Laboratorio 05

OpenGL

Part II

Outline

- Shaders
- Texture

Shaders

Shaders are little programs that rest on the GPU and are run for each specific section of the graphics pipeline. Shaders are also very isolated programs in that they're not allowed to communicate with each other; the only communication they have is via their inputs and outputs.

GLSL

Shaders are written in the C-like language GLSL. This language is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

Shaders always begin with a version declaration, followed by a list of **input** and **output variables**, **uniforms** and its **main** function. Each shader's entry point is at its main function, where any input variables are processed. The shader output the results in its output variables.

GLSL

A shader typically has the following structure:

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
  // process input(s) and do some weird graphics stuff
  ...
  // output processed stuff to output variable
  out_variable_name = weird_stuff_we_processed;
}
```

GLSL

When specifically talking about the vertex shader, each input variable is also known as a vertex attribute. There is a maximum number of vertex attributes that can be declared, limited by the hardware.

OpenGL guarantees there are always at least 16 4-component vertex attributes available, but some hardware may allow for more.

The limit can be retrieved by querying GL_MAX_VERTEX_ATTRIBS

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " <<
nrAttributes << std::endl;
```

Types

GLSL has, like any other programming language, data types for specifying what kind of variable developers want to work with.

GLSL has most of the default basic types: `int`, `float`, `double`, `uint` and `bool`.

GLSL also features two container types, namely `vectors` and `matrices`

Vectors

A vector in GLSL is a 2,3 or 4 component container for any of the basic types just mentioned. They can take the following form (n represents the number of components):

- `vecn`: the default vector of *n* floats.
- `bvecn`: a vector of *n* booleans.
- `ivecn`: a vector of *n* integers.
- `uvecn`: a vector of *n* unsigned integers.
- `dvecn`: a vector of *n* double components.

Most of the time, the basic `vecn` will be used since floats are sufficient for most of the purposes of this course.

Vectors

Components of a vector can be accessed via `vec.x` where `x` is the first component of the vector. The `.x`, `.y`, `.z` and `.w` can be used to access the first, second, third and fourth component, respectively. GLSL also allows to use `rgba` for colors or `stpq` for texture coordinates, accessing the same components.

The vector datatype allows for some flexible component selection called **swizzling**, that makes it possible to use syntax like:

```
vec2 someVec;
vec4 differentVec = someVec.xyxx;
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

Any combination of up to 4 letters can be used create a new vector (of the same type) as long as the original vector has those components. For instance, it is not allowed to access the `.z` component of a `vec2`.

Vectors

Vectors can also be passed as arguments to different vector constructor calls, reducing the number of arguments required.

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

Inputs and outputs

GLSL defined the `in` and `out` keywords specifically to have inputs and outputs that can be used to move stuff from one shader to another. Each shader can specify inputs and outputs using those keywords and wherever an output variable matches with an input variable of the next shader stage they're passed along. The vertex and fragment shader differ a bit though.

To send data from one shader to the other it is necessary to declare an output in the sending shader and a similar input in the receiving shader. When the types and the names are equal on both sides OpenGL will link those variables together and then it is possible to send data between shaders (this is done when linking a program object).

Inputs and outputs

Exceptions 1 – Vertex shader

The vertex shader should receive some form of input otherwise it would be pretty ineffective. The vertex shader differs in its input, in that it receives its input straight from the vertex data. To define how the vertex data is organized the input variables are specified with location metadata, i.e., `layout (location = 0)`.

The vertex shader thus requires an extra layout specification for its inputs so it can be linked to the vertex data.

Inputs and outputs

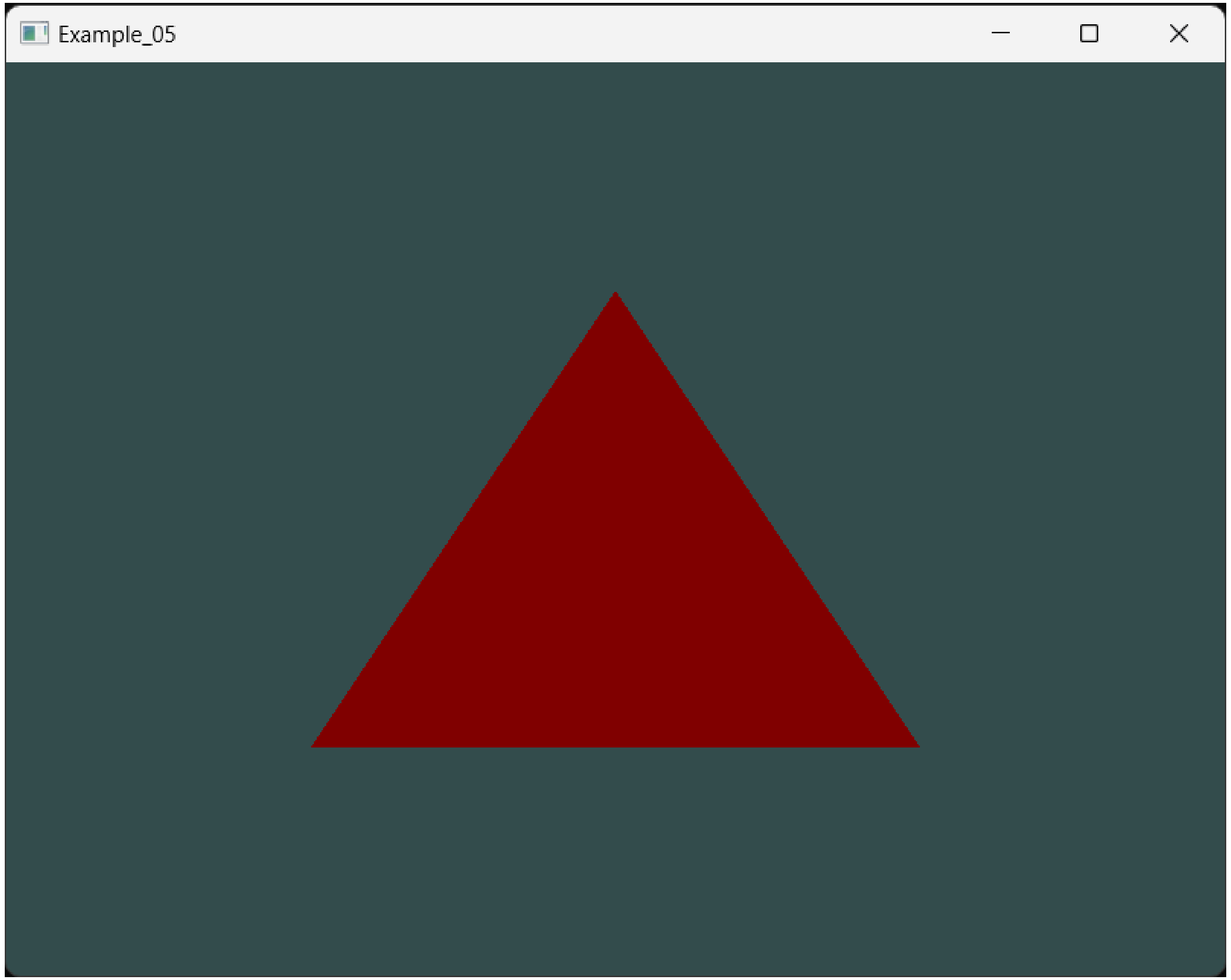
Exceptions 2 – Fragment shader

The fragment shader requires a `vec4` color output variable, since the fragment shaders needs to generate a final output color. If the operation fails hence, it is not possible to specify an output color in the fragment shader, the color buffer output for those fragments will be undefined (which usually means OpenGL will render them either black or white).

Inputs and outputs

Example5

To show how the process based on inputs and outputs works, a new example is realized. Starting from Example3, the new one lets the vertex shader decide the color for the fragment shader.



Inputs and outputs

Vertex Shader

```
#version 330 core

layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0);
    // set the output variable to a dark-red color
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0);
}
```

Inputs and outputs

Fragment Shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader
(same name and same type)

void main()
{
    FragColor = vertexColor;
}
```

Uniforms

Uniforms are another way to pass data from an application on the CPU to the shaders on the GPU. Uniforms are however slightly different compared to vertex attributes.

First of all, uniforms are **global**. Global, meaning that a uniform variable is unique per shader program object, and can be accessed from any shader at any stage in the shader program.

Second, whatever you set the uniform value to, uniforms will keep their values until they're either reset or updated.

Uniforms

A uniform in GLSL can be defined by adding the `uniform` keyword to a shader. From that point on, the newly declared uniform can be used in the shader. The code below show how to set the color for the triangle using the uniform.

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // this variable is set in the OpenGL code.

