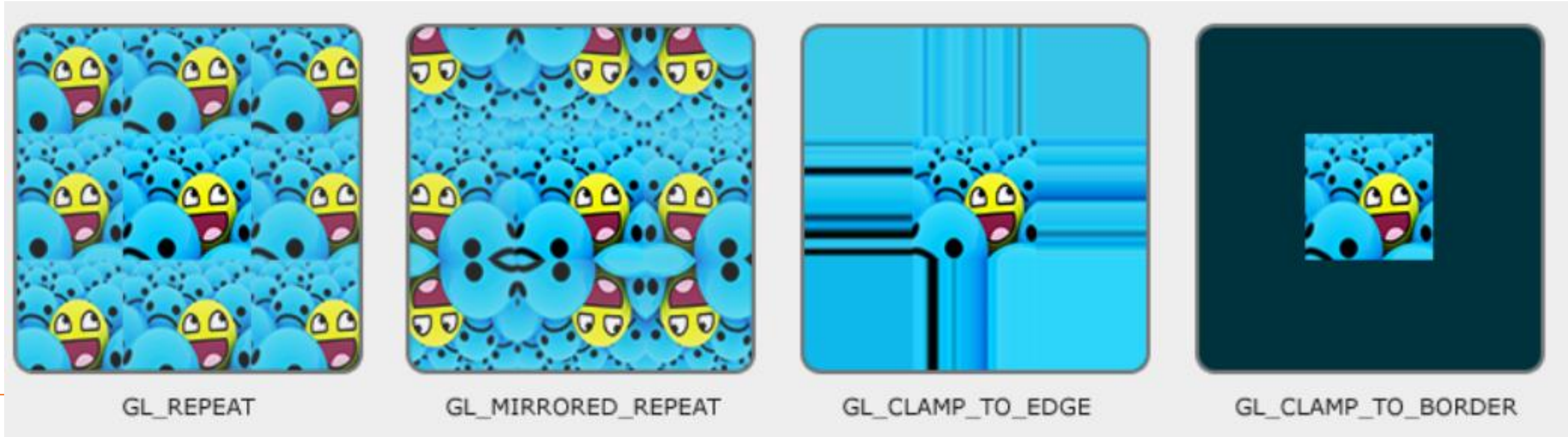
# Textures – Texture Wrapping

Wrapping → embalaje, envoltorio

The default behavior of OpenGL is to repeat the texture images (we basically ignore the integer part of the floating point texture coordinate), but there are more options OpenGL offers:

- **GL_REPEAT**: The default behavior for textures. Repeats the texture image.
- **GL_MIRRORED_REPEAT**: Same as GL_REPEAT but mirrors the image with each repeat.
- **GL_CLAMP_TO_EDGE**: Clamps the coordinates between 0 and 1. The result is that higher coordinates become clamped to the edge, resulting in a stretched edge pattern.
- **GL_CLAMP_TO_BORDER**: Coordinates outside the range are now given a user-specified border color.

Each of the options have a different visual output when using texture coordinates outside the default range.



# Textures – Texture Wrapping

Each of the aforementioned options can be set per coordinate axis (**s, t** (and **r** if you're using 3D textures) equivalent to **x,y,z**) with the glTexParameter* function:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

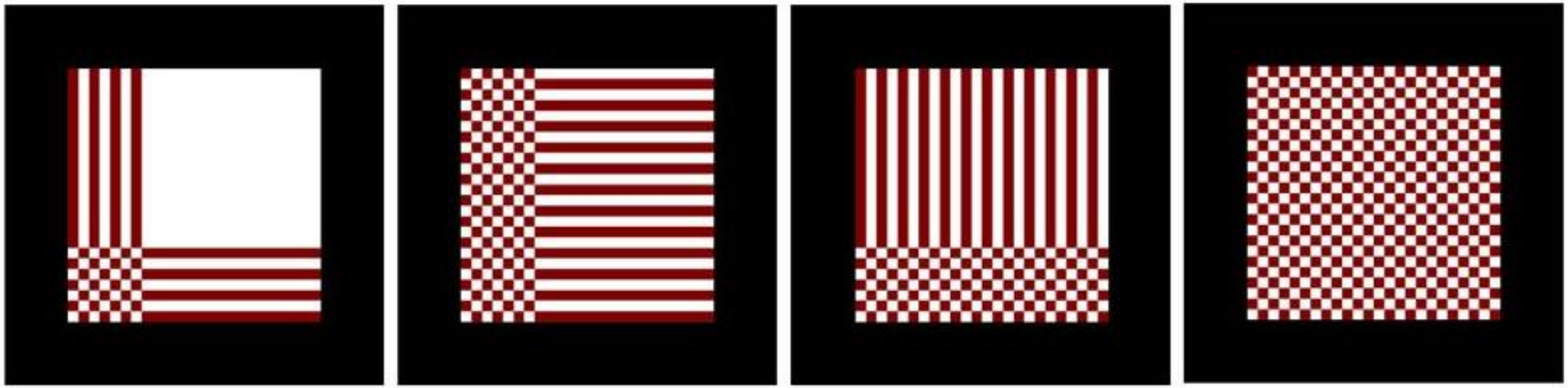
- The **first argument** specifies the **texture target**; we're working with 2D textures so the texture target is **GL_TEXTURE_2D**.
- The **second argument** requires us to tell what option we want to set and for which **texture axis**; we want to configure it for both the **S** and **T** axis.
- The **last argument** requires us to pass in the **texture wrapping mode** we'd like and in this case OpenGL will set its texture wrapping option on the currently active texture with **GL_MIRRORED_REPEAT**.

# Textures – Texture Wrapping

Each of the aforementioned options can be set per coordinate axis (**s, t** (and **r** if you're using 3D textures) equivalent to **x,y,z**) with the glTexParameter* function:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Configuration Examples:

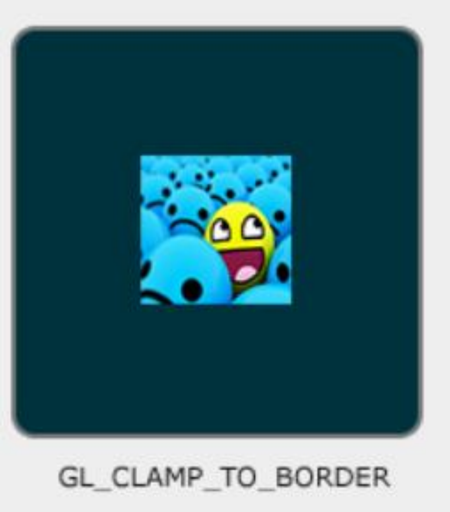


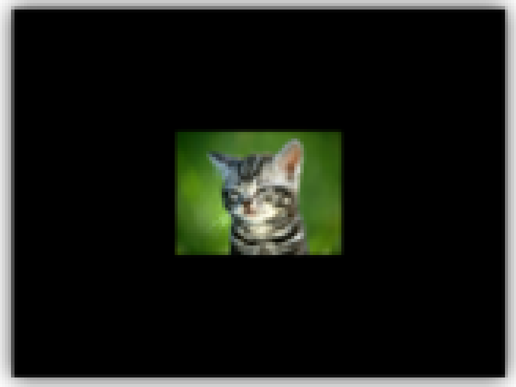
| | | | |
|---|---|---|---|
| Wrap S : GL_CLAMP<br>Wrap T : GL_CLAMP | Wrap S : GL_CLAMP<br>Wrap T : GL_REPEAT | Wrap S : GL_REPEAT<br>Wrap T : GL_CLAMP | Wrap S : GL_REPEAT<br>Wrap T : GL_REPEAT |

# Textures – Texture Wrapping

- If we choose the **GL_CLAMP_TO_BORDER** option we should also specify a border color.

