

Notes
CSC-271
Intro to Computer Graphics

Dr. Qixin Deng

September 2025

Contents

1	Introduction	3
2	Environment Setup	4
2.1	GLFW and GLAD	4
2.1.1	CMake Configuration	4
2.1.2	Test code	5
2.1.3	Common Compilation Errors	9
3	Modern Graphics Pipeline	10
4	Vertex Processing	11
4.1	Data Flow in OpenGL Pipeline	11
4.1.1	Vertex Buffer Object	11
4.1.2	Vertex Shader	12
4.1.3	Fragment Shader	13
4.1.4	Vertex Array Buffer Object	13
4.1.5	ShaderProgram Class	14
4.1.6	Element Buffer Object	19
4.2	GLSL	21
4.2.1	Vector	21
4.2.2	Ins & Outs	22
4.2.3	layout	23
4.2.4	Uniforms	23
4.2.5	More Attributes	25
4.3	Textures	27
4.3.1	Texture Coordinates	27
4.3.2	Texture Wrapping	27
4.3.3	Texture Filtering	28
4.3.4	Mipmaps	29
4.3.5	Loading And Creating Textures	30
4.3.6	Texture Units	33
4.4	Mesh Class	37
4.4.1	Mesh.h	37
4.4.2	Mesh.cpp	38
4.4.3	Rewrite Main Function	40
5	Transformations	41
5.1	Vector Basics	41
5.1.1	Vector-Vector Multiplication	43
5.2	Matrices	44
5.2.1	Matrix-Vector Multiplication	45
5.2.2	Combining Matrices	46
5.3	GLM	47
5.4	Coordinate Systems	49
5.4.1	From Local Space to World Space	50
5.4.2	From World Space to View Space	51

5.4.3	From View Space to Clip Space	52
5.4.4	From Clip Space to Normalize Device Space	52
5.4.5	Putting It All Together	53
5.4.6	An Example	53
5.5	Going 3D	55
5.5.1	Tinyobjloader Library	55
5.5.2	Z-buffer	58
5.6	Cameras	60
5.6.1	Camera Position	60
5.6.2	Camera Direction	60
5.6.3	Right Axis	60
5.6.4	Up axis	61
5.6.5	Why do we recompute the Up vector?	61
5.6.6	LookAt Matrix	62
5.6.7	Move Around	62
5.6.8	Look Around	64
5.6.9	Zoom	67
5.6.10	Camera Class	67
6	Basic Lighting	71
6.1	A lighting Scene	71
6.2	Basic Lighting Concepts	73
6.2.1	Ambient Lighting	73
6.2.2	Diffuse Lighting	73
6.2.3	Specular Lighting	78
6.3	Materials	80
6.4	Lighting Maps	85
6.4.1	Diffuse Map	85
6.4.2	Specular Map	87
6.5	Light Casters	88
6.5.1	Directional Light	88
6.5.2	Point Light	90
6.5.3	Spot Light	94
7	Advanced OpenGL	104
7.1	Depth Testing	104
7.1.1	z-fighting	106
7.2	Stencil Testing	107
7.2.1	Stencil Functions	108
7.2.2	Application: Object Outlining	108
7.3	Blending	110
7.4	Cubemaps	114
7.4.1	Skybox	114
7.4.2	Environment Mapping: Reflection	119
7.4.3	Environment Mapping: Refraction	121
7.5	Shadows	122
7.5.1	Preparation	122
7.5.2	Intro to Directional Light Shadow Mapping	123
7.5.3	Intro to Point Light Shadow Mapping	133

1 Introduction

Computer graphics is the field of generating, manipulating, and displaying visual content with computers. It spans everything from 2D charts and UI icons to complex 3D worlds in games and simulations. At its core, computer graphics turns abstract data (numbers describing positions, colors, lights, materials) into images. Two big branches are offline rendering (high-quality images for films and design, where render time can be minutes per frame) and real-time rendering (interactive graphics for games, AR/VR, and visualization, where you need 30–144+ frames every second).

To draw pictures efficiently, modern systems use a graphics pipeline. In simplified form: you define geometry as vertices, the GPU transforms those vertices into camera space, clips what’s off-screen, rasterizes triangles into pixels, then shades each pixel to determine its final color. Programmers can customize key stages with small programs called shaders, which control how vertices move and how pixels are lit, textured, and colored.

OpenGL is a cross-platform, cross-language graphics API specification maintained by the Khronos Group. It defines what functions and behavior a conforming driver must provide so your code can talk to the GPU in a portable way. Vendors (NVIDIA, AMD, Intel, Apple) implement OpenGL in their drivers, and your program calls those functions to issue drawing commands. Because it’s a spec, OpenGL itself doesn’t create windows or handle input—that’s why we pair it with libraries like GLFW (window + input) and a loader like GLAD (to access modern GL function pointers).