void main()
{
    FragColor = ourColor;
}
```

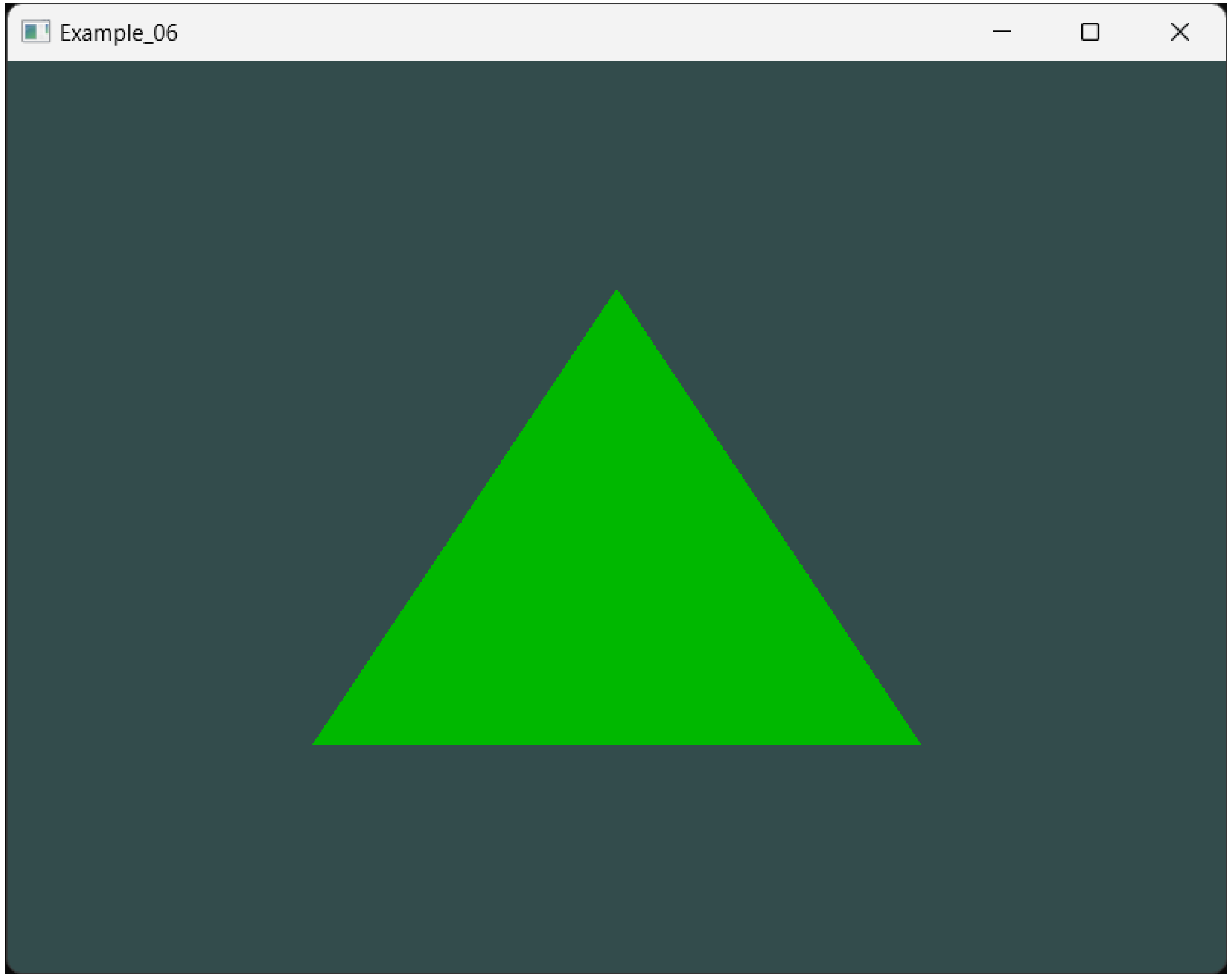
Since uniforms are global variables, they can be defined in any shader stage or in the OpenGL code part.

Uniforms

To add data to the uniform, it is necessary to find the index/location of the uniform attribute in the shader. Once the index/location of the uniform has been identified, its value can be updated.

Uniforms
Example6

It is possible to use uniforms, to gradually change the color of the triangle over time, instead of passing a single color to the fragment shader.



Uniforms

```
// be sure to activate the shader before any calls to glUniform
glUseProgram(shaderProgram);

// update shader uniform
double  timeValue = glfwGetTime();
float greenValue = static_cast<float>(sin(timeValue) / 2.0 + 0.5);
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

The function `glfwGetTime()` is used to retrieve the running time in seconds. Then the color is changed in the range 0.0-1.0 by using the `sin` function and the result is stored in `greenValue`.

Uniforms

```
// be sure to activate the shader before any calls to glUniform
glUseProgram(shaderProgram);

// update shader uniform
double  timeValue = glfwGetTime();
float greenValue = static_cast<float>(sin(timeValue) / 2.0 + 0.5);
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f);
```

The location of the `ourColor` uniform is requested by using `glGetUniformLocation`. The shader program and the name of the uniform to retrieve the location from, are provided as input to the function. If `glGetUniformLocation` returns -1, it could not find the location. Lastly, the uniform value is set using the `glUniform4f` function.

The code is added in the render loop, so the green value is calculated and the uniform updated in each render iteration.

Function overloading

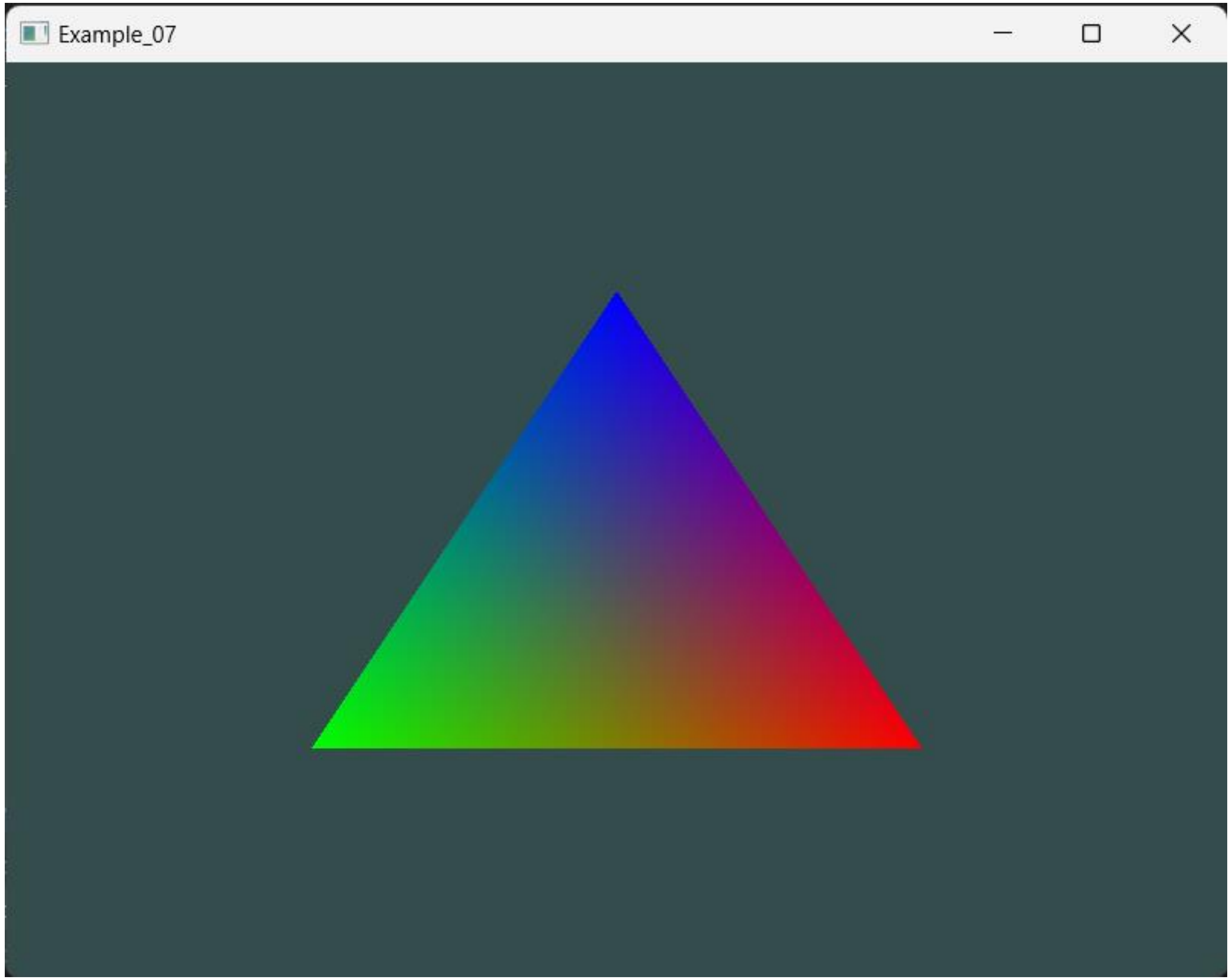
Because OpenGL is in its core a C library it does not have native support for function overloading, so wherever a function can be called with different types OpenGL defines new functions for each type required. For example, `glUniform` requires a specific postfix for the type of the uniform to be set. A few of the possible postfixes are:

- f: the function expects a float as its value.
- i: the function expects an int as its value.
- ui: the function expects an unsigned int as its value.
- 3f: the function expects 3 floats as its value.
- fv: the function expects a float vector/array as its value.

Draw a multi-colored triangle

Example7

The VBO and VAO can be configured to store not only the positional data of the vertices but also the color of each vertex.



Draw a multi-colored triangle

Adding more attributes

The color data can be added to the vertex data as 3 additional floats to the `vertices` array. To assign a red, green and blue color to each of the corners of our triangle the following code is used:

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

Draw a multi-colored triangle

Adding more attributes

Since more data have to be send to the vertex shader, it is necessary to adjust the vertex shader to also receive the color value as a vertex attribute input. Note that the location of the `aColor` attribute is set to 1 with the layout specifier:

```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

Draw a multi-colored triangle

Adding more attributes

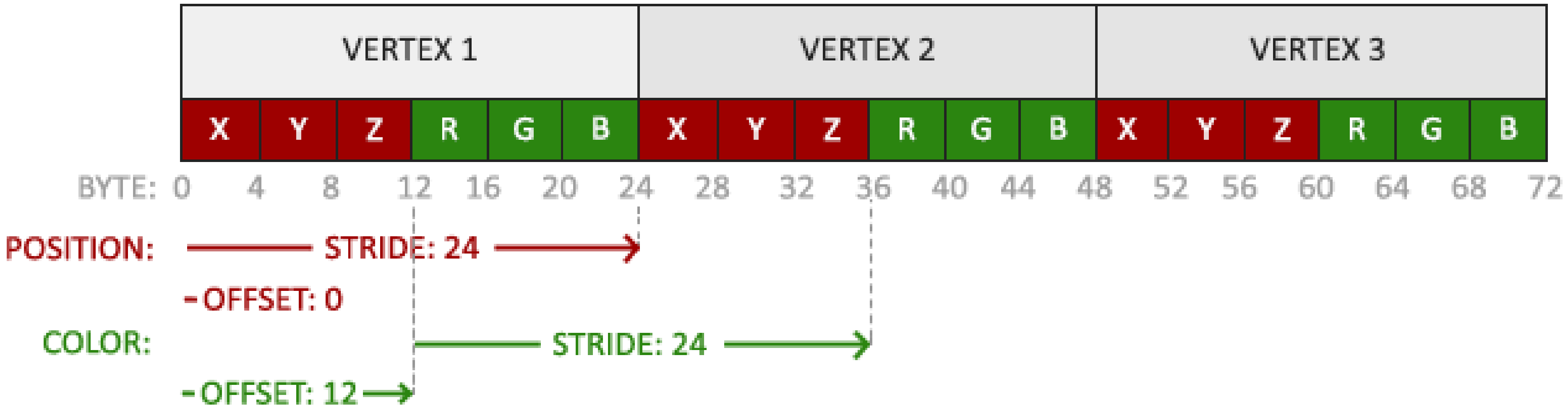
The fragment shader can be configured as follow:

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

Draw a multi-colored triangle

Adding more attributes

Since another vertex attribute was added thus also changing the VBO's memory, it is necessary to update the vertex attribute pointer. The updated data in the VBO's memory now looks a bit like this:



Draw a multi-colored triangle

Adding more attributes

Knowing the current layout, the vertex format can be updated with `glVertexAttribPointer`:

```
                                                                  stride            offset
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

                                                                  stride                      offset
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Differently than previous examples, the vertex attribute on attribute location 1 has to be configured. The color values have a size of 3 floats and normalization is not needed. Since there are two vertex attributes, it is necessary to re-calculate the stride and the offset values.