- This is done using the **fv** equivalent of the **glTexParameter** function with **GL_TEXTURE_BORDER_COLOR** as its option where we pass in a float array of the border's color value:

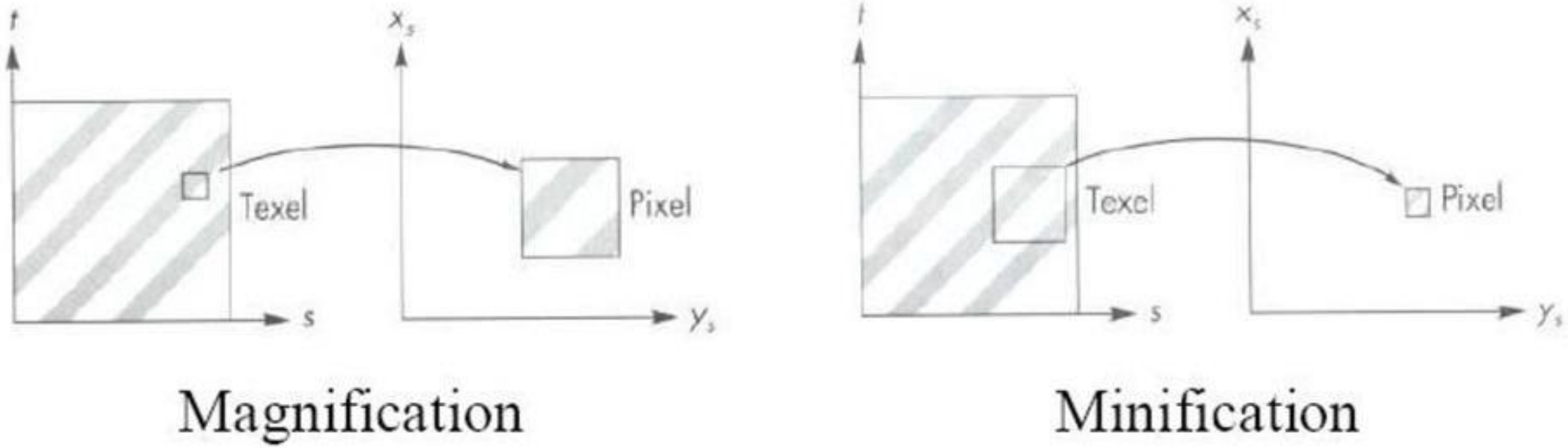


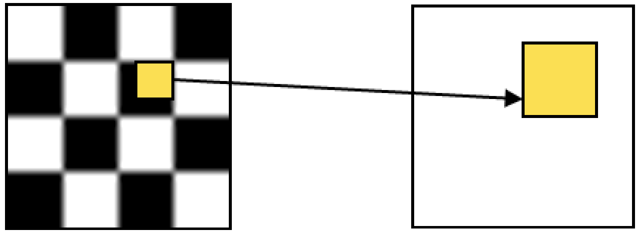

```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

# Textures – Texture Filtering

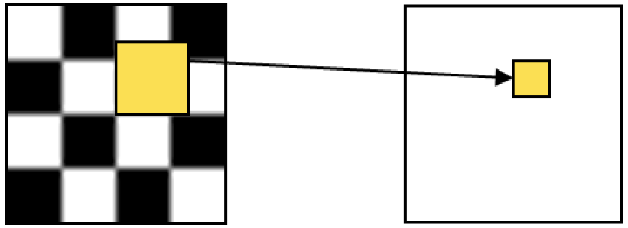
- Texture coordinates do not depend on resolution but can be any floating point value, thus OpenGL has to figure out which **texture pixel (also known as a texel )** to map the texture coordinate to.
- This becomes especially important if you have a **very large object and a low resolution texture**.
- You probably guessed by now that OpenGL has options for this texture filtering as well.



# Textures – Texture Filtering



Magnification
one texel to many pixels



Minification
many texels to one pixel

Example:

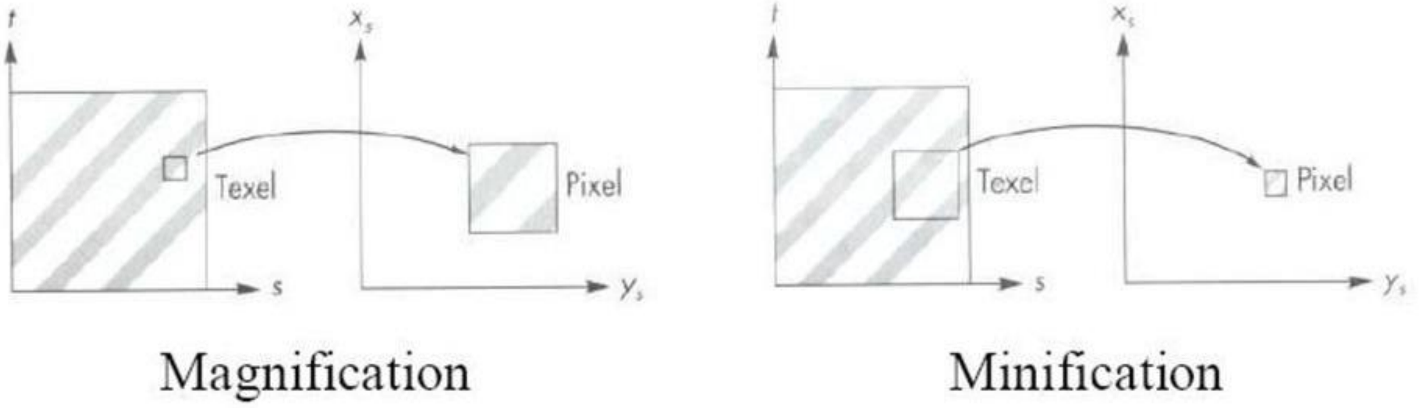
Zoom In

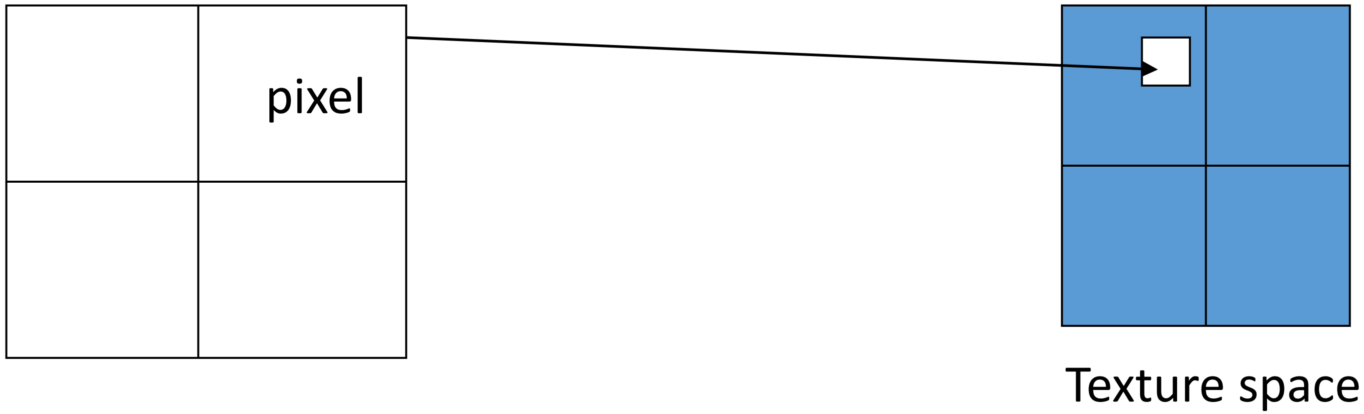
many pixels correspond to one texel
→ “blockiness” / jaggies / aliasing

solution: apply averaging
(magnification filter)

# Textures – Texture Filtering

There are several options available but for now we'll discuss the most important options: **GL_NEAREST and GL_LINEAR.**

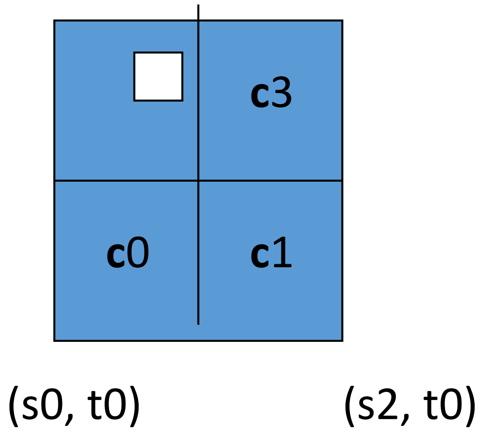


# Textures – Texture Filtering

**GL_NEAREST** (also known as nearest neighbor or point filtering) is the **default** texture filtering method of OpenGL.

- When set to GL_NEAREST, OpenGL selects the **texel that center is closest to the texture coordinate**.
- In Figure, you can see 4 pixels where the cross represents the exact texture coordinate.
- The upper-left texel has its center closest to the texture coordinate and is therefore chosen as the sampled color:




GL_LINEAR (also known as (bi)linear filtering) takes an interpolated value from the texture coordinate's neighboring texels, approximating a **color between the texels.**

- The smaller the distance from the texture coordinate to a texel's center, the more that texel's color contributes to the sampled color.
- In Figure, you can see that a mixed color of the neighboring pixels is returned:

Bilinear interpolation

b1 = (s – s0)/(s2 – s0)

b2 = (t – t0)/(t2 – t0)

**c** = (1-b2)*( (1-b1)*c0 + b1*c1)
  + b2*( (1-b1)*c2 + b1*c3)

# Textures – Texture Filtering

**GL_NEAREST** (also known as nearest neighbor or point filtering) is the **default** texture filtering method of OpenGL.

- When set to GL_NEAREST, OpenGL selects the **texel that center is closest to the texture coordinate**.
- In Figure, you can see 4 pixels where the cross represents the exact texture coordinate.
- The upper-left texel has its center closest to the texture coordinate and is therefore chosen as the sampled color:


GL_LINEAR (also known as (bi)linear filtering) takes an interpolated value from the texture coordinate's neighboring texels, approximating a **color between the texels.**

- The smaller the distance from the texture coordinate to a texel's center, the more that texel's color contributes to the sampled color.