Modern OpenGL (core profile 3.x/4.x) emphasizes a programmable pipeline. You provide data via buffer objects and vertex array objects (VBOs/VAOs), write GLSL shaders (vertex, fragment, and optionally geometry/tessellation/-compute), and control state like depth testing and blending. This is different from the older “fixed-function” style that is now deprecated—it gives you much more flexibility and performance, but also expects you to be explicit about resources and shader code.

There are related APIs worth knowing: OpenGL ES (a streamlined version used on mobile and embedded devices), WebGL (OpenGL ES for the web), and newer low-level APIs like Vulkan, Direct3D 12, and Metal that expose more explicit control over the GPU. For learning and many desktop projects, OpenGL remains a solid, well-documented choice with a huge ecosystem of tutorials, tools, and examples.

2 Environment Setup

For this course project, we will use **CLion** as our Integrated Development Environment (IDE). CLion is a powerful cross-platform C/C++ IDE developed by JetBrains, designed to streamline development with intelligent code assistance, debugging tools, and seamless integration with build systems. One of its key features is native support for **CMake**, which is used to configure and manage the build process of our project. CMake allows for flexible and portable project setups, making it ideal for cross-platform development. CLion also supports modern C++ standards, integrates with version control systems, and provides a user-friendly interface for managing complex projects. More information and downloads are available at the official website: <https://www.jetbrains.com/clion/>.

2.1 GLFW and GLAD

GLFW (Graphics Library Framework) is an open-source, multi-platform library designed for creating windows with OpenGL contexts and managing input and events. It provides a simple API for handling window creation, keyboard, mouse, and joystick input, as well as context management and timing. GLFW is widely used in graphics programming and game development due to its lightweight design and ease of integration with OpenGL and other rendering libraries. It is particularly well-suited for educational projects, prototyping, and applications that require direct control over the rendering pipeline. Download its pre-built versions from the official website: <https://www.glfw.org/download.html>. Unzip the file in the directory of the project.

GLAD (GL Loader Generator) is a multi-language generator for managing OpenGL function pointers. Since OpenGL functions are not directly accessible without a loader, GLAD simplifies this process by generating the necessary code to load and use OpenGL functions dynamically at runtime. It supports multiple specifications and profiles, including OpenGL and Vulkan, and is compatible with various platforms and languages. GLAD is commonly used alongside libraries like GLFW to create cross-platform graphics applications, ensuring that developers can access the full range of OpenGL features in a clean and efficient way.

Go to the GLAD <https://glad.dav1d.de/> and setup the configurations:

- the language is set to **C++**
- in the API section select an OpenGL version of at least **3.3**
- profile is set to **Core**
- the **Generate a loader** option is ticked.

Click **Generate** to produce and download the resulting library files. Unzip the file in the directory of the project.

This section outlines the steps taken to integrate the GLFW/GLAD library into a CLion C++ project using CMake on macOS (ARM64 architecture). It includes configuration details, common errors, and their resolutions. Before we start, you need all three items to be ready:

- GLFW version: 3.4 (download suitable version for your OS)
- GLAD generated by the configurations listed above
- Create a C++17 project in Clion.

2.1.1 CMake Configuration

The `CMakeLists.txt` file included the following:

```
1  # Specify the minimum required version of CMake to process this file
2  cmake_minimum_required(VERSION 3.27)
3
4  # Declare the name of the project. This name is used to identify the build target.
5  project(CGProject)
6
7  # Set the C++ standard to C++17 for compiling the project.
8  set(CMAKE_CXX_STANDARD 17)
9
10 # Add the GLFW and GLAD include directory to the compiler's search path.
11 include_directories(${CMAKE_SOURCE_DIR}/glfw-3.4.bin.MACOS/include
12                    ${CMAKE_SOURCE_DIR}/glad/include)
13
14 # Add the directory containing the compiled GLFW library to the linker's search path.
15 link_directories(${CMAKE_SOURCE_DIR}/glfw-3.4.bin.MACOS/lib-arm64)
16
17 # Add GLAD source file
```

```

18 add_library(glad ${CMAKE_SOURCE_DIR}/glad/src/glad.c)
19
20 # Define the executable target named CGProject, built from main.cpp.
21 add_executable(CGProject main.cpp)
22
23 # Link the GLFW dynamic library (.dylib) directly using its full path.
24 # Link GLAD library defined in line 18
25 # Also link the macOS OpenGL framework, which is required for rendering.
26 target_link_libraries(CGProject
27     glad
28     libglfw.3.dylib
29     "-framework OpenGL"
30 )

```

CMAKE_SOURCE_DIR is a built-in CMake variable that automatically resolves to the absolute path of the directory containing the top-level CMakeLists.txt. It does not need to be manually set. Here it represents the absolute directory of your project.