Draw a multi-colored triangle

Adding more attributes

Concerning the stride, to get the next attribute value (e.g., the next x component of the position vector) in the data array, a movement of 6 floats to the right is needed (three for the position values and three for the color values). This results in a stride value of 6 times the size of a float in bytes (= 24 bytes).

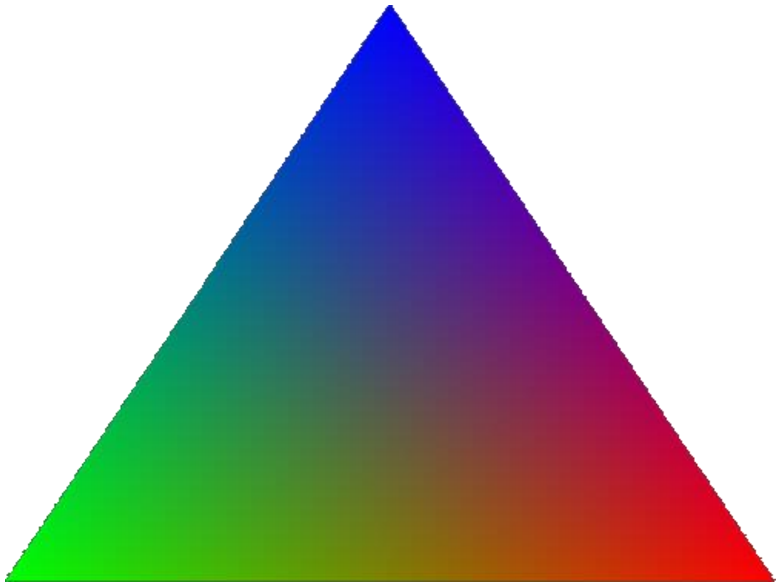
For what it concerns the offset values, for each vertex, the position vertex attribute is first (so the offset is 0). The color attribute starts after the position data so the offset is 3 * sizeof(float) in bytes (= 12 bytes).

Draw a multi-colored triangle

Fragment interpolation

The resulting image may differ from expectations since only 3 colors were supplied but the overall color palette is shown. This is all the result of **fragment interpolation** in the fragment shader.

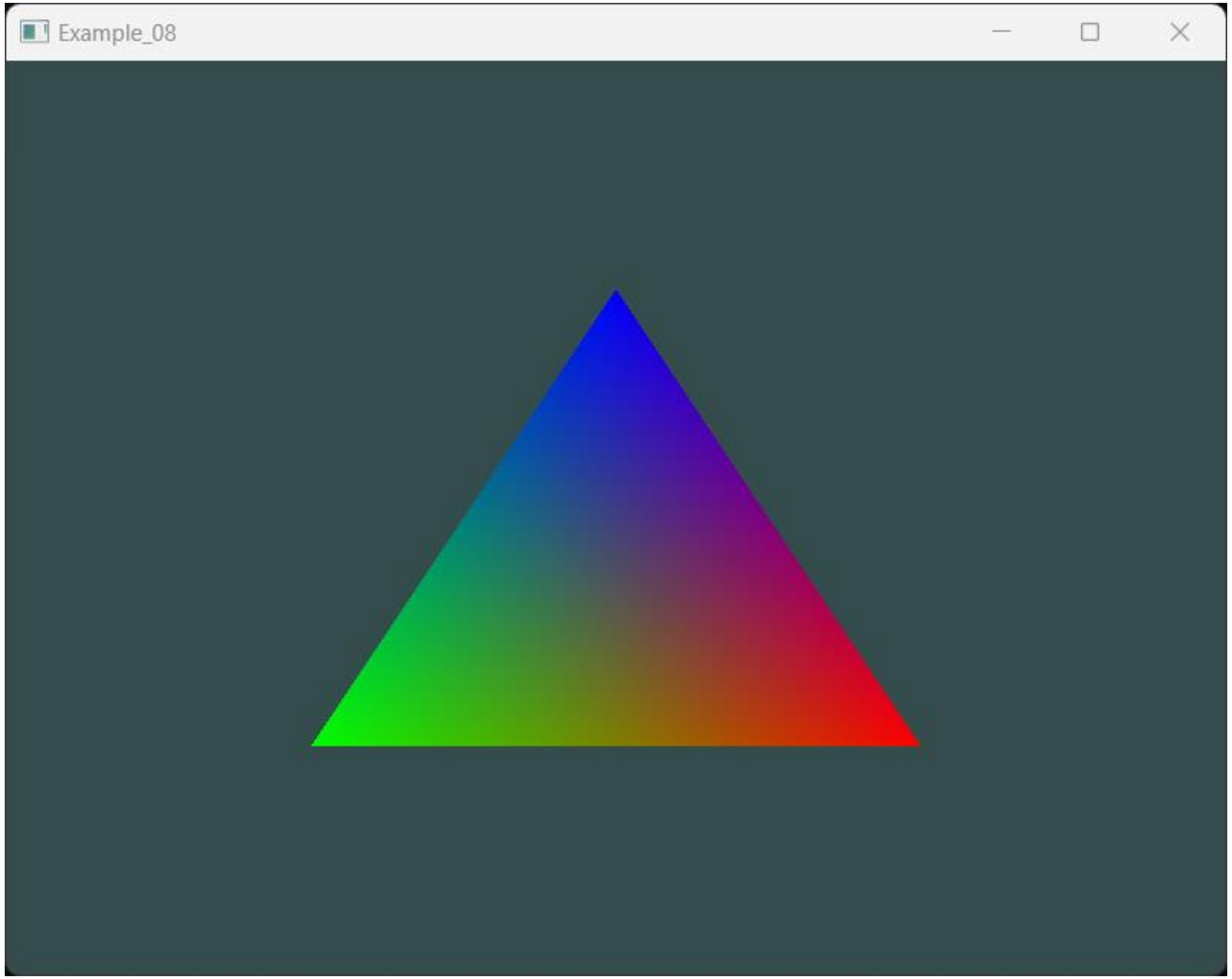
When rendering a triangle the rasterization stage usually results in a lot more fragments than vertices originally specified. The rasterizer then determines the positions of each of those fragments based on where they reside on the triangle shape. Based on these positions, it interpolates all the fragment shader's input variables.



A shader class

Example8

Writing, compiling and managing shaders can be quite cumbersome. Abstract objects (i.e., class) can be used to encapsulate some part of code regarding, e.g., the shaders.



A shader class

Creating the Shader class

The shader class can be entirely defined in a header file, mainly for learning purposes and portability. Let's start by adding the required includes.

```
#ifndef SHADER_H
#define SHADER_H

#include <glad/glad.h>

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader {...}

#endif
```

Preprocessor directives at the top of the header file are used to inform the compiler to only include and compile this header file if it hasn't been included yet, even if multiple files include the shader header. This prevents linking conflicts.

A shader class

Creating the Shader class

Then the class structure is defined:

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    Shader(const char* vertexPath, const char* fragmentPath)
    // use/activate the shader
    void use()
    // utility uniform functions
    void setBool(const std::string& name, bool value) const
    void setInt(const std::string& name, int value) const
    void setFloat(const std::string& name, float value) const
};
```

The shader class holds the ID of the shader program

A shader class

Creating the Shader class

Then the class structure is defined:

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    Shader(const char* vertexPath, const char* fragmentPath)
    // use/activate the shader
    void use()
    // utility uniform functions
    void setBool(const std::string& name, bool value) const
    void setInt(const std::string& name, int value) const
    void setFloat(const std::string& name, float value) const
};
```

The constructor requires the file paths of the source code of the vertex and fragment shader, respectively. Shader can be stored on disk as simple text files.

A shader class

Creating the Shader class

Then the class structure is defined:

```
class Shader
{
public:
    unsigned int ID;
    // constructor generates the shader on the fly
    Shader(const char* vertexPath, const char* fragmentPath)
    // use/activate the shader
    void use()
    // utility uniform functions
    void setBool(const std::string& name, bool value) const
    void setInt(const std::string& name, int value) const
    void setFloat(const std::string& name, float value) const
};
```

Some utility functions are introduced to ease the use of uniforms.

A shader class

Reading from file

The C++ `filestreams` are used to read the content of the shader from the file into several string objects:

```
Shader(const char* vertexPath, const char* fragmentPath)
{
	// 1. retrieve the vertex/fragment source code from filePath
	std::string vertexCode;
	std::string fragmentCode;
	std::ifstream vShaderFile;
	std::ifstream fShaderFile;
	// ensure ifstream objects can throw exceptions:
	vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
	fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
	try
	{
		// open files
		vShaderFile.open(vertexPath);
		fShaderFile.open(fragmentPath);
		std::stringstream vShaderStream, fShaderStream;
		// read file's buffer contents into streams
		vShaderStream << vShaderFile.rdbuf();
		fShaderStream << fShaderFile.rdbuf();
		// close file handlers
		vShaderFile.close();
		fShaderFile.close();
		// convert stream into string
		vertexCode = vShaderStream.str();
		fragmentCode = fShaderStream.str();
	}
	catch (std::ifstream::failure& e)
	{
		std::cout << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " << e.what() <<
std::endl;
	}
	const char* vShaderCode = vertexCode.c_str();
	const char* fShaderCode = fragmentCode.c_str();
	[...]
```

A shader class

Compile shaders

Next, shaders are compiled and linked. Note that compilation/linking errors are also reviewed and printed. This is extremely useful when debugging.

```
// 2. compile shaders
unsigned int vertex, fragment;
// vertex shader
vertex = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex, 1, &vShaderCode, NULL);
glCompileShader(vertex);
checkCompileErrors(vertex, "VERTEX");

// fragment Shader
fragment = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment, 1, &fShaderCode, NULL);
glCompileShader(fragment);
checkCompileErrors(fragment, "FRAGMENT");

// shader Program
ID = glCreateProgram();
glAttachShader(ID, vertex);
glAttachShader(ID, fragment);
glLinkProgram(ID);
checkCompileErrors(ID, "PROGRAM");

// delete shaders as they're linked into program now,
// hence they are no longer needed
glDeleteShader(vertex);
glDeleteShader(fragment);
```