- In Figure, you can see that a mixed color of the neighboring pixels is returned:

# Textures – Texture Filtering

But what is the visual effect of such a texture filtering method?  Let's see how these methods work when using a texture with a low resolution on a large object –Magnification- (texture is therefore scaled upwards and individual texels are noticeable):

- **GL_NEAREST** results in blocked patterns where we can clearly see the pixels that form the texture while **GL_LINEAR** produces a smoother pattern where the individual pixels are less visible.

- **GL_LINEAR** produces a more realistic output, but some developers prefer a more 8-bit look and as a result pick the GL_NEAREST option.

GL_NEAREST GL_LINEAR

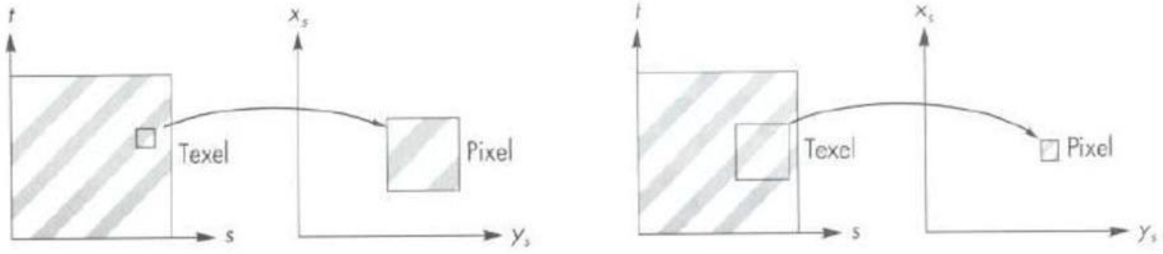
GL_NEAREST GL_LINEAR

# Textures – Texture Filtering

- **Texture filtering can be set for magnifying and minifying operations** (when scaling up or downwards) so you could for example use nearest neighbor filtering when textures are scaled downwards and linear filtering for upscaled textures.

- We thus have to specify the filtering method for both options via **glTexParameter***.

- The code should look similar to setting the wrapping method:




Magnification Minification

GL_NEAREST

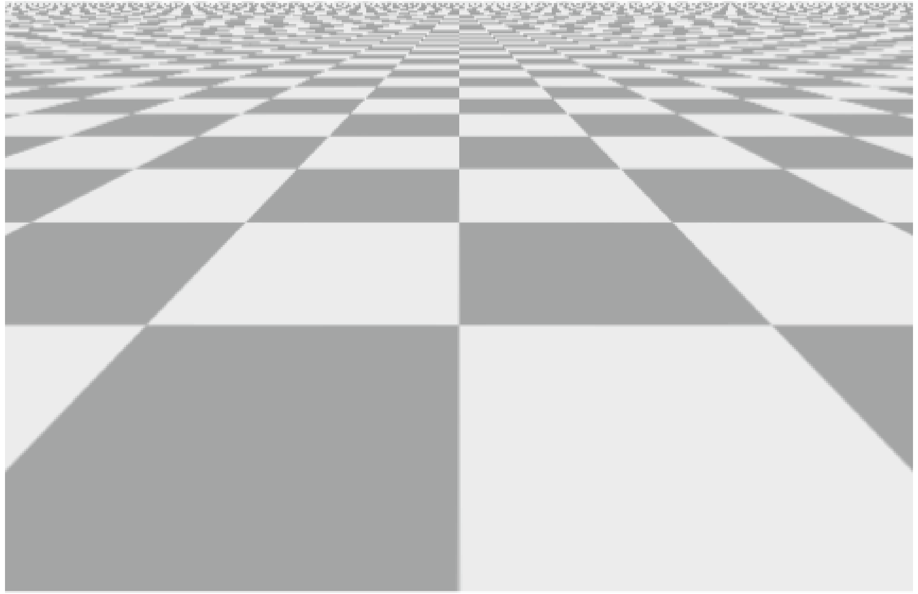
GL_LINEAR

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

# Textures – Texture Filtering - Mipmaps

Mip mapping is a technique used to prevent **texel swimming**.

- **Texel swimming** happens because the further away a textured pixel is, the more super sampled it becomes.
- That is, the span of **3 texels on the source image are compressed into perhaps a single pixel on screen**.
- When applying a perspective distortion to an image, the swimming artifacts appear.



None

This will produce visible artifacts on small objects, not to mention the waste of memory bandwidth using high resolution textures on small objects.

# Textures – Texture Filtering - Mipmaps

To solve this issue OpenGL uses a concept called MIPMAPS

Mipmaps consist on using several resolutions of an image in memory, and sampling the correct one depending on how far away screen pixel is. This way you are more likeley to get a 1 to 1 texture mapping from source to screen and less likeley to see texel swimming.

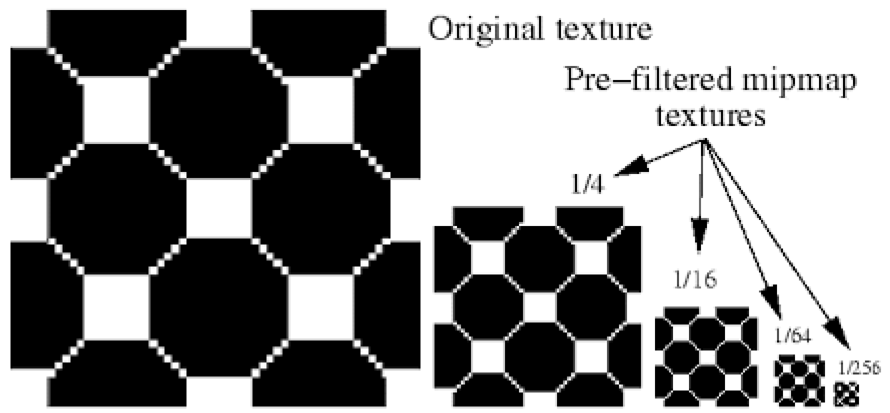
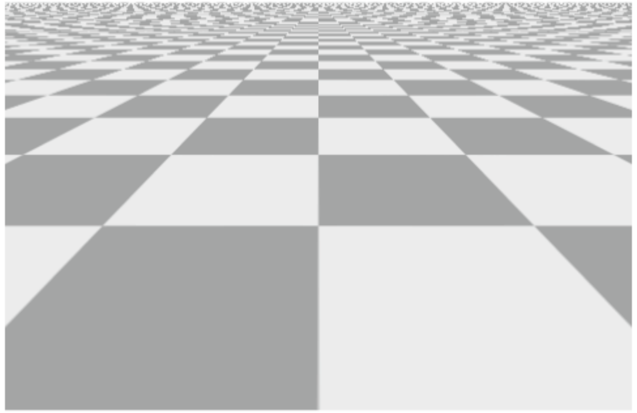
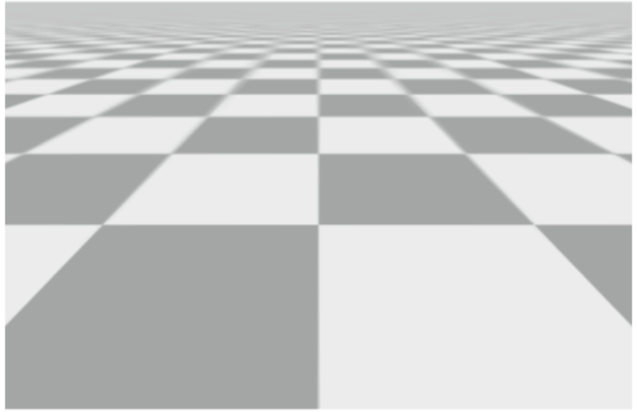



Figure 28. Multiple Levels of Texture Detail using Mipmaps



None

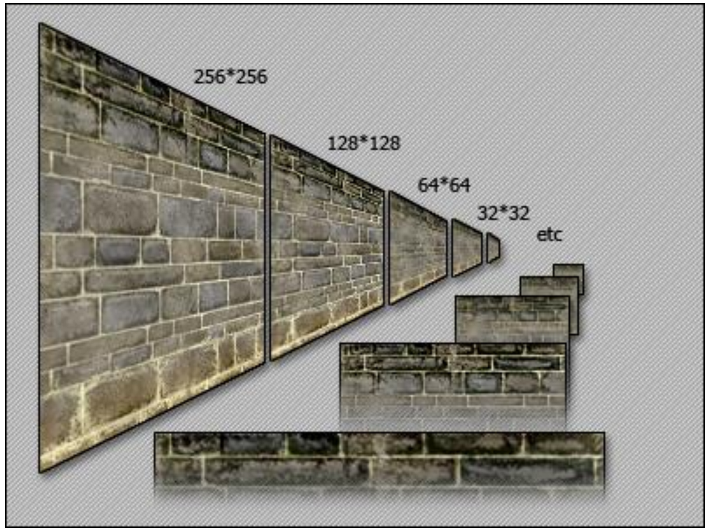


Linear

# Textures – Texture Filtering - Mipmaps

To solve this issue OpenGL uses a concept called MIPMAPS

After a certain distance threshold from the viewer, OpenGL will use a different mipmap texture that best suits the distance to the object. Because the object is far away, **the smaller resolution will not be noticeable to the user**.





There's **less cache memory** involved when sampling that part of the mipmaps.

# Textures – Texture Filtering - Mipmaps

Creating a collection of mipmapped textures for each texture image is cumbersome to do manually, but luckily OpenGL is able to do all the work for us with a single call to glGenerateMipmaps after we've created a texture.

