

TEXTURES

A **texture** is a 2D image (usually) that is used to add detail to an object.

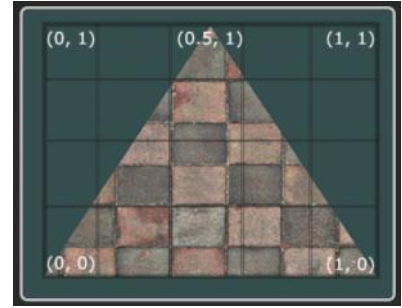
To map a **texture** to an object, each **vertex** must have a **texture coordinate** associated with it that specifies what part of the **texture** image to sample from.

- Fragment interpolation handles the in-between of the texture coordinates for the other fragments.

Texture coordinates range from 0 to 1 in the x and y axis, from bottom-left to top-right.

Retrieving the **texture** color using **texture coordinates** is called **sampling**.

- Texture sampling has a loose interpretation and can be done in many different ways, so it is our job to tell OpenGL how it should sample its textures.



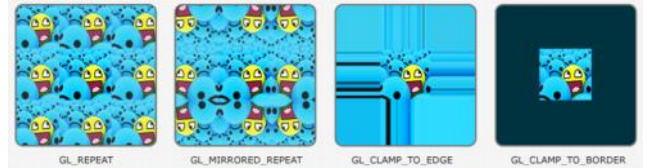
Texture Wrapping

By default, if you specify **texture coordinates** outside of the range of (0, 0) to (1, 1), the **texture** will be repeated.

You can also tell OpenGL specifically how to interpret **texture coordinates** outside this range:

- GL_REPEAT**: The default wrapping for **textures**. Repeats the **texture** image.
- GL_MIRRORED_REPEAT**: Same as **GL_REPEAT**, but mirrors the **texture** 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.

These options can be set per coordinate axis(s, t (and r if using 3D textures) equivalent to x, y, z).



Configure the **texture coordinates** of the brick **texture** to a triangle and set the wrapping type to mirrored. Also, set up code for clamping to the borders.

- [Brick Wall Texture](#)
- [Configuring Texture Coordinates and Wrapping](#)

Texture Filtering

Texture coordinates do not depend on resolution but can be any floating point value. Because of this, OpenGL has to figure out which **texture pixel**, or **texel**, to map the coordinate to. This is known as **texture filtering**.

- This becomes especially important when mapping a low-resolution **texture** to a large object.

There are two common ways to approach this problem in OpenGL (though there are others):

- GL_NEAREST**: The default **filtering** for **textures**. The **texel** that's center is closest to the **texture coordinate** is the sampled color for the **fragment**.
 - Also known as **Nearest Neighbor** or **Point filtering**.
 - Causes the **texture** to look pixelated.
- GL_LINEAR**: The sampled color for the **fragment** is interpolated from the **texture coordinate's** neighboring **texels**, approximating a color between the **texels**.
 - Also known as **Linear/Bilinear filtering**.
 - Causes the **texture** to look blurry.



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

Configure the **texture filtering** technique for the minifying operation to be **nearest neighbor filtering** and the magnifying operation to be **linear filtering**.

- [Configuring Minifying and Magnifying Operations](#)

Mipmaps

A **mipmap** is a collection of **texture** images where each subsequent **texture** is a fourth the size compared to the previous one.

After a certain distance from the camera has been reached, OpenGL will use a different **mipmap texture** that best suits the distance to the object.

- Makes **texel** sampling easier and more accurate, reducing the cache memory required for sampling.

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

- It is possible to filter between **mipmap** levels using **NEAREST** and **LINEAR filtering** (just like with normal **texture filtering**) for switching between **mipmap** levels.
- To do so, we can replace the original **filtering** methods with one of the following:
 - 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** to match the 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**.

NOTE: **Mipmap filtering** options do not typically get set for magnifying operations because **mipmaps** are primarily used for when **textures** get downsampled.

Change the minifying operation to **linearly interpolate between mipmaps** and **sample the interpolated level using nearest neighbor interpolation**. Also, change the magnifying operation to use **linear filtering** for **textures**.

- [Configuring Mipmaps](#)

Loading and Creating Textures

In order to use **textures** in an application, their data needs to be loaded, depending on their file type (.png, .jpg, .gif, etc.), using an **image loader**.

Rather than creating our own **image loader** for each different type of image we want to load, we can use an **image-loading** library that supports multiple popular formats.

std_image.h

stb_image.h is a very popular, single-header **image-loading** library that is able to load most popular file formats and is easy to integrate into a project.

- [STB Image Header Repository](#)

For the following **texture** sections, use this **texture**:

- [Wooden Container Texture](#)

Add **texture coordinates** as a **vertex attribute**. Load the wooden container **texture** image using **stb_image.h**, then generate and configure the **texture** object.

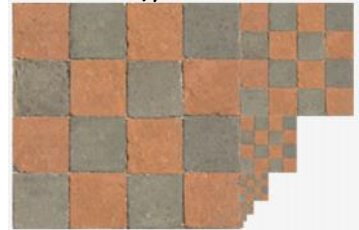
- [Loading and Configuring a Texture](#)

Texture Units

We can set multiple **textures** at once in a **fragment shader** by assigning a location value to the sampler. This location of a **texture** is commonly known as a **texture unit**.