A shader class

Compile shaders

The function `checkCompileErrors` can be defined as a private component:

```cpp
private:
    // checking shader compilation/linking errors.
    void checkCompileErrors(unsigned int shader, std::string type)
    {
        int success;
        char infoLog[1024];
        if (type != "PROGRAM")
        {
            glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
            if (!success)
            {
                glGetShaderInfoLog(shader, 1024, NULL, infoLog);
                std::cout << "ERROR::SHADER_COMPILATION_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
            }
        }
        else
        {
            glGetProgramiv(shader, GL_LINK_STATUS, &success);
            if (!success)
            {
                glGetProgramInfoLog(shader, 1024, NULL, infoLog);
                std::cout << "ERROR::PROGRAM_LINKING_ERROR of type: " << type << "\n" << infoLog << "\n -- --------------------------------------------------- -- " << std::endl;
            }
        }
    }
```

A shader class

Implement class functions

Then the use function is straightforward:

```
// activate the shader
void use()
{
    glUseProgram(ID);
}
```

A shader class

Define class functions

The setter can be implemented as follow:

```
// utility uniform functions
void setBool(const std::string& name, bool value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), (int)value);
}

void setInt(const std::string& name, int value) const
{
    glUniform1i(glGetUniformLocation(ID, name.c_str()), value);
}

void setFloat(const std::string& name, float value) const
{
    glUniform1f(glGetUniformLocation(ID, name.c_str()), value);
}
```

A shader class

Using the shader class

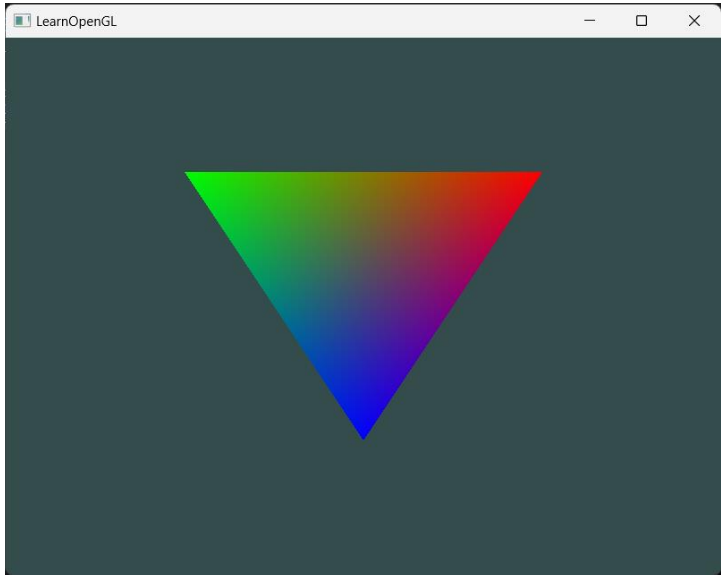
To use the Shader class, first the object has to be created and then used by calling its functions.

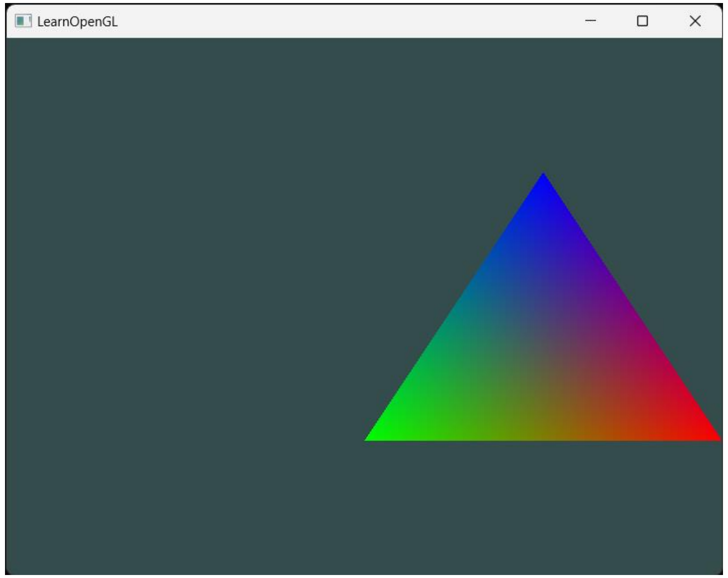
```
Shader ourShader("shader.vs", "shader.fs");

...

// render loop
while (!glfwWindowShouldClose(window))
{
    // draw the triangle
    ourShader.use();
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    ...
}
```

Exercise #3

- 3.1) Adjust the vertex shader so that the triangle is upside down.
  - Starting from Example#8 add a scaling factor to the vertex position
- 3.2) Specify a horizontal offset of 0.5 via a uniform and move the triangle to the right side of the screen in the vertex shader using this offset value
  - The uniform can be set in the rendering loos as follow:

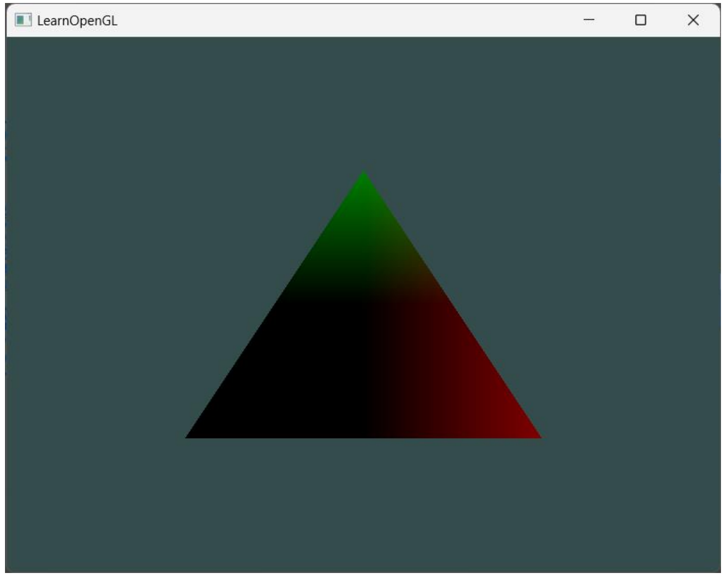
```
ourShader.use();
ourShader.setFloat("xOffset", offset);
```





Exercise #3

- 3.3) Output the vertex position to the fragment shader using the out keyword and set the fragment's color equal to this vertex position (see how even the vertex position values are interpolated across the triangle).
  - Why is the bottom-left side of the triangle black?



Texture

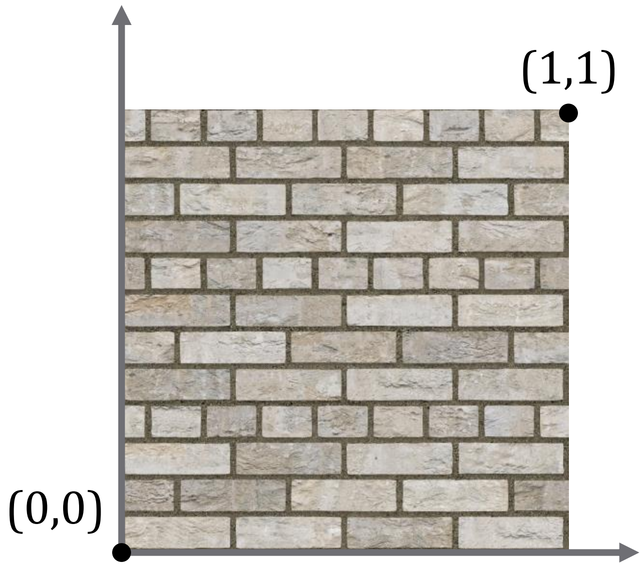
A texture is a 2D image used to add detail to an object. A texture can be seen as a piece of paper with a brick image (for example) on it, that can be folded over the 3D object to make it look like a wall of bricks.

Texture

To map a texture to the triangle, it is necessary to indicate for each vertex of the triangle which part of the texture it corresponds to.

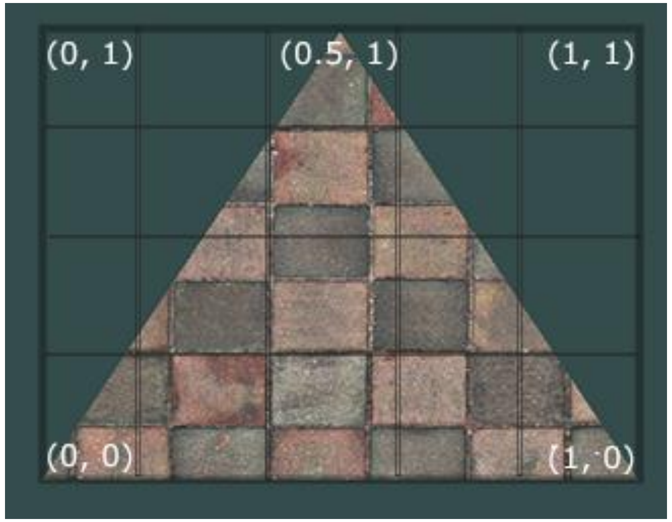
Each vertex should thus have a **texture coordinate** associated with them that specifies what part of the texture image to sample from. Fragment interpolation then does the rest for the other fragments.

Texture coordinates range from 0 to 1 in the x and y axis. 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.



Texture coordinates

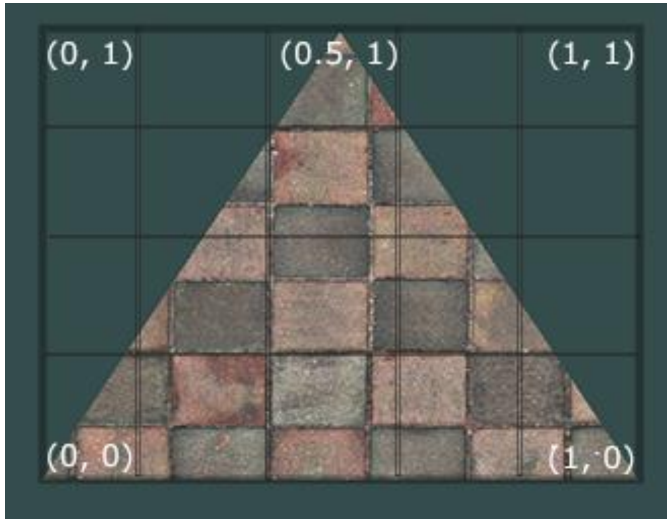
In the example of the triangle, 3 texture coordinate points have to be defined. The bottom-left side of the triangle has to correspond with the bottom-left side of the texture, so the (0,0) texture coordinate for the triangle's bottom-left vertex is used. The same applies to the bottom-right side with a (1,0) texture coordinate. The top of the triangle should correspond with the top-center of the texture image so (0.5,1.0) is taken as its texture coordinate.