- When switching between mipmaps levels during rendering OpenGL might show some artifacts **like sharp edges visible between the two mipmap layers**.

- Just like normal texture filtering, it is also possible to filter between mipmap levels using **NEAREST and LINEAR filtering for switching between mipmap levels**.

# Textures – Texture Filtering - Mipmaps

To specify the filtering method between mipmap levels we can replace the original filtering methods with one of the following four options:

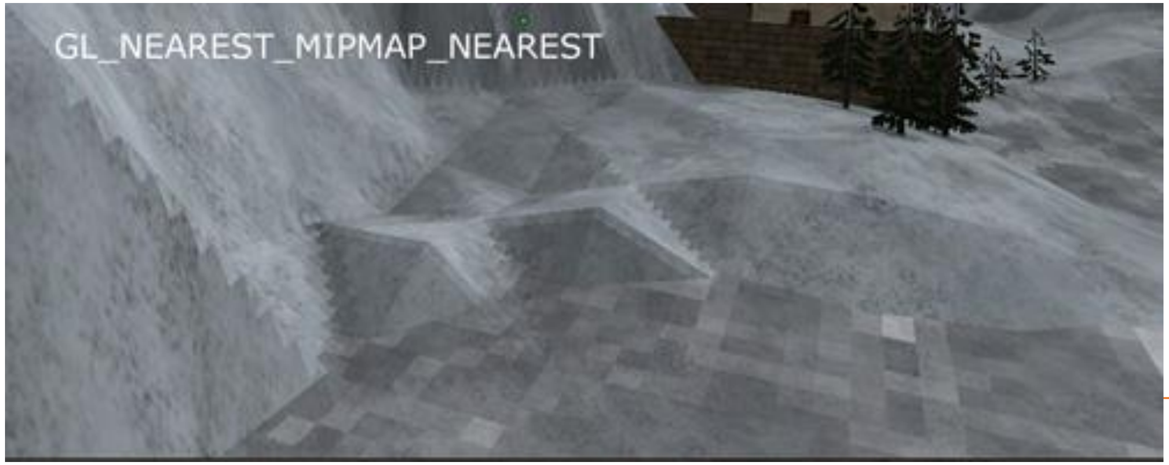
- **GL_NEAREST_MIPMAP_NEAREST:** takes the nearest mipmap to match the pixel size and uses nearest neighbor interpolation for texture sampling.
- **GL_LINEAR_MIPMAP_NEAREST**: takes the nearest mipmap level and samples that level using linear interpolation.
- **GL_NEAREST_MIPMAP_LINEAR:** linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation.
- **GL_LINEAR_MIPMAP_LINEAR:** linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation.

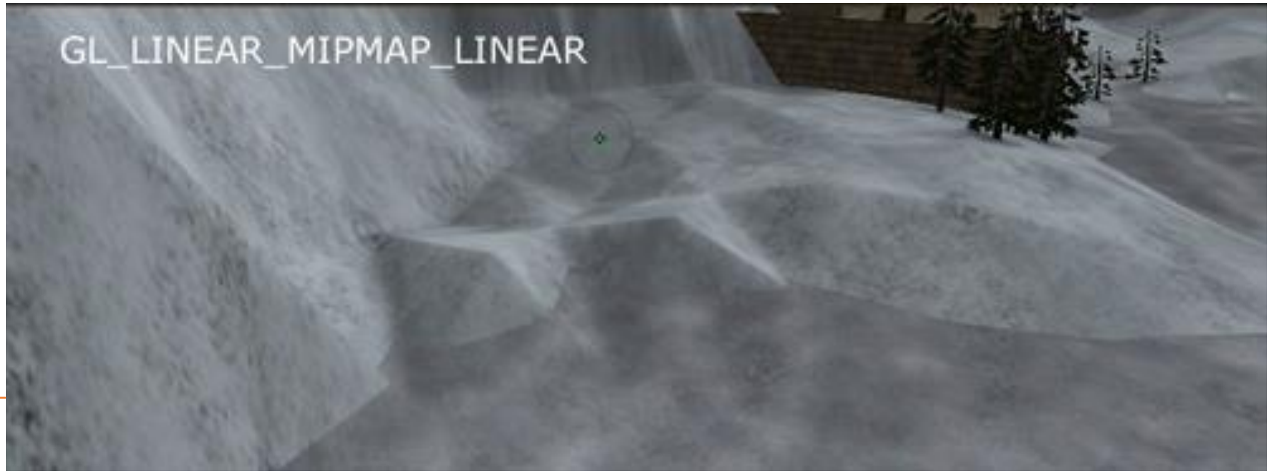
# Textures – Texture Filtering - Mipmaps

Just like texture filtering we can set the filtering method to one of the 4 aforementioned methods using **glTexParameteri**:

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

A common **mistake** is to set one of **the mipmap filtering options as the magnification filter**. This doesn't have any effect since mipmaps are primarily used for when textures get downscaled: texture magnification doesn't use mipmaps and giving it a mipmap filtering option will generate an OpenGL **GL_INVALID_ENUM** error code.



# Textures – Loading and Creating Textures

- Texture images can be stored in dozens of file formats, each with their own structure and ordering of data
- One solution would be to choose a file format we'd like to use, say .PNG and write our own image loader to convert the image format into a large array of bytes.
- More file formats? You'd then have to write an image loader for each format you want to support.

Another solution is to use an image-loading library that supports several popular formats and does all the hard work for us. A library like **stb_image.h**.

# Textures – Loading and Creating Textures

## stb_image.h

stb_image.h is a very popular single header image loading library by Sean Barrett

Download the single header file, add it to your project as **stb_image.h** and create an additional C++ file with the following code:

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

By defining **STB_IMAGE_IMPLEMENTATION** the preprocessor modifies the header file such that it only contains the relevant definition source code, effectively turning the header file into a .cpp file, and that's about it. Now simply include stb_image.h somewhere in your program and compile.

Download link: https://github.com/nothings/stb/blob/master/stb_image.h

# Textures – Loading and Creating Textures

To load an image using **stb_image.h** we use its **stbi_load** function:

```
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
```

- The function first takes as input the location of an image file.
- It then expects you to give **three ints** as its second, third and fourth argument that stb_image.h will fill with the resulting **image's width, height and number of color channels**.
- We need the image's width and height for generating textures later on.

Image → container.jpg

Image source: https://learnopengl.com/img/textures/container.jpg

# Textures – Loading and Creating Textures

## Generating a Texture

Like any of the previous objects in OpenGL, textures are referenced with an ID; let's create one:

```
unsigned int texture;
glGenTextures(1, &texture);
```

The **glGenTextures** function first takes as input how many textures we want to generate and stores them in a unsigned int array given as its second argument (in our case just a single unsigned int).

- Just like other objects we need to bind it so any subsequent texture commands will configure the currently bound texture:

```
glBindTexture(GL_TEXTURE_2D, texture);
```

# Textures – Loading and Creating Textures

## Generating a Texture

Textures are generated with **glTexImage2D**:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

- The **first argument** specifies the **texture target**; setting this to GL_TEXTURE_2D means this operation will generate a texture on the currently bound texture object at the same target (so any textures bound to targets GL_TEXTURE_1D or GL_TEXTURE_3D will not be affected).

- The **second argument** specifies the **mipmap level** for which we want to create a texture for if you want to set each mipmap level manually, but we'll leave it at the base level which is **0**.

- The **third argument** tells OpenGL in what kind of **format** we want to **store the texture.** Our image has only RGB values so we'll store the texture with RGB values as well.

# Textures – Loading and Creating Textures

## Generating a Texture

Textures are generated with **glTexImage2D**:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
glGenerateMipmap(GL_TEXTURE_2D);
```

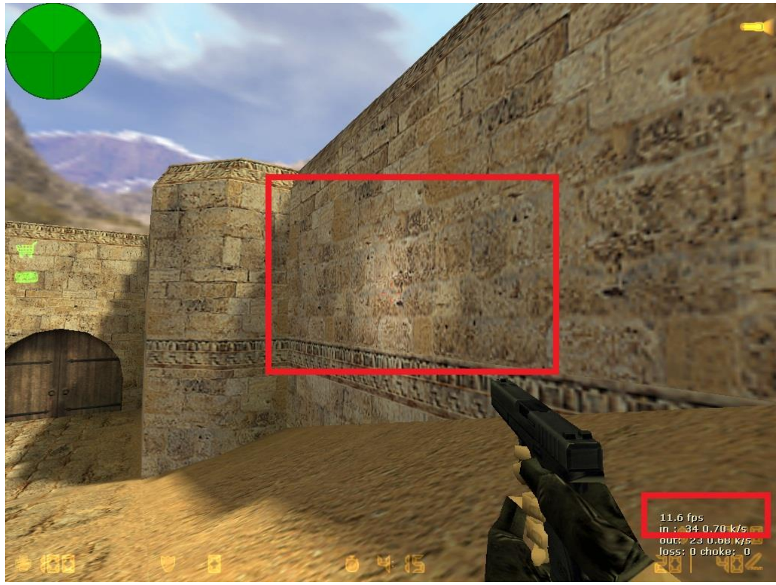
- The **4th and 5th** argument sets the **width and height of the resulting texture**. We stored those earlier when loading the image so we'll use the corresponding variables.
- The next argument should always be 0 (some legacy stuff).
- The **7th and 8th** argument specify **the format and datatype** of the source image. We loaded the image with *RGB values and stored them as chars (bytes)* so we'll pass in the corresponding values.
- The **last argument** is the actual **image data**.

# Textures – Loading and Creating Textures

## Generating a Texture

Once **glTexImage2D** is called, the currently bound texture object now has the texture image attached to it.

- It only has the base-level of the texture image loaded.
- If we want to use mipmaps we have to specify all the different images manually (by continually incrementing the second argument)
- **Or we could call glGenerateMipmap after generating the texture.**
- This will automatically generate all the required mipmaps for the currently bound texture.



After we're done generating the texture and its corresponding mipmaps, it is good practice to free the image memory:

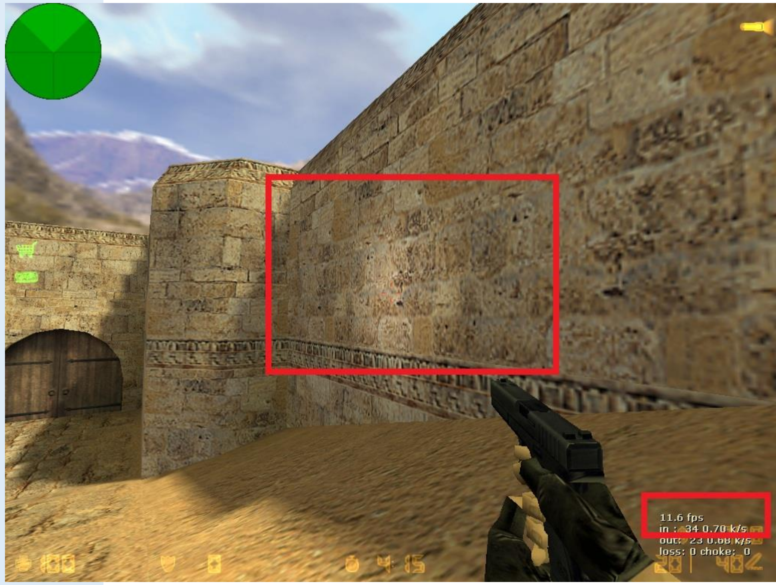
```
stbi_image_free(data);
```

# Textures – Loading and Creating Textures

## Generating a Texture

The whole process of generating a texture thus looks something like this:

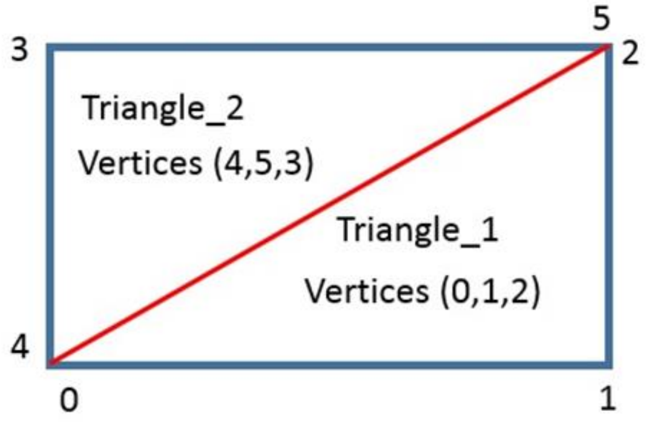
```
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// set the texture wrapping/filtering options (on the currently bound texture object)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load and generate the texture
int width, height, nrChannels;
unsigned char *data = stbi_load("container.jpg", &width, &height, &nrChannels, 0);
if (data)
{
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
   glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
   std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);
```