Texture units allow us to use more than one **texture** in our shaders.

Mipmapped Texture



- By assigning **texture units** to the samplers, we can bind to multiple **textures** at once as long as we activate the corresponding **texture unit** first.
- OpenGL should have at least 16 **texture units** for you to use. You can also, for example, select the 9th **texture unit** like this: `GL_TEXTURE0 + 8` (useful for iteration)

Assign the container **texture** to the 0th **texture unit** and the smiley **texture** to the 1st **texture unit**. Display a 50/50 mix of the **textures** on the object.

- [Smiley Texture](#)
- **NOTE:** Images usually have (0, 0) at the top-left of the image, but OpenGL expects (0, 0) at the bottom-left. Don't forget to flip the image right-side up!
- [Using Texture Units](#)

EXERCISES

1. Make sure **only** the happy face looks in the other/reverse direction by changing the fragment shader.
2. Experiment with the different texture wrapping methods by specifying texture coordinates in the range of `0.0f` to `2.0f` instead of `0.0f` to `1.0f`. See if you can display 4 smiley faces on a single container image clamped at its edge. Also, see if you can experiment with other wrapping methods as well.
3. Try to display only the center pixels of the texture image on the rectangle in such a way that the individual pixels are getting visible by changing the texture coordinates. Try to set the texture filtering method to `GL_NEAREST` to see the pixels more clearly.
4. Use a uniform variable as a mix function's third parameter to vary the amount the two textures are visible. Use the up and down arrow keys to change how much the container or the brick wall is visible.

Configuring Texture Coordinates and Wrapping

Monday, March 21, 2022 1:24 PM

```
// Texture coordinates from triangle
float texCoords[] = {
    0.5f, 1.0f,      // top-center
    0.0f, 0.0f,      // bottom-left
    1.0f, 0.0f       // bottom-right
};
```

```
// Sets the wrapping type on the s and t (x and y) axes for mirrored repeating
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

```
// Sets the wrapping type on the s and t axes for clamping to the border
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

Configuring Minifying and Magnifying Operations

Monday, March 21, 2022 2:14 PM

```
// Sets the minifying operation to use nearest neighbor filtering and the magnifying operation to use linear filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Configuring Mipmaps

Monday, March 21, 2022 2:40 PM

```
// Sets the minifying filter to use linear filtering for mipmaps and textures
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Loading and Configuring a Texture

Monday, March 21, 2022 3:59 PM

Fragment Shader

```
#version 330 core

out vec4 FragColor;
in vec4 vertexColor;
in vec2 texCoord;

uniform sampler2D texture1;

void main() {
    FragColor = texture(texture1, texCoord);
}
```

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec4 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec4 vertexColor;
out vec2 texCoord;

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

main.cpp

```
// Generates and binds a texture object
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// Loads the textured data and configures it to a texture object with automatically generated mipmaps
int width, height, numChannels;
unsigned char* data = stbi_load("Textures/container.jpg", &width, &height, &numChannels, 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;
}
// Frees the image data memory and unbinds the texture object
stbi_image_free(data);
glBindTexture(GL_TEXTURE_2D, 0);
```

Don't forget to bind the texture in the render loop!

Fragment Shader

```
#version 330 core

out vec4 FragColor;
in vec4 vertexColor;
in vec2 texCoord;

uniform sampler2D texture1;
uniform sampler2D texture2;

void main() {
    FragColor = mix(texture(texture1, texCoord), texture(texture2, texCoord), 0.5);
}
```

main.cpp

```
// Generates and binds a texture object to the 0th texture unit
unsigned int texture1;
glGenTextures(1, &texture1);
glActiveTexture(GL_TEXTURE0); // This is technically redundant because texture unit 0 is active by default.
glBindTexture(GL_TEXTURE_2D, texture1);

// Loads the textured data and configures it to a texture object with automatically generated mipmaps
int width, height, numChannels;
stbi_set_flip_vertically_on_load(true); // Flips images to be right-side up, due to different orientation standards
unsigned char* data = stbi_load("Textures/container.jpg", &width, &height, &numChannels, 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 texture1" << std::endl;
}
glBindTexture(GL_TEXTURE_2D, 0);

// Generates and binds a texture object to the 1st texture unit
unsigned int texture2;
glGenTextures(1, &texture2);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);
data = stbi_load("Textures/brick_wall.jpg", &width, &height, &numChannels, 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 texture2" << std::endl;
}

// Frees the image data memory and unbinds the texture object
stbi_image_free(data);
glBindTexture(GL_TEXTURE_2D, 0);

// Tells OpenGL to use the shaders in the shader program for rendering
shaderProgram.use();

// Sets the texture uniforms
glUniform1i(glGetUniformLocation(shaderProgram.id, "texture1"), 0); // Set it manually
shaderProgram.setInt("texture2", 1); // Set it using our shader class
```

Render Loop

```
// Binds the textures to render
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
```

```
glBindTexture(GL_TEXTURE_2D, texture2);

// Binds the object we want to render
glBindVertexArray(vao);

// Renders the rectangle to the screen
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

// Unbinds the vao
glBindVertexArray(0);

// Unbinds the textures
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, 0);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, 0);
```