Texture coordinates

The three texture coordinates have to be passed to the vertex shader, which then passes those to the fragment shader that interpolates all the texture coordinates for each fragment.

The resulting texture coordinates would then look like this:

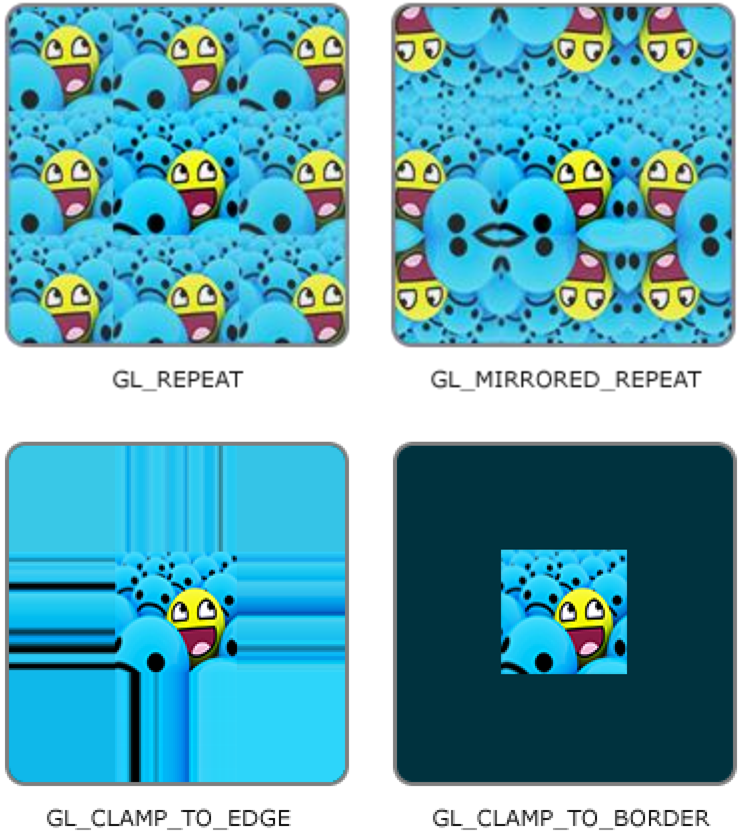



```
float textCoords[] = {
   0.0f, 0.0f, // lower-left corner
   1.0f, 0.0f, // lower-right corner
   0.5f, 1.0f, // top-center corner
};
```

Texture Wrapping

Texture coordinates usually range from (0,0) to (1,1) but what happens if coordinates outside this range are specified? The default behavior of OpenGL is to repeat the texture images, basically ignoring the integer part of the floating-point texture coordinate. Alternative options are:

- `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.



Texture Wrapping

Each options can be set per coordinate axis, i.e., `s`, `t` (and `r` if 3D textures are used), 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);
```

glTexParameter

This function set texture parameters

Usage: `void glTexParameteri(GLenum target, GLenum pname, GLint param);`

Parameters:

- `target`: Specifies the target texture, which must be either `GL_TEXTURE_2D`, `GL_TEXTURE_3D`, etc.
- `pname`: Specifies the symbolic name of a single-valued texture parameter. It can be one of the following: `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, `GL_TEXTURE_WRAP_R`, `GL_TEXTURE_MIN_FILTER`, or `GL_TEXTURE_MAG_FILTER`.
- `param`: Specifies the value of `pname`.

glTexParameter: https://docs.gl/es3/glTexParameter

Texture Wrapping

If the `GL_CLAMP_TO_BORDER` option is chosen, the border color should also be specified. This is done using the `fv` equivalent of the `glTexParameter` function with `GL_TEXTURE_BORDER_COLOR` as its option and passing in a float array the border's color value:

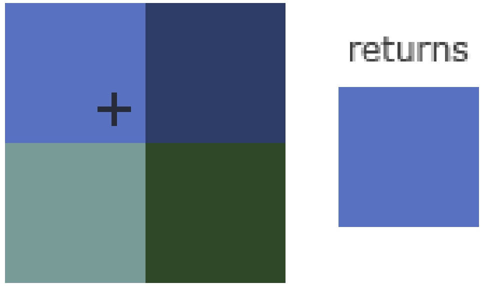
```
float borderColor[] = { 1.0f, 1.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

Texture Filtering

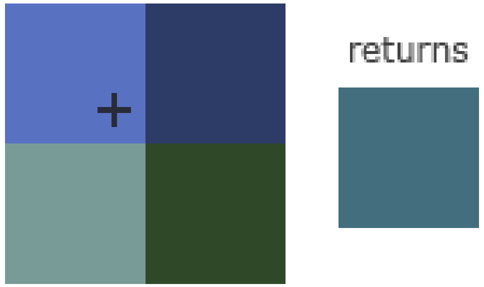
Texture coordinates do not depend on resolution but can be any floating-point value, thus OpenGL has to figure out which texture pixel (also known as a **texel**) to map the texture coordinate to. This becomes especially important in case of a very large object and a low-resolution texture. OpenGL has several options for this: texture filtering. The most important options are: `GL_NEAREST` and `GL_LINEAR`.

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.



`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.



Texture Filtering

GL_NEAREST

GL_LINEAR

GL_NEAREST results in blocked patterns where pixels that form the texture can be clearly seen, whereas 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.

Texture Filtering

Texture filtering can be set for magnifying and minifying operations (i.e., scaling up or downwards) so for example the nearest neighbor filtering can be used when textures are scaled downwards and linear filtering for upscaled textures. For this reason, it is necessary to specify the filtering method for both options via `glTexParameter*`.

The code should look similar to setting the wrapping method:

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

Mipmaps

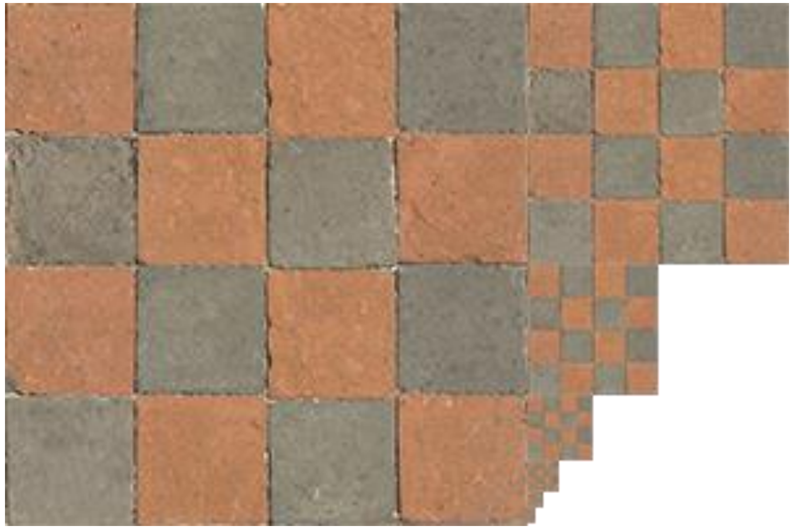
Imagine having thousands of objects, each with an attached texture. There will be objects far away that have the same high-resolution texture attached as the objects close to the viewer. Since the objects are far away and probably only produce a few fragments, OpenGL has difficulties retrieving the right color value for its fragment from the high-resolution texture, since it has to pick a texture color for a fragment that spans a large part of the texture. This will produce visible artifacts on small objects, not to mention the immoderation of memory bandwidth using high-resolution textures on small objects.

To solve this issue OpenGL uses the concept called mipmaps which is basically a collection of texture images where each subsequent texture is twice as small compared to the previous one.

Mipmaps

The idea behind mipmaps is that 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. OpenGL is then able to sample the correct texels, and there's less cache memory involved when sampling that part of the mipmaps.

A mipmapped texture looks like:



Mipmaps

Creating a collection of mipmapped textures for each texture image is cumbersome to do manually, but luckily OpenGL is able to do this work with a single call to `glGenerateMipmap` after creating a texture.

Like normal texture filtering, it is possible to filter between mipmap levels using `NEAREST` and `LINEAR` filtering for switching between mipmap levels. To specify the filtering method it is possible to use one of the following 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

Texture Filtering

The filtering method can be set by using the `glTexParameter*` function as follow:

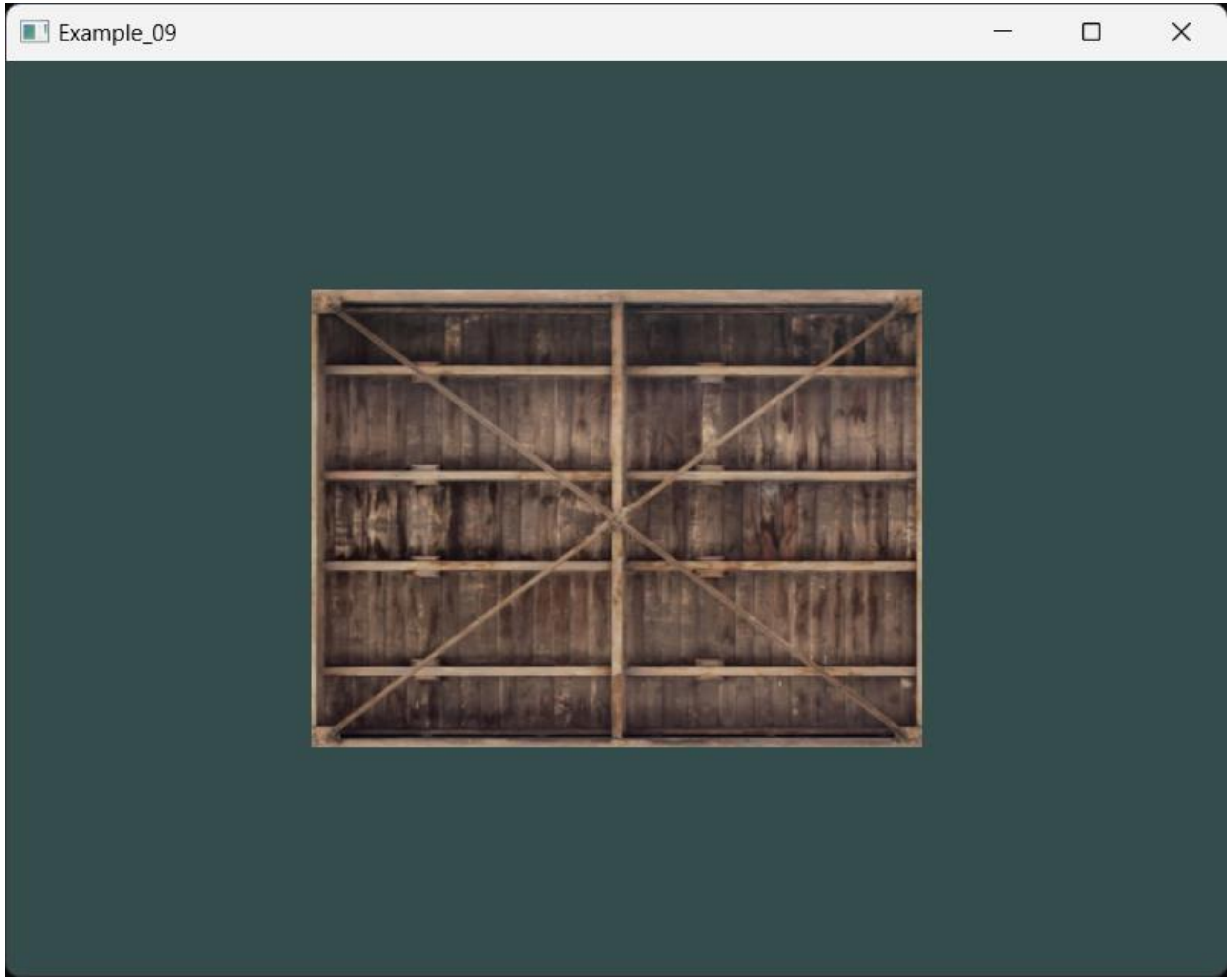
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);