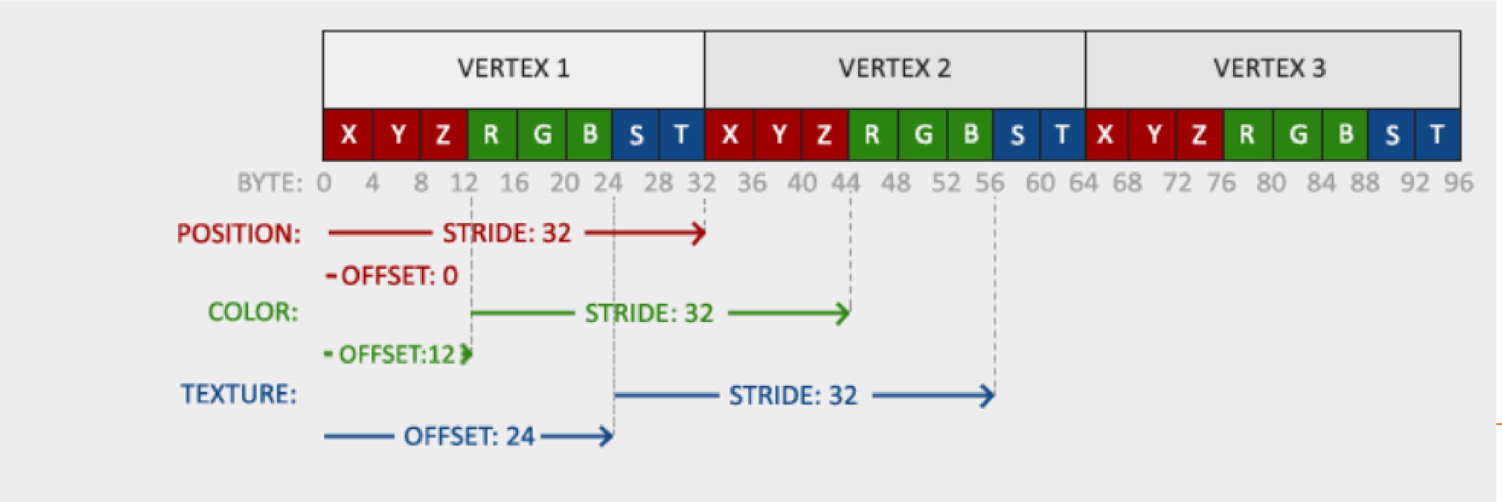
# Textures - Applying textures

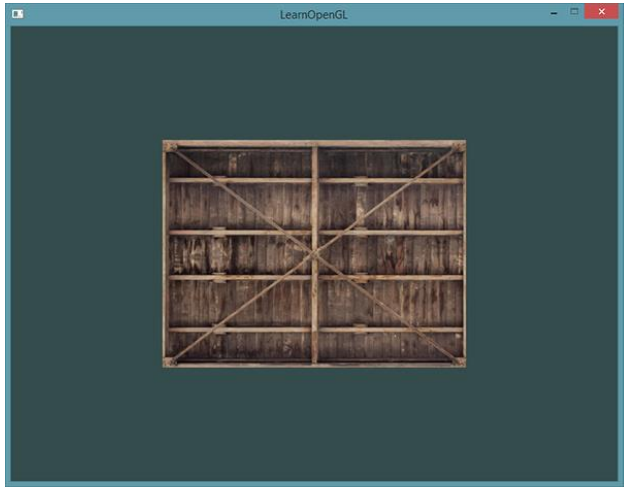
- We need to inform OpenGL how to sample the texture so we'll have to update the **vertex data** with the texture coordinates.
- We will use the rectangle shape:

```
float vertices[] = {
    // positions          // colors           // texture coords
     0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,   // top right
     0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f    // top left
};
```

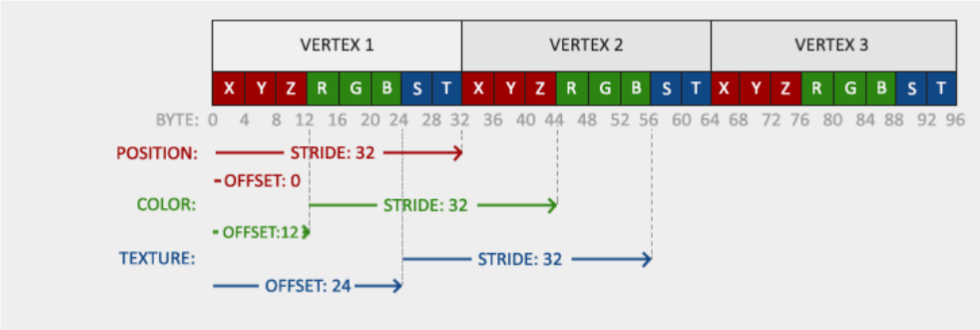


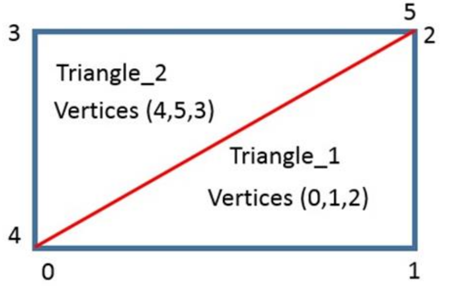

Since we've added an extra vertex attribute we again have to notify OpenGL of the **new vertex format**:



# Textures - Applying textures



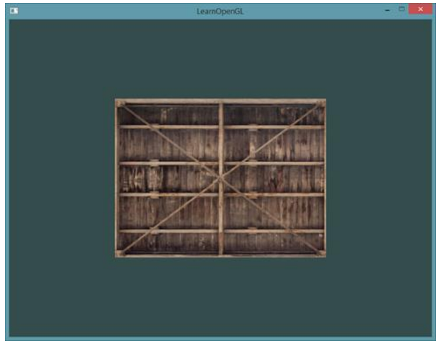

## glVertexAttribPointer

Shaders

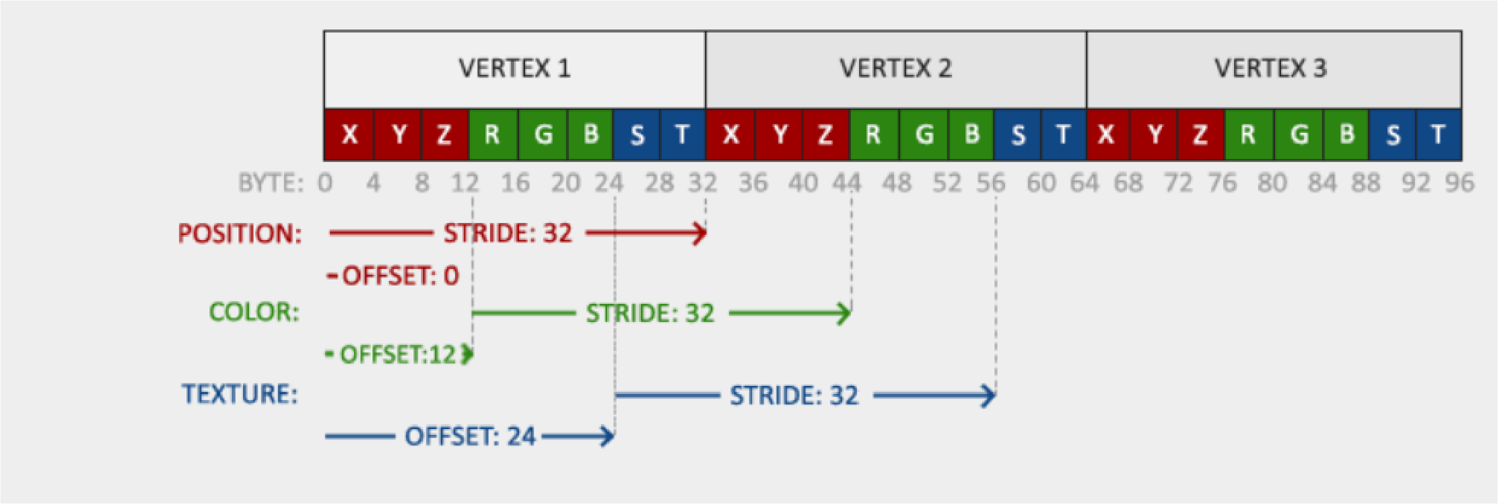
The function `glVertexAttribPointer` specifies how OpenGL should interpret the vertex buffer data whenever a drawing call is made. The interpretation specified is stored in the currently bound vertex array object saving us all quite some work.