2.1.2 Test code

Now, let's add code to the main.cpp file to verify that our GLFW/GLAD setup is working correctly.

```

1 #include <glad/glad.h>
2 #include <GLFW/glfw3.h>
3 #include <iostream>

```

- `#include <glad/glad.h>`, pulls in GLAD, the OpenGL function loader that gives your app access to modern OpenGL calls at runtime. **It must be included before GLFW.**
- `#include <GLFW/glfw3.h>`, brings in GLFW, which creates the window and OpenGL context and handles input (keyboard/mouse).
- `#include <iostream>`, enables basic C++ I/O (e.g., `std::cout`) for printing messages or errors to the console.

```

1 int main() {
2     if (!glfwInit()) {
3         std::cerr << "Failed to initialize GLFW\n";
4         return -1;
5     }
6
7     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
8     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
9     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
10
11     #ifdef __APPLE__
12         glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
13     #endif
14
15     glfwTerminate();
16     return 0;
17 }

```

`int glfwInit(void)` initializes the GLFW library. Before most GLFW functions can be used, **GLFW must be initialized**, and before an application terminates GLFW should be terminated in order to free any resources allocated during or after initialization. If this function fails, it calls `glfwTerminate` before returning automatically. **If it succeeds, you should call `glfwTerminate` before the application exits.**

This code sets GLFW to create a modern OpenGL 4.1 core context. `glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4)` and `...MINOR, 1` request OpenGL 4.1 specifically; `GLFW_OPENGL_CORE_PROFILE` selects the core profile, which excludes legacy, fixed-function features so you use the programmable pipeline (shaders, buffers, etc.). The `#ifdef __APPLE__` block sets `GLFW_OPENGL_FORWARD_COMPAT` on macOS, where Apple requires a forward-compatible core context for 3.2+—removing deprecated functionality and ensuring the context is created successfully. Together, these hints give you a consistent, modern GL environment across platforms (assuming the GPU/driver supports 4.1).

```

1   GLFWwindow *window = glfwCreateWindow(800, 600, "Hello CG", nullptr, nullptr);
2   if (!window) {
3       std::cerr << "Failed to create GLFW window\n";
4       glfwTerminate();
5       return -1;
6   }
7   glfwMakeContextCurrent(window);

```

In C++, a namespace is a way to group related code (like functions, classes, and variables) under a named scope to avoid name conflicts and organize code logically. for example, functions defined in Standard Library: `std::vector`, `std::cout`

After GLFW successful initialization, we can now create a window. `GLFWwindow *glfwCreateWindow(int width, int height, const char *title, GLFWmonitor *monitor, GLFWwindow *share)` creates a window and its associated context.

- **width:** The desired width, in screen coordinates, of the window. This must be greater than zero.
- **height:** The desired height, in screen coordinates, of the window. This must be greater than zero.
- **title:** The initial, UTF-8 encoded window title.
- **monitor:** The monitor to use for full screen mode, or NULL for windowed mode.
- **share:** The window whose context to share resources with, or NULL to not share resources.

Returns the handle of the created window, or 'NULL' if an error.

Successful creation does not change which context is current. **Before you can use the newly created context, you need to make it current.** `void glfwMakeContextCurrent(GLFWwindow *window)` makes the context of the specified window current for the calling thread. This function makes the OpenGL or OpenGL ES (OpenGL for Embedded Systems) context of the specified window current on the calling thread. It can also detach the current context from the calling thread without making a new one current by passing in 'NULL'. A context must only be made current on a single thread at a time and each thread can have only a single current context at a time. Making a context current detaches any previously current context on the calling thread.

```

1   glViewport(0, 0, 800, 600);

```

`glViewport` tells OpenGL where to draw on the window. It defines a rectangular region in pixel coordinates. The first two parameters (0, 0) represent the bottom-left corner of the viewport. `width` represents the width of the rendering area. `height` represents the height of the rendering area.

Why is it necessary? We already defined the window size when we create the window, what the difference between viewport and a window?