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.

Draw a textured object

Example9

For the upcoming slides the rectangle shape drawn in Example4 will be used.



Draw a textured object

Loading textures

To use textures into an OpenGL application, it is necessary to load them. Several file format are available to store texture images. For this reason, instead of implementing an image loader for each format, it is better to use an image-loading library that supports several popular formats.

In this course, the `stb_image.h` library is used. This is a very popular single header image loading library that is able to load most popular file formats and is easy to integrate in the OpenGL project(s).

The library can be download from
https://github.com/nothings/stb/blob/master/stb_image.h

Draw a textured object

Install and use `stb_image.h`

To use the library, after downloading the single header file, add it to your project as `stb_image.h`. Then create an additional C++ file (i.e., stb_image.cpp) containing 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.

Now simply include `stb_image.h` in the program and compile.

```
#include "stb_image.h"
```

Draw a textured object

Loading textures

To load an image using `stb_image.h` it is possible to call the `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 three `ints` as its second, third and fourth argument. The three `ints` will be filled by the `stb_image.h` function with the resulting image's width, height and number of color channels. The data will be used later for generating the textures.

Draw a textured object

Generating a textures

Like any of the previous objects in OpenGL, textures are referenced with an ID. To create one use the following code:

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

The glGenTextures function first takes as input how many textures have to be generated, that are stored in a unsigned int array given as its second argument (in our case just a single unsigned int). Like other objects, it is needed to bind it so any subsequent texture commands will configure the currently bound texture:

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

Draw a textured object

Generating a textures

Once the texture is bound, it is possible to set, e.g., texture wrapping and filtering parameters:

```
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);// set texture wrapping to GL_REPEAT (default wrapping method)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Draw a textured object

Generating a textures

Now that the texture is bound and parameters are configured, it is possible to start generating a texture using the previously loaded image data. Textures are generated with `glTexImage2D`:

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

- 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 a texture has to be created manually, 0 means that is left at the base level.
- The third argument tells OpenGL in what kind of format the texture has to be stored. The sample image has only RGB values so the texture is stored with RGB values as well.

Draw a textured object

Generating a textures

Now that the texture is bound and parameters are configured, it is possible to start generating a texture using the previously loaded image data. Textures are generated with `glTexImage2D`:

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

- The 4th and 5th argument sets the width and height of the resulting texture. These values should have been stored earlier when loading the image, so the corresponding variables are used.
- The 6th argument should always be 0 (used for legacy stuff).
- The 7th and 8th argument specify the format and datatype of the source image. The image is loaded with RGB values and these values are stored as chars (bytes) so the corresponding values are passed.
- The last argument is the actual image data.

Draw a textured object

Generating a textures

Once `glTexImage2D` is called, the currently bound texture object has the texture image attached to it. However, currently it only has the base-level of the texture image loaded and if a new level of mipmaps has to be used it has to be specified. To specify all the different images manually it is possible to continually increment the second argument. In alternative the `glGenerateMipmap` function can be called after generating the texture. This will automatically generate all the required mipmaps for the currently bound texture.

```
glGenerateMipmap(GL_TEXTURE_2D);
```

After generating the texture and its corresponding mipmaps, it is a good practice to free the image memory:

```
stbi_image_free(data);
```

Draw a textured object

Applying textures

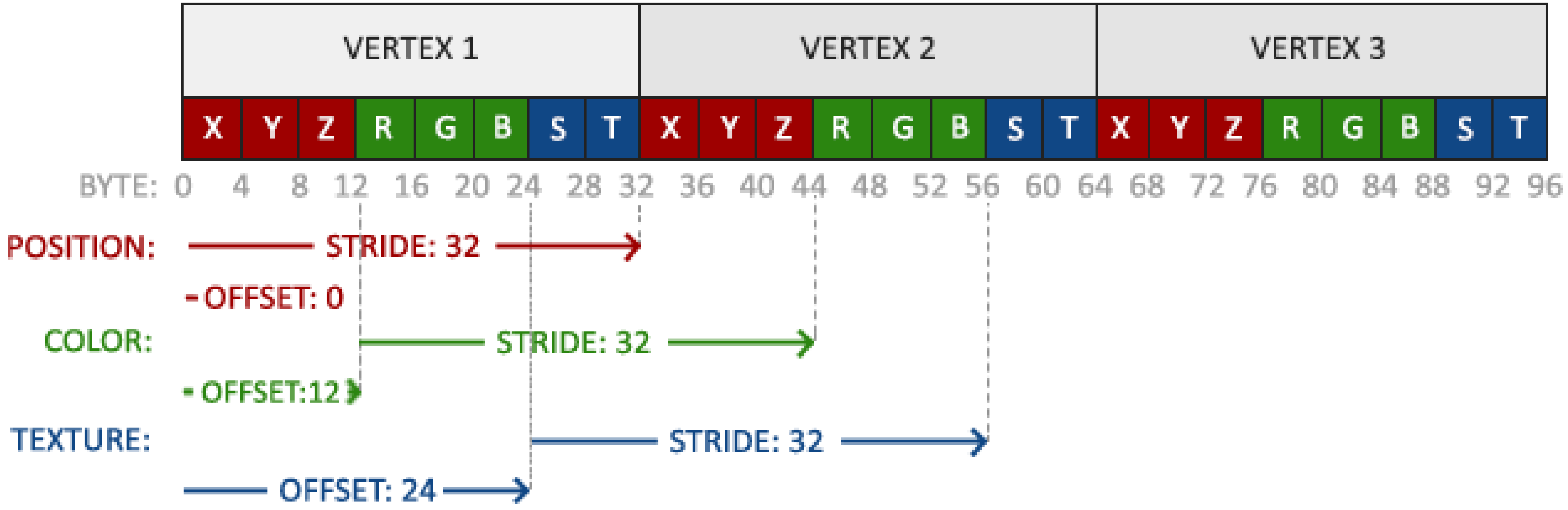
It is necessary to inform OpenGL how to sample the texture, so the vertex data have to be updated with the texture coordinates:

```
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
};
```

Draw a textured object

Applying textures

Since an extra vertex attribute has been added, it is necessary to notify OpenGL of the new vertex format:



Draw a textured object

Applying textures

Beside configuring the new texture coordinate attributes, the stride parameter of the previous two vertex attributes has to be updated according to the new format.

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3
* sizeof(float)));
glEnableVertexAttribArray(1);

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

Draw a textured object

Applying textures

The vertex shader is altered 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;
}
```

Draw a textured object

Applying textures

The fragment shader should then accept the `TexCoord` output variable as an input variable. The fragment shader should also have access to the texture object, but how the texture object is passed 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, e.g., sampler1D, sampler2D and sampler3D. To add a texture to the fragment shader it is simply declared a uniform sampler2D to which a texture is assigned later

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D ourTexture;
void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```

Draw a textured object

Applying textures

To sample the color of a texture the GLSL's built-in `texture` function can be used. It 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 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);
}
```

Draw a textured object

Applying textures

The last thing to do is in the render loop. It is necessary to bind the texture before calling `glDrawElements` and it will then automatically assign the texture to the fragment shader's sampler:

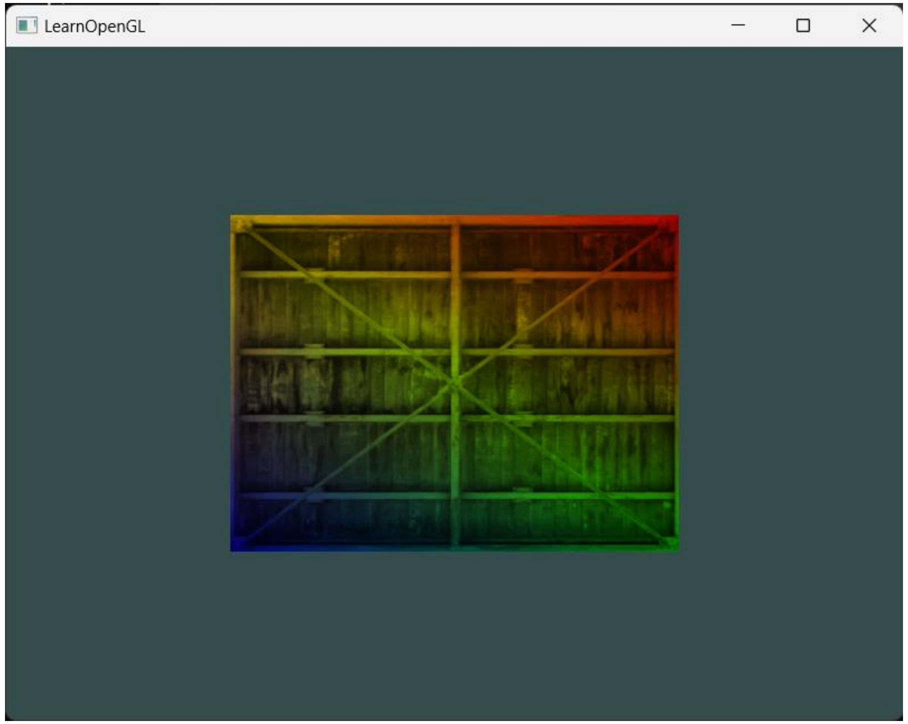
```
// bind Texture
glBindTexture(GL_TEXTURE_2D, texture);

// render container
ourShader.use();
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Draw a textured object

Mixing texture and colors

It is also possible to mix the resulting texture color with the vertex colors. This is achieved by multiply the resulting texture color with the vertex color in the fragment shader to mix both colors.