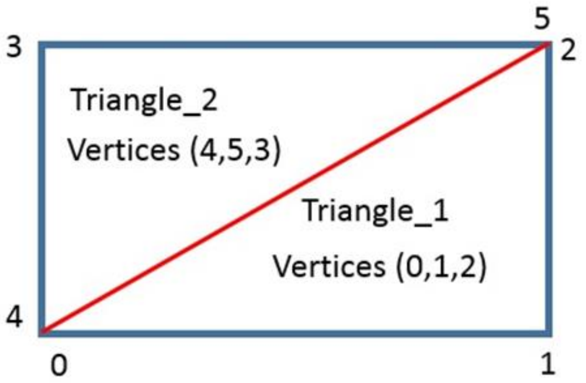
The parameters of `glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer)` are as follows:

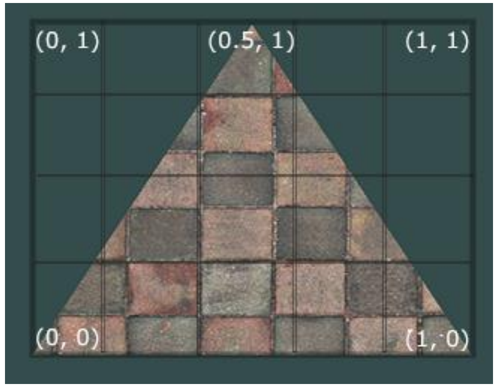
- `index`: Specifies the index of the vertex attribute.
- `size`: Specifies the number of components per vertex attribute. Must be 1, 2, 3, 4.
- `type`: Specifies the data type of each component in the array.
- `normalized`: Specifies wheter data should be normalized (clamped to the range -1 to 1 for signed values and 0 to 1 for unsigned values).
- `stride`: Specifies the byte offset between consecutive vertex attributes. If stride is 0, the generic vertex attributes are understood to be tightly packed in the array
- `pointer`: Specifies an offset of the first component of the first vertex attribute in the array.



# Textures - Applying textures

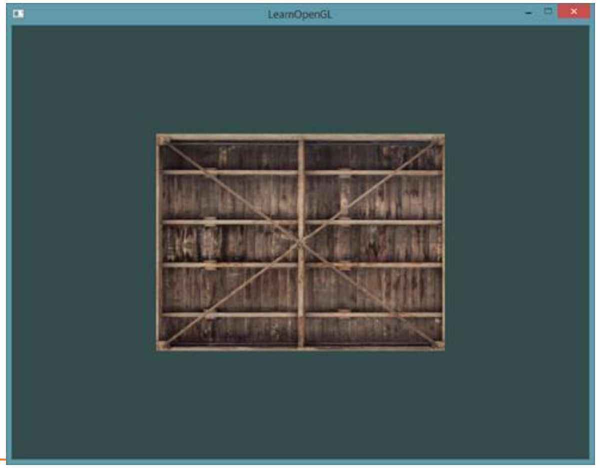


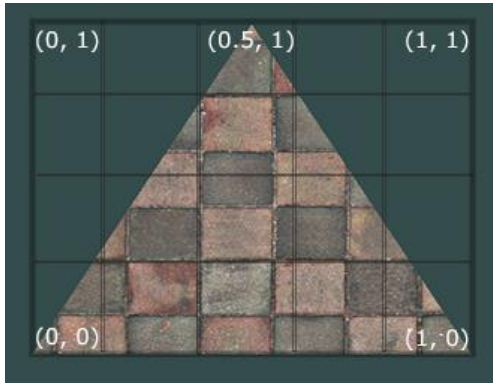




```
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);
```

Note that we have to adjust the stride parameter of the previous two vertex attributes to 8 * sizeof(float) as well.

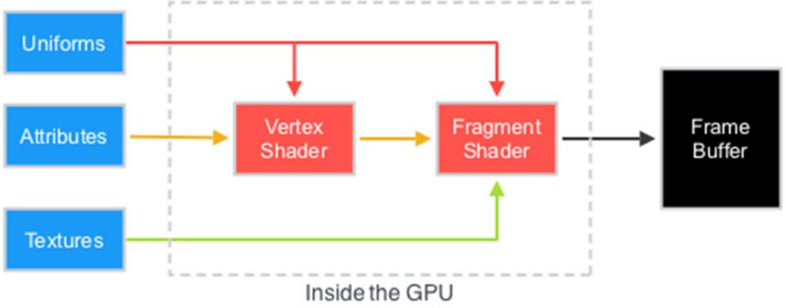


# Textures - Applying textures



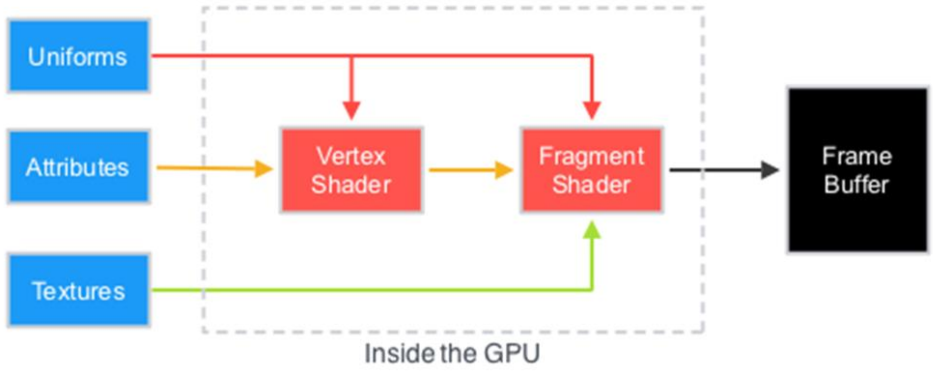

Next we need to alter the **vertex shader** to accept the texture coordinates as a vertex attribute and then forward the coordinates to the fragment shader:

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = aTexCoord;
}
```




The **fragment shader** should then accept the **TexCoord** output variable as an input variable.

# Textures - Applying textures



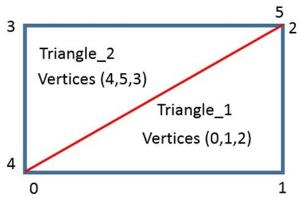

- The **fragment shader** should also have access to **the texture object**, but how do we pass the texture object to the fragment shader?

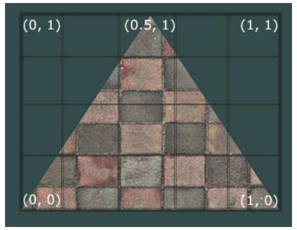
- GLSL has a **built-in data-type** for texture objects called a **sampler** that takes as a postfix the texture type we want e.g. sampler1D, sampler3D or in our case sampler2D.

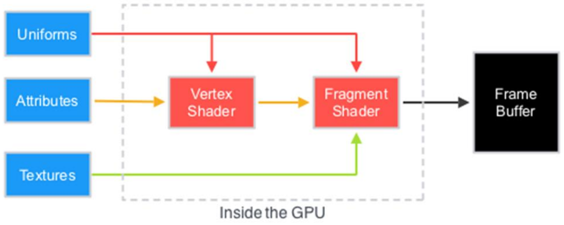
- We can then add a texture to the fragment shader by simply declaring a uniform sampler2D that we later assign our texture to.

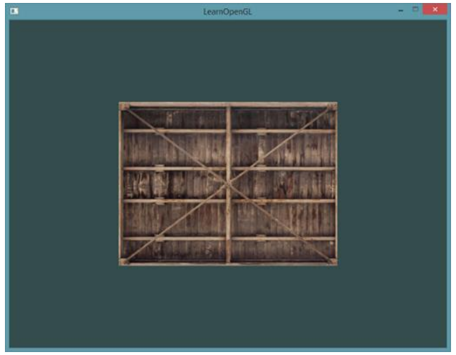
```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

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

# Textures - Applying textures



- To sample the color of a texture we use GLSL's built-in texture function that takes as its **first argument a texture sampler** and as its **second argument** the corresponding **texture coordinates**.