`glViewport` represents how we want to display the data and coordinates with respect to the window. You can set `glViewport` to a rectangle smaller than the GLFW window, so OpenGL draws only inside that subregion and you can use the remaining area for other UI or overlays. Internally, `glViewport` defines how OpenGL maps normalized device coordinates (NDC) (-1..1) to screen pixels: for a viewport at (0,0) with size 800*600, a point at (-0.5, 0.5) lands at pixel (200, 450), i.e., values in (-1..1) are mapped to (0..800) and (0..600).

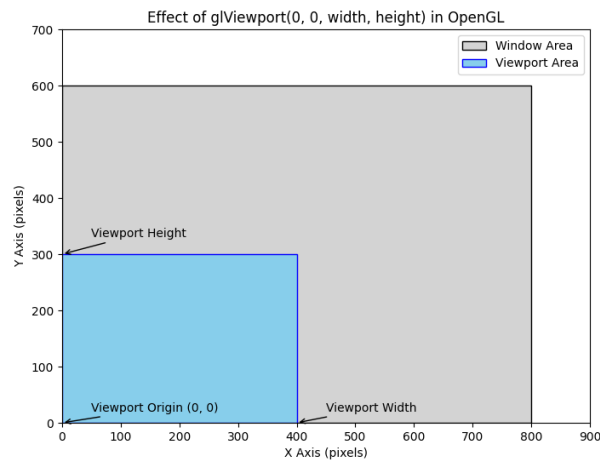


Figure 1: The gray rectangle represents the full window (e.g., 800×600 pixels). The blue rectangle is the viewport defined by `glViewport(0, 0, 400, 300)`. Arrows and labels show how the viewport starts at the origin and extends to the specified width and height. If you change width and height, the blue area will grow or shrink accordingly, affecting where and how OpenGL renders content.

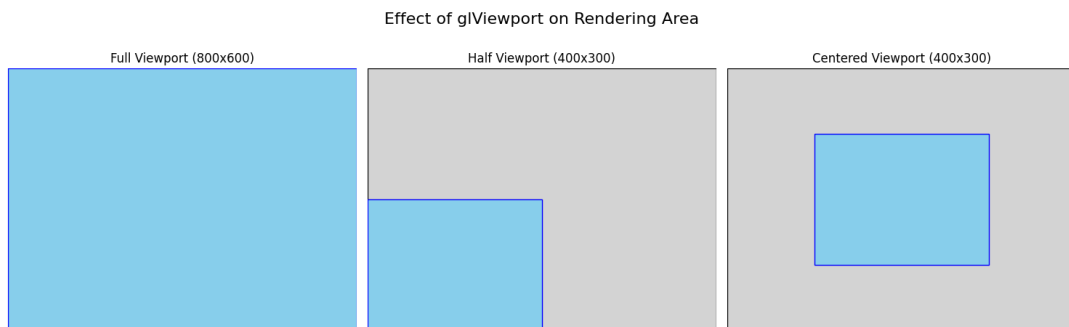


Figure 2: Left (Full Viewport): `glViewport(0, 0, 800, 600)` — the entire window is used for rendering. This is the default and most common setup. Middle (Half Viewport): `glViewport(0, 0, 400, 300)` — only the bottom-left quarter of the window is used. Rendering will be scaled to fit this smaller area. Right (Centered Viewport): `glViewport(200, 150, 400, 300)` — the viewport is centered in the window. This is useful for rendering to a specific region, like a minimap or split-screen.

```

1  void framebuffer_size_callback(GLFWwindow* window, int width, int height)
2  {
3      glViewport(0, 0, width, height);
4      std::cout<< width << " " << height << std::endl;
5  }
6
7  int main{
8      ... ..
9      glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
10     ... ..
11 }

```

Because we are using window mode now, we want the viewport can be updated automatically when the user resize the GLFW window. We can achieve this by defining a callback function (A callback function is a function passed as an argument to another function, which is then invoked ("called back") by the outer function when a specific event occurs or a task is completed.) The way we define `framebuffer_size_callback` function is decided by `glfwSetFramebufferSizeCallback` implementation, which specifies the `callback_signature` must be `void function_name(GLFWwindow* window, int width, int height)`.

```

1 void framebuffer_size_callback(GLFWwindow* window, int width, int height);
2
3 int main{
4     ... ..
5     glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
6     ... ..
7 }
8
9 void framebuffer_size_callback(GLFWwindow* window, int width, int height)
10 {
11     glViewport(0, 0, width, height);
12     std::cout<< width << " " << height << std::endl;
13 }

```

In C++, you must declare the function before using it. For example, if I move the function `framebuffer_size_callback` after the main function, then we need to declare it before we call it in main function.

```

1 if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
2     std::cerr << "Failed to initialize GLAD\n";
3     return -1;
4 }

```

We have finished the test for GLFW, now let's do the test for GLAD. GLAD is a library that handles the tedious task of loading OpenGL function pointers at runtime. Modern OpenGL requires these pointers to be retrieved from the system's graphics driver, as they are not directly linked at compile time. This allows GLAD to correctly set up the OpenGL environment and make all the modern OpenGL functions available for use in your application.

```

1 while (!glfwWindowShouldClose(window)) {
2     /*****
3     here is the place where you input the data
4     and do the rendering tasks
5     *****/
6     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
7     glClear(GL_COLOR_BUFFER_BIT);
8     glfwSwapBuffers(window);
9     glfwPollEvents();
10 }

```

This is your render loop (aka game loop). It runs once per frame until the window is told to close.

- The `int glfwWindowShouldClose(GLFWwindow *window)` function checks at the start of each loop iteration if GLFW has been instructed to close.
- `glClearColor(...)` sets the clear color.
- `glClear(GL_COLOR_BUFFER_BIT)` clears the color buffer so you start the new frame with a clean screen. Without this, the previous frame's contents would remain on the screen, causing visual artifacts or flickering.
- `void glfwSwapBuffers(GLFWwindow *window)` displays the rendered frame by swapping the back buffer with the front buffer. Modern graphics use double buffering to avoid flickering. You draw to the back buffer, then swap it to the front to show it.
- `void glfwPollEvents(void)` processes input and window events (like keyboard, mouse, window close). Without this, the window would freeze and not respond to user input or OS events.

```

1 while (!glfwWindowShouldClose(window)) {
2     if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
3         glfwSetWindowShouldClose(window, true);
4     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
5     glClear(GL_COLOR_BUFFER_BIT);
6     glfwSwapBuffers(window);
7     glfwPollEvents();
8 }

```

We add a small example to show how GLFW handle the input data. We check whether the user pressed the `Esc` key (otherwise `glfwGetKey` returns `GLFW_RELEASE`). If `Esc` is pressed, we call `glfwSetWindowShouldClose(window, true)` to request the window to close. On the next iteration, `glfwWindowShouldClose(window)` returns `true`, the loop condition fails, and the application exits.

2.1.3 Common Compilation Errors

- If the project name you defined in line 6 is different from the project name you gave when you created the project in Clion, you need to click **File > Reload CMake Project**, otherwise you will receive an error similar to the one listed below:

```
1  ninja: error: unknown target 'project', did you mean 'CGProject'?
```

- Everytime you modify your `CMakeLists.txt`, please reload and rebuilt your project instead of using build button. Click **File > Reload CMake Project** and click **build > Rebuild Project** to reload and rebuild your project and apply all new configurations.
- For MacOS, you will see a message popped up when you first time run your program:

```
1  \libglfw.3.dylib" can't be opened because Apple cannot check it for malicious software.
```

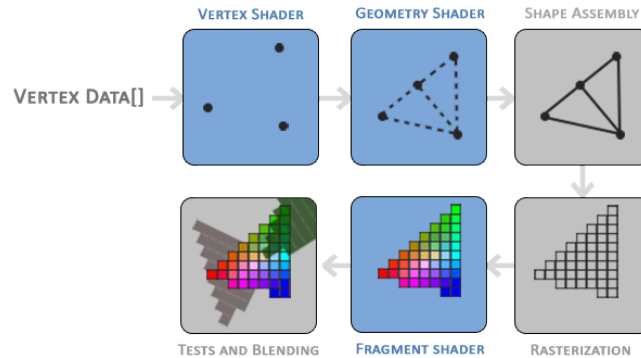
go **System Settings > Privacy & Security**, scroll down to Security section, you should see a message like:

```
1  \libglfw.3.dylib" was blocked from use because it is not from an identified developer.
```

Click **Allow Anyway**.

3 Modern Graphics Pipeline

The transformation of 3D coordinates into 2D pixels is handled by OpenGL’s graphics pipeline. This pipeline consists of two major stages: the first converts 3D coordinates into 2D screen coordinates, and the second maps those 2D coordinates to actual colored pixels.



The pipeline is composed of several sequential steps, where each stage depends on the output of the previous one. These steps are highly specialized, each performing a distinct function, and are designed to run in parallel. Modern GPUs are equipped with thousands of small processing cores to efficiently handle these operations.

Each stage of the pipeline executes small programs called **shaders**, which run on the GPU. Developers can customize certain shaders to replace the default ones, allowing for fine-grained control over specific parts of the rendering process. This customization not only enhances visual effects but also offloads work from the CPU, improving overall performance. Shaders are written in the OpenGL Shading Language (GLSL).

As you can see, the graphics pipeline is made up of many distinct stages, each responsible for a specific part of the process that converts vertex data into fully rendered pixels. We’ll now walk through each stage in a simplified manner to give you a clear overview of how the pipeline functions.

- **Vertex Shader:** Each vertex is processed independently by a small program you write. Its main job is to transform positions from an object’s local coordinates into clip space via model, view, and projection transforms. It can also perform skinning, morphing, or compute per-vertex values (like transforming normals or passing UVs). The only position the rasterizer understands is the vertex shader’s clip-space output; everything else is “varying data” carried to later stages.
- **Geometry Shader (optional).** Operating on whole primitives (a triangle/line/point at a time), this stage can discard them, modify them, or emit new ones (for example, it generates one more triangle by splitting the original one). It’s powerful for special effects and debugging, but it can be expensive; many modern engines avoid it unless they need exactly what it offers. We will not discuss it in this course.
- **Primitive assembly.** It takes as input all the vertices from the vertex (or geometry) shader that form one or more primitives and assembles all the point(s) in the primitive shape given; Rasterization requires complete geometric shapes to convert them into pixel fragments. Shape assembly ensures these shapes are correctly formed.
- **Rasterization.** Triangles that survive clipping are converted into a grid of candidate fragments (potential pixels). During this conversion, per-vertex varyings (like color or UVs) are interpolated across the triangle’s surface so the next stage receives smoothly varying inputs at each fragment.
- **Fragment Shader.** A second program you write runs for each generated fragment. Here you compute the fragment’s final appearance—sampling textures, applying lighting models (Lambert, Phong, PBR), decoding normal maps, or any other shading logic. The shader outputs a color (and optionally a depth value); that output is a proposal for the pixel and still must pass a few tests.
- **Tests And Blending.** Once all the color values for a fragment have been calculated, the object proceeds to a final stage known as alpha testing and blending. In this stage, the system evaluates the fragment’s depth (and stencil) values to determine whether it should be visible or discarded—based on whether it lies in front of or behind other fragments. Additionally, the fragment’s alpha value, which represents its opacity, is used to blend it with other fragments appropriately. As a result, even if a pixel’s color is determined in the fragment shader, the final rendered color may differ significantly when multiple overlapping triangles are involved.

4 Vertex Processing

I'll split this part into three subsections: coordinates, transformations and data representation. We'll begin by examining the data structures and containers used to store and process vertex data within the graphics pipeline, then learn how to transform a 3D object into a target coordinate space using matrix operations. Finally, we'll introduce the coordinate systems used throughout the pipeline, you will finally understand how the 3D models are rendered to your screen.

4.1 Data Flow in OpenGL Pipeline

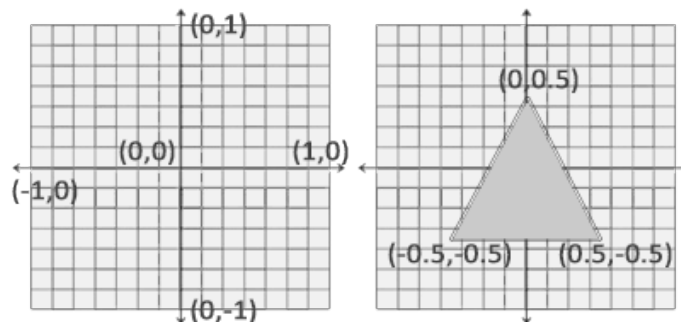
To begin rendering in OpenGL, we first need to supply it with vertex data. Since OpenGL is a 3D graphics API, all coordinates are specified in three dimensions—x, y, and z.

However, OpenGL doesn't automatically convert all 3D coordinates into pixels on the screen. It only processes coordinates that fall within a specific range: -1.0 to 1.0 on each axis. This range is known as Normalized Device Coordinates (NDC). Only vertices within this range will be visible; anything outside it will be clipped and discarded.

After vertex coordinates are processed by the vertex shader, they are expected to be in Normalized Device Coordinates—a compact coordinate space where x, y, and z values range from -1.0 to 1.0. Any vertex that falls outside this range is clipped, meaning it won't be rendered on screen. Unlike traditional screen coordinates, NDC has its origin (0, 0) at the center of the screen, with the positive y-axis pointing upward. To ensure visibility, all transformed vertex positions should end up within this space.

We will go through more details later for mapping a general 3D object to NDC. Now, we directly give a simple triangle that three vertices are specified directly in NDC such that we can directly render it:

```
1 float vertices[] = {  
2     -0.5f, -0.5f, 0.0f,  
3     0.5f, -0.5f, 0.0f,  
4     0.0f, 0.5f, 0.0f  
5 };
```



Although OpenGL operates in 3D space, we can render a 2D triangle by setting the z-coordinate of each vertex to 0.0. This keeps the triangle flat along the depth axis, making it appear two-dimensional on the screen.

4.1.1 Vertex Buffer Object

The next step is to pass it to the first stage of the graphics pipeline: the vertex shader. To do this efficiently, we allocate memory on the GPU, configure how OpenGL should interpret that memory, and specify how the data should be delivered to the graphics card. The vertex shader will then process a specified number of vertices directly from this GPU memory.

To manage this memory, we use **Vertex Buffer Objects (VBOs)**—these are special OpenGL objects that can hold large sets of vertex data in the GPU's memory. The main benefit of VBOs is performance: instead of sending individual vertices one at a time from the CPU (which is slow), we send large batches in a single operation. Once the data resides in GPU memory, the vertex shader can access it nearly instantly, which makes rendering much faster. A VBO is identified by a unique ID, which we create using the `glGenBuffers` function. This ID allows us to reference and manipulate the VBO later in the code.

```
1 unsigned int VBO;  
2 glGenBuffers(1, &VBO);
```

`void glGenBuffers(GLsizei n, GLuint *buffers)`: parameter `n` represents the number of buffer object IDs you want to generate. parameter `buffers` represents a pointer to an array (or a single variable) where OpenGL will store the generated buffer ID(s). `glGenBuffers(1, &VBO)` tells OpenGL to create one buffer object. OpenGL assigns a unique ID to it and stores it in the variable `VBO`. This ID is later used to bind the buffer, upload data, and manage it.

OpenGL has many types of buffer objects and the buffer type of a vertex buffer object is `GL_ARRAY_BUFFER`. OpenGL allows us to bind to several buffers at once as long as they have a different buffer type. We can bind the newly created buffer to the `GL_ARRAY_BUFFER` target with the `glBindBuffer` function:

```
1 glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

From this point on any buffer calls we make (on the `GL_ARRAY_BUFFER` target) will be used to configure the currently bound buffer, which is `VBO`. Then we can make a call to the `glBufferData` function that copies the previously defined vertex data into the buffer's memory:

```
1 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

`void glBufferData(GLenum target, GLsizeiptr size, const void *data, GLenum usage)` is used to copy user-defined data into the currently bound buffer. Its first argument is the type of the buffer we want to copy data into; The second argument specifies the size of the data (in bytes) we want to pass to the buffer (a simple `sizeof` of the vertex data suffices); The third parameter is the actual data we want to send.

The fourth parameter specifies how we want the graphics card to manage the given data. This can take 3 forms:

- `GL_STREAM_DRAW`: the data is set only once and used by the GPU at most a few times.
- `GL_STATIC_DRAW`: the data is set only once and used many times.
- `GL_DYNAMIC_DRAW`: the data is changed a lot and used many times.

The position data of the triangle does not change, is used a lot, and stays the same for every render call so its usage type should best be `GL_STATIC_DRAW`. If, for instance, one would have a buffer with data that is likely to change frequently, a usage type of `GL_DYNAMIC_DRAW` ensures the graphics card will place the data in memory that allows for faster writes.

4.1.2 Vertex Shader

Modern OpenGL requires that we at least set up a vertex and fragment shader if we want to do some rendering so we will briefly introduce shaders and configure two very simple shaders for drawing our first triangle. Later we'll discuss shaders in more detail.

The first thing we need to do is write the vertex shader in the shader language `GLSL` (**OpenGL Shading Language**) and then compile this shader so we can use it in our application. Let's create a shader folder `shaders` in the project directory and create a plain text file `vertex.vert`. Below you'll find the source code of a very basic vertex shader in `GLSL`:

```
1 #version 410 core
2 layout (location = 0) in vec3 aPos;
3
4 void main(){
5     gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
6 }
```

- `#version 330 core`. This tells the GLSL compiler which version of the shading language to use. 330 core means GLSL 3.30, which corresponds to OpenGL 3.3 core profile.
- `layout (location = 0) in vec3 aPos`. This declares an input variable to the vertex shader. In `vec3 aPos` means the shader will receive a 3-component vector (`x`, `y`, `z`) from the vertex data. `layout(location = 0)` is a position mark used for locating vertex data. We will check it later.
- `void main() { ... }`. Every shader must have a main function — it's the entry point.

- `gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0)`. `gl_Position` is a special built-in variable that the vertex shader must output. (There is no `return` statement, `gl_Position` holds the return value.) It represents the vertex's position in clip space (will be converted to NDC by the OpenGL pipeline automatically) and is ready for rasterization.

Here, we're simply taking the input vertex (x, y, z) and extending it into a 4D homogeneous vector (x, y, z, w) , with $w = 1.0$. We will explain what is homogeneous vector and why it is useful here.

The current vertex shader is the most simple vertex shader we can imagine because we did no processing whatsoever on the input data (we directly define our triangle in NDC Space) and simply forwarded it to the shader's output.

4.1.3 Fragment Shader

The fragment shader is the second and final shader we're going to create for rendering a triangle. The fragment shader is all about calculating the color output of your pixels. To keep things simple the fragment shader will always output an orange-ish color.

Colors in computer graphics are represented as an array of 4 values: the red, green, blue and alpha (opacity) component, commonly abbreviated to RGBA. When defining a color in OpenGL or GLSL we set the strength of each component to a value between 0.0 and 1.0. If, for example, we would set red to 1.0 and green to 1.0 we would get a mixture of both colors and get the color yellow. Given those 3 color components we can generate over 16 million different colors!

```

1  #version 410 core
2  out vec4 FragColor;
3
4  void main(){
5      FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
6  }
```

The fragment shader only requires one output variable and that is a vector of size 4 that defines the final color output that we should calculate ourselves. We can declare output values with the `out` keyword, that we here promptly named `FragColor`. Next we simply assign a `vec4` to the color output as an orange color with an alpha value of 1.0 (1.0 being completely opaque).

4.1.4 Vertex Array Buffer Object

A vertex array object (also known as VAO) can be bound just like a vertex buffer object and **any subsequent vertex attribute calls from that point on will be stored inside the VAO**. This has the advantage that when configuring vertex attribute pointers you only have to make those calls once and whenever we want to draw the object, we can just bind the corresponding VAO. This makes switching between different vertex data and attribute configurations as easy as binding a different VAO. All the state we just set is stored inside the VAO.

```

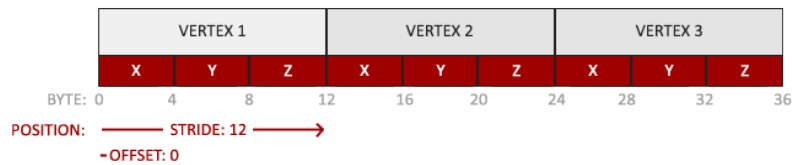
1  ... ...
2  float vertices[] = {
3      -0.5f, -0.5f, 0.0f, // left
4      0.5f, -0.5f, 0.0f, // right
5      0.0f, 0.5f, 0.0f // top
6  };
7
8  unsigned int VBO, VAO;
9  glGenVertexArrays(1, &VAO);
10 glGenBuffers(1, &VBO);
11 // bind the Vertex Array Object first, then bind and set vertex buffer(s), and then configure vertex attributes(s).
12 glBindVertexArray(VAO);
13
14 glBindBuffer(GL_ARRAY_BUFFER, VBO);
15 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
16
17 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
18 glEnableVertexAttribArray(0);
19
20 // Unbind VAO
```

```

21     glBindVertexArray(0);
22     ...

```

First let's see how the vertex data is stored in the GPU's memory:



- The position data is stored as 32-bit (4 byte) floating point values.
- Each position is composed of 3 of those values (x, y, z).
- There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
- The first value in the data is at the beginning of the buffer.

Then, we need to manually tell the data layout to OpenGL via `glVertexAttribPointer` function call:

- The first parameter specifies which vertex attribute we want to configure. **Remember that we specified the location of the position vertex attribute in the vertex shader with layout (location = 0).** This sets the location of the vertex attribute to 0 and since we want to pass data to this vertex attribute, we pass in 0.
- The next argument specifies the size of the vertex attribute. The vertex attribute is a `vec3` so it is composed of 3 values.
- The third argument specifies the type of the data which is `GL_FLOAT` (a `vec*` in GLSL consists of floating point values).
- The next argument specifies if we want the data to be normalized. If we're inputting integer data types (`int`, `byte`) and we've set this to `GL_TRUE`, the integer data is normalized to 0 (or -1 for signed data) and 1 when converted to float. This is not relevant for us so we'll leave this at `GL_FALSE`.
- The fifth argument is known as the **stride** and tells us the space between consecutive vertex attributes. Since the next set of position data is located exactly 3 times the size of a float away we specify that value as the stride.
- The last parameter is of type `void*` and thus requires that weird cast. This is the offset of where the position data begins in the buffer. Since the position data is at the start of the data array this value is just 0.

Now we have defined an object we want to render in our window, we need one last step to render the triangle we define