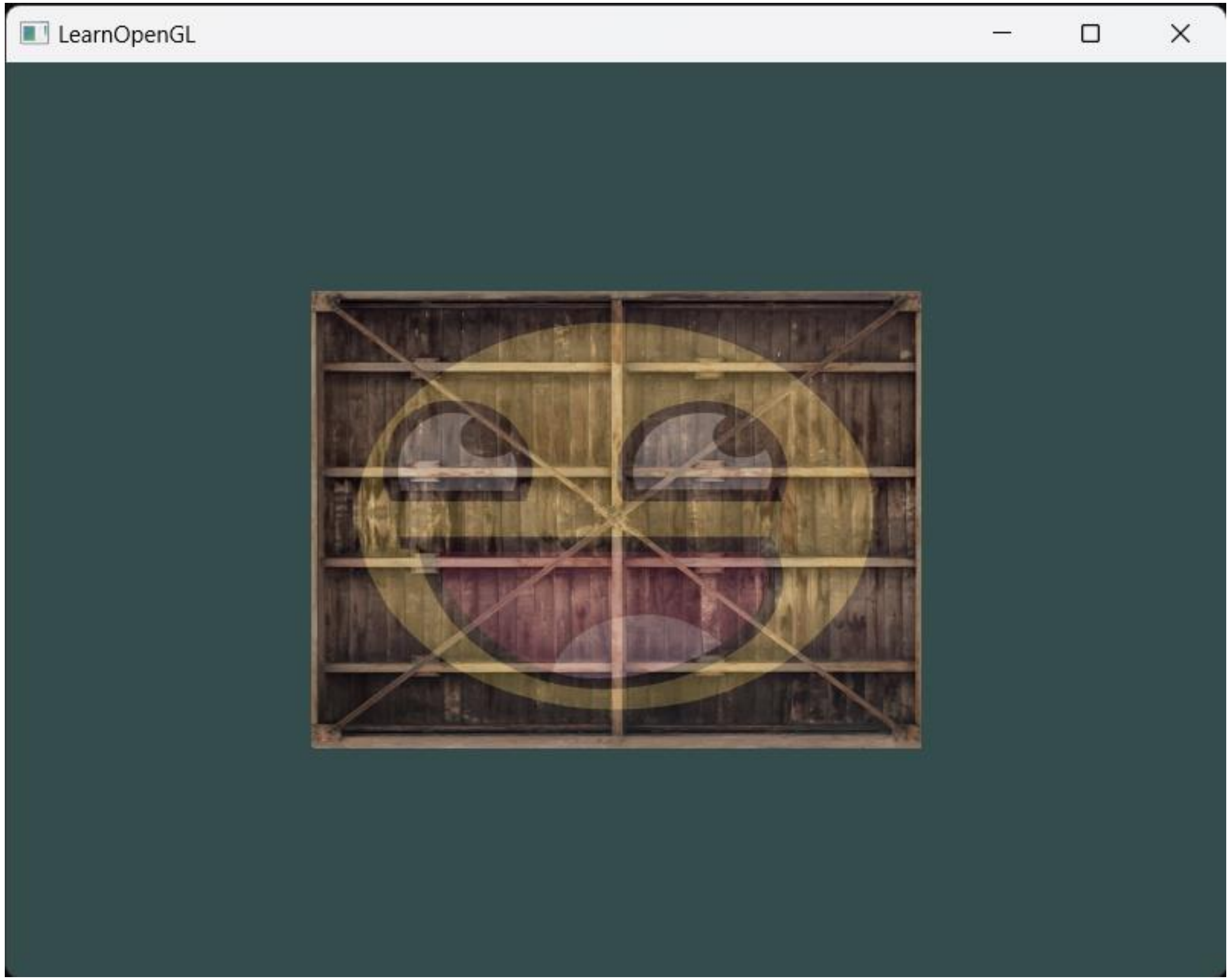
```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D texture1;

void main()
{
    FragColor = texture(texture1,
TexCoord) * vec4(ourColor, 1.0);
}
```



Draw a multi-textured object

Example10

It is possible to assign multiple textures to the same shader using the `glUniform` function



Draw a multi-textured object

Texture Unit

Using `glUniform1i` it is possible to actually assign a location value to the texture sampler so multiple textures can be set at once in a fragment shader. The location of a texture is more commonly known as a **texture unit**. The default texture unit for a texture is 0 which is the default active texture unit. For this reason, in the previous example it was not needed to assign a location to the texture.

The main purpose of texture units is to support more than 1 texture in a shader. By assigning texture units to the samplers, it is possible to bind to multiple textures at once as long as the corresponding texture unit first is activated.

Draw a multi-textured object

Texture Unit

Like `glBindTexture`, texture units can be activated using `glActiveTexture` and passing the texture unit to be used.

```
// bind textures on corresponding texture units
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);
```

After activating a texture unit, a subsequent `glBindTexture` call will bind that texture to the currently active texture unit. Texture unit `GL_TEXTURE0` is always by default activated, so it was not necessary to activate any texture units in the previous example when using `glBindTexture`.

Draw a multi-textured object

Texture Unit

OpenGL have at least a minimum of 16 texture units to be used that can be activated using `GL_TEXTURE0` to `GL_TEXTURE15`. They are defined in order so GL_TEXTURE8 can be reached via GL_TEXTURE0 + 8 for example. This is useful when there is the need to loop over several texture units.

Draw a multi-textured object

Texture Unit

The fragment shader is altered to accept another sampler:

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
uniform sampler2D texture2;
void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
}
```

The final output color is the combination of two texture lookups. GLSL's built-in `mix` function takes two values as input and linearly interpolates between them based on its third argument. If the third value is 0.0 it returns the first input; if it's 1.0 it returns the second input value. A value of 0.2 will return 80% of the first input color and 20% of the second input color, resulting in a mixture of both our textures.

Draw a multi-textured object

Texture Unit

It is necessary to load and create another texture, by using the `glTexImage2D`.

```
unsigned char* data = stbi_load("awesomeface.png", &width, &height,
&nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
```

Note that the provide .png image includes an alpha (transparency) channel. This means that it is necessary to specify that the image data contains an alpha channel as well by using GL_RGBA; otherwise OpenGL will incorrectly interpret the image data.

Draw a multi-textured object

Texture Unit

To use the second texture (and the first texture) the rendering procedure has to be changed a bit by binding both textures to the corresponding texture unit:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, texture2);

ourShader.use();
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

Draw a multi-textured object

Texture Unit

Finally, it is necessary to tell OpenGL to which texture unit each shader sampler belongs to by setting each sampler using `glUniform1i`. This has to be set once, so it can be done before entering in the render loop. Rember to activate the shader before setting uniforms. The uniform can be set manually or by using the shader class.

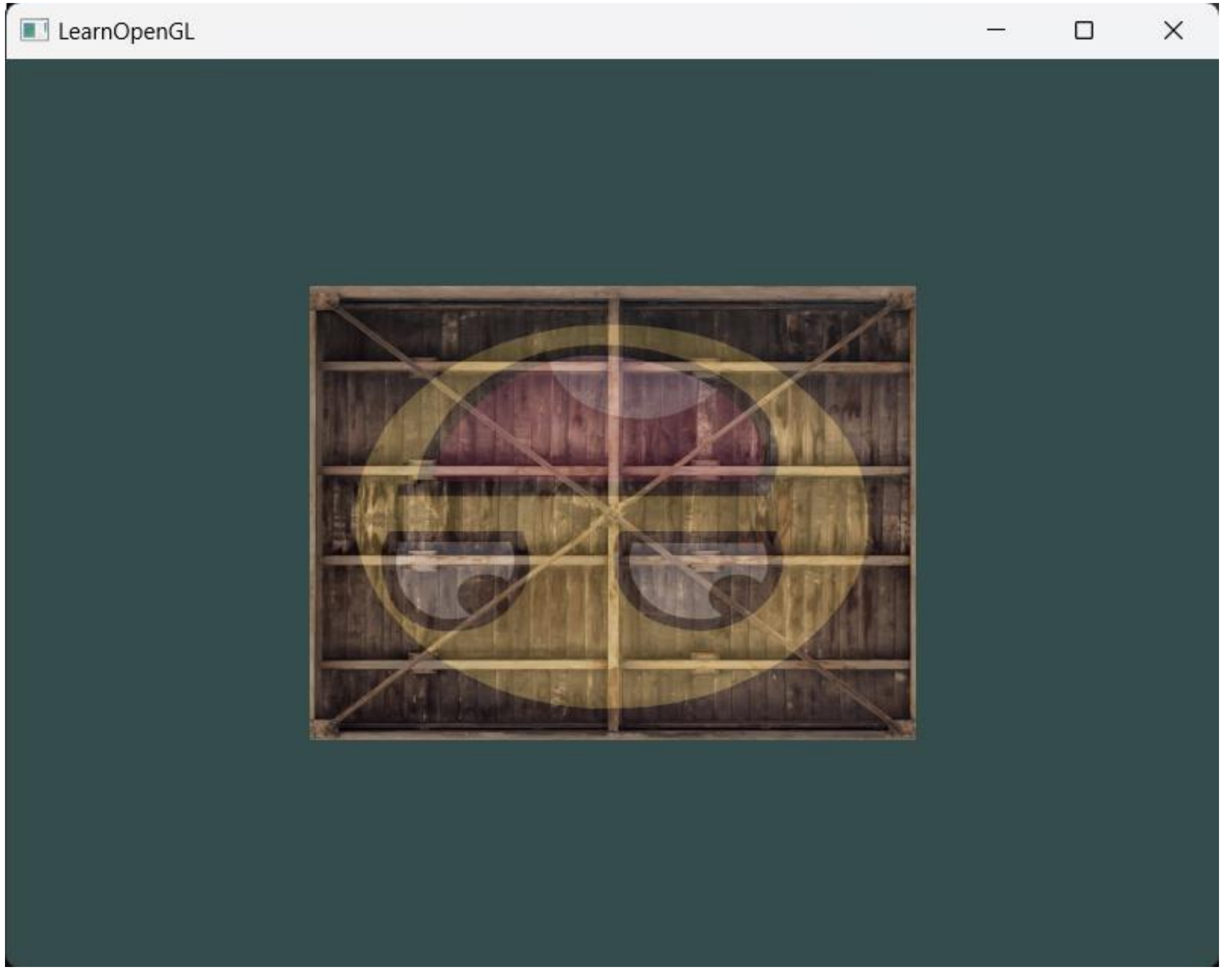
```
ourShader.use();
// set it manually:
glUniform1i(glGetUniformLocation(ourShader.ID, "texture1"), 0);
// set it via the shader class
ourShader.setInt("texture2", 1);
```

Draw a multi-textured object

Flipped texture

The result shown in the figure should be get when running the code. It can ne noticed that the smile texture is flipped upside-down.

This happens because OpenGL expects the 0.0 coordinate on the y-axis to be on the bottom side of the image, but images usually have 0.0 at the top of the y-axis.



Draw a multi-textured object

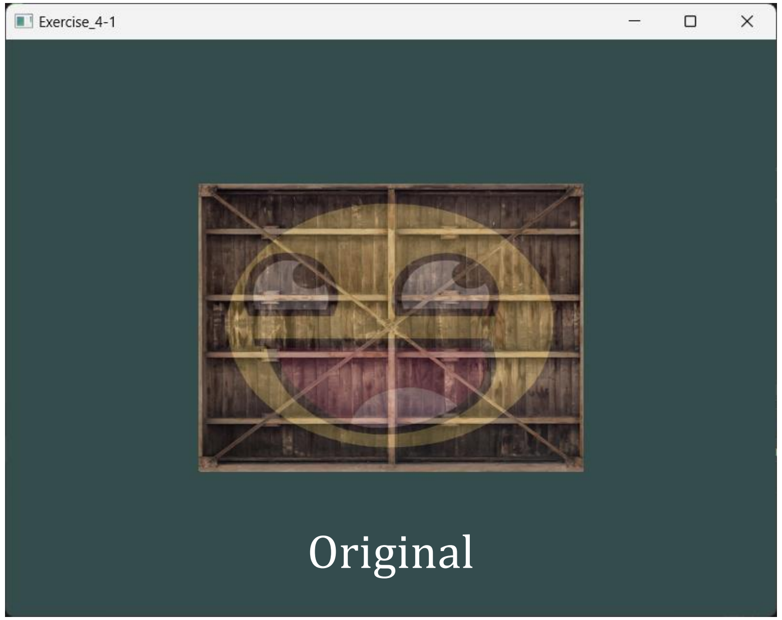
Flipped texture

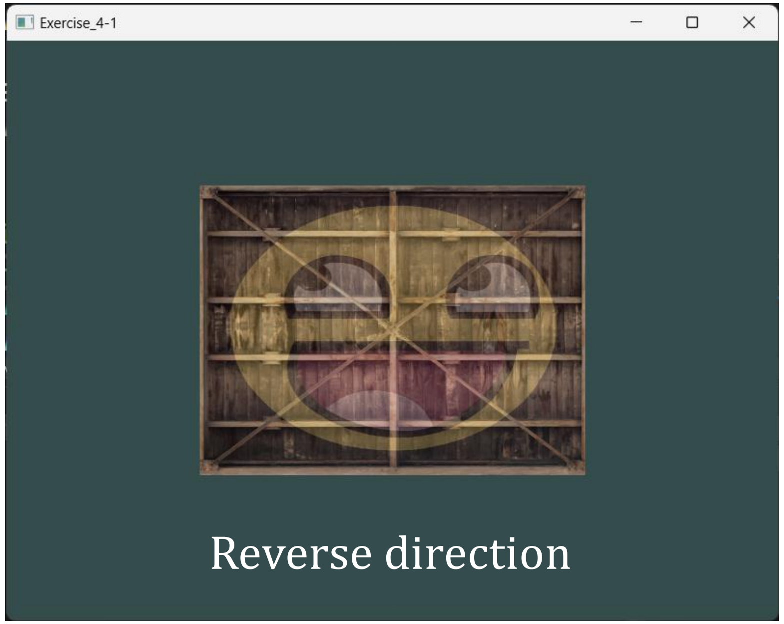
To cope with this issue, it is possible to use a dedicated function of the `stb_image.h` library, that flip the y-axis during image loading:

```
stbi_set_flip_vertically_on_load(true);
```

Exercise #4

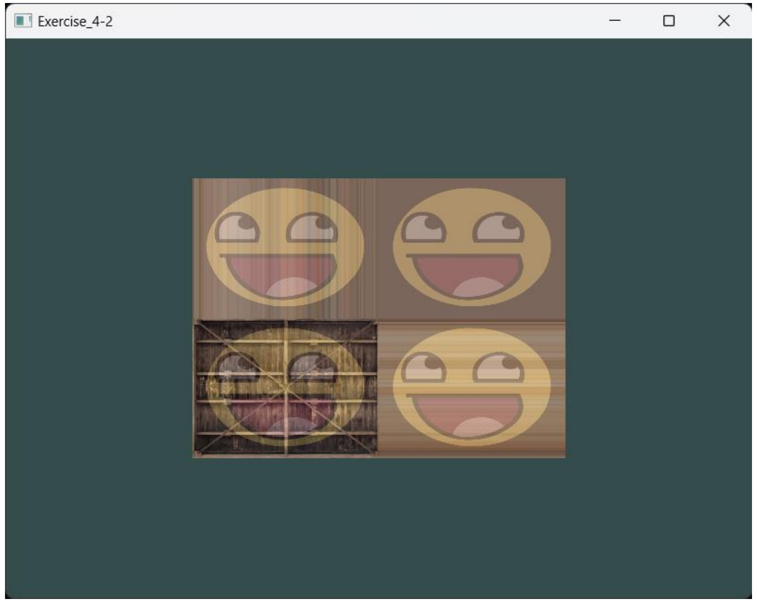
- 4.1) Make sure only the happy face looks in the other/reverse (flipping the x direction only) by changing the fragment shader
  - Starting from Example#10 and apply changes to the fragment shader





Exercise #4

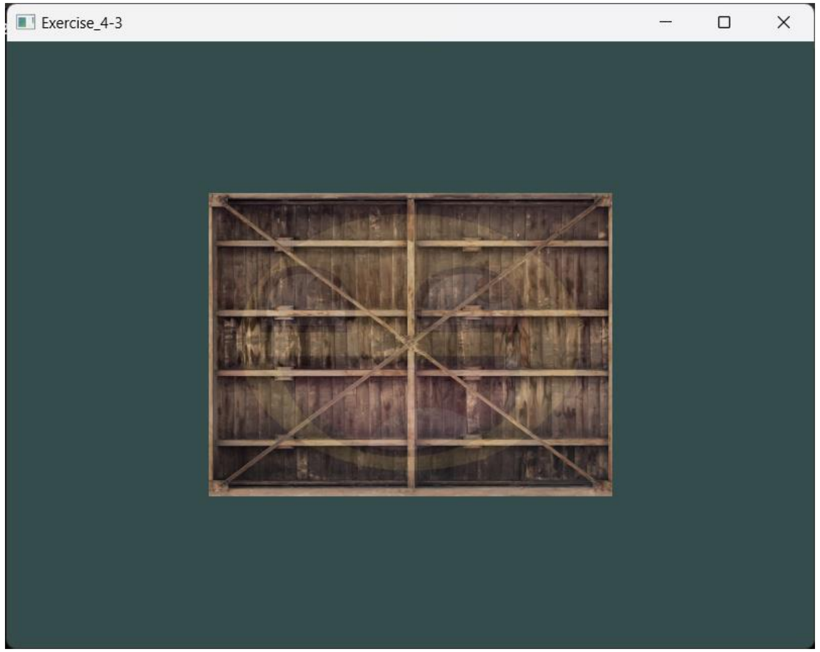
- 4.2) Experiment with the different texture wrapping methods by specifying texture coordinates in the range 0.0f to 2.0f instead of 0.0f to 1.0f. See if you can display 4 smiley faces on a single container image clamped at its edge
  - Configure the two different behaviors of the textures using the `glTexParameteri()` for each texture.

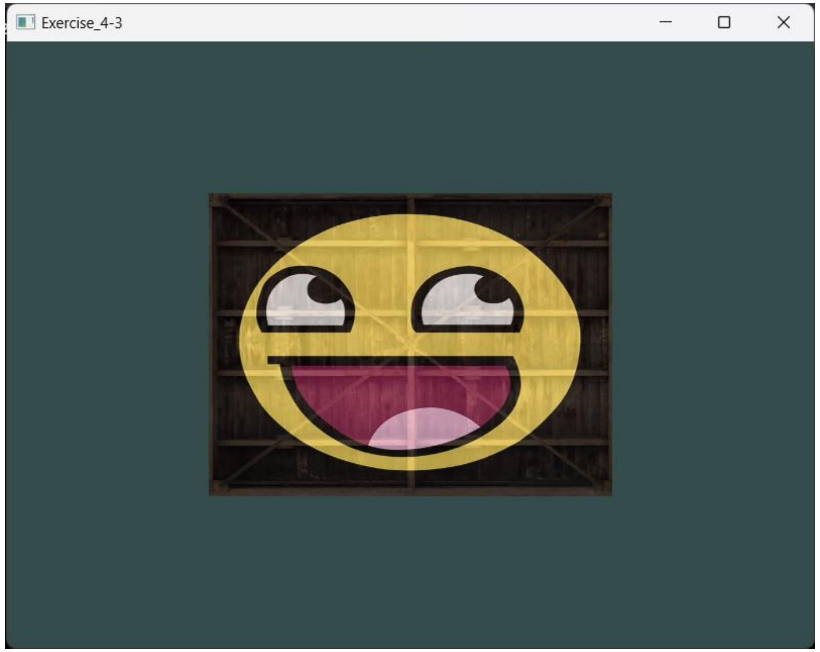



```
float vertices[] = {
    // positions          // colors           // texture coords
     0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   2.0f, 2.0f, // top right
     0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   2.0f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f, // bottom left
    -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 2.0f  // top left
};
```

Exercise #4

- 4.3) Use a uniform variable as the mix function's third parameter to vary the amount the two textures are visible. Use the up (`GLFW_KEY_UP`) and down (`GLFW_KEY_DOWN`) arrow keys to change how much the container or the smiley face is visible





Exercise #4

- Submit the three exercises by uploading a folder (for each exercise) with the following files:
  - Main.cpp
  - Vertex shader
  - Fragment shader