- The texture function then samples the corresponding color value using the texture parameters we set earlier.

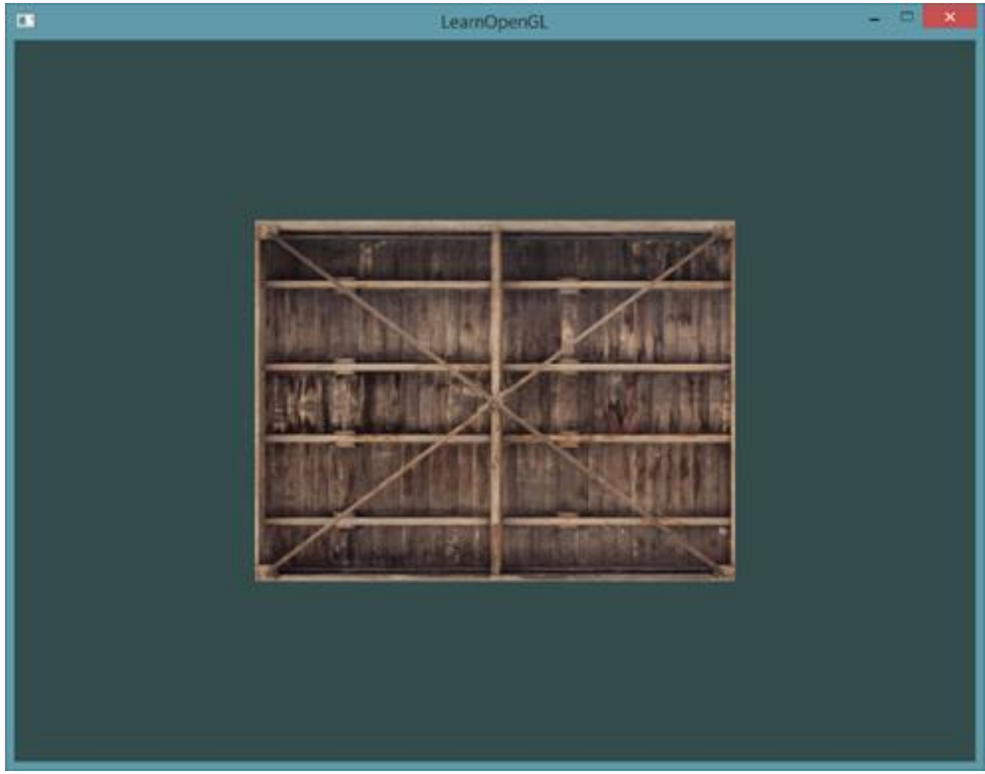
- The **output** of this fragment shader is then **the (filtered) color of the texture at the (interpolated) texture coordinate**.

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

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

# Textures - Applying textures

Finally, **Bind** the texture before calling **glDrawElements** and it will then automatically assign the texture to the fragment shader's sampler:

```
glBindTexture(GL_TEXTURE_2D, texture);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```




**Exercise 8:**

Apply the Textures in OpenGL using your own texture files and coordinates. C2_Exercise_8_TexureIntro

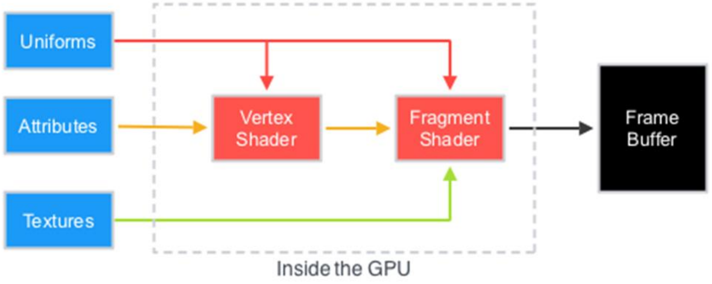
If your texture code doesn't work or shows up as completely black, continue reading and work your way to the last example that should work. On some drivers it is required to assign a texture unit to each sampler uniform, which is something we'll discuss further in this chapter.

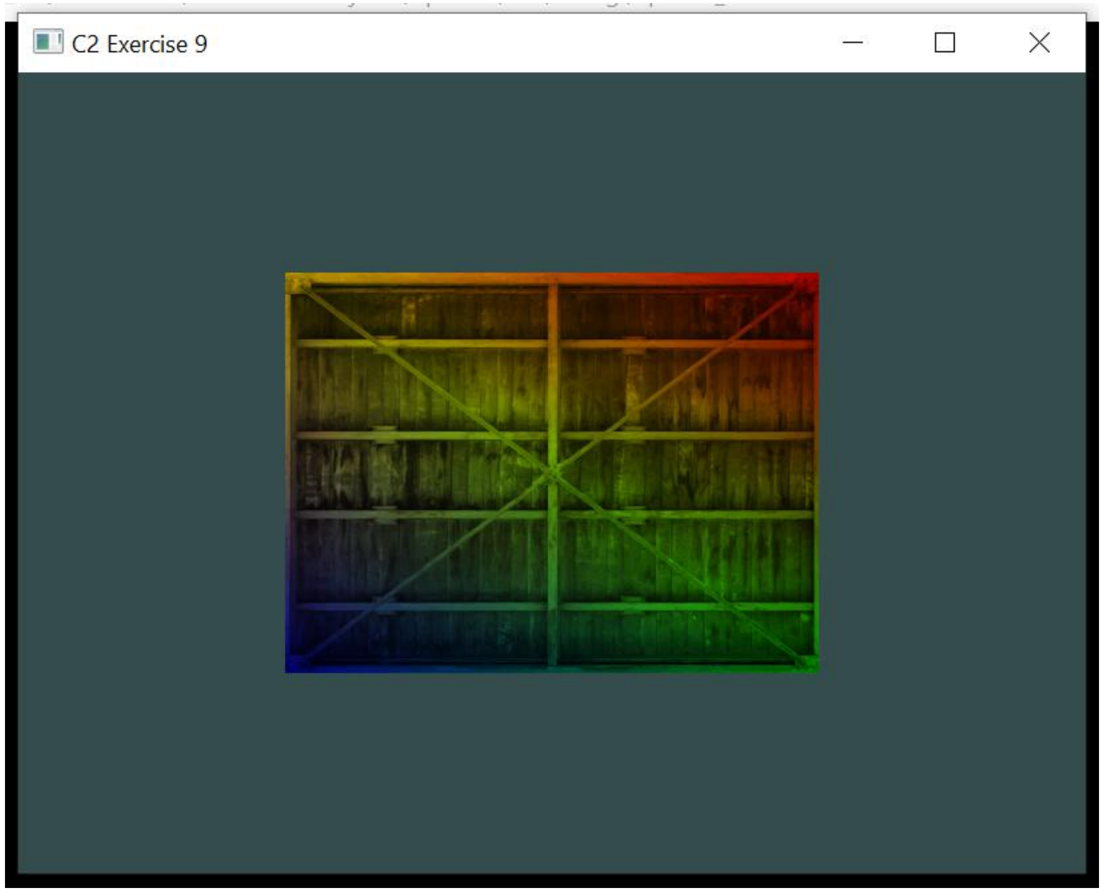
# Textures - Applying textures

**Exercise 8:**

Apply the Textures in OpenGL using your own texture files and coordinates. C2_Exercise_8_TexureIntro

- Mix the resulting texture color with the vertex colors. We simply multiply the resulting texture color with the vertex color in the fragment shader to mix both colors:




```
FragColor = texture(ourTexture, TexCoord) * vec4(ourColor, 1.0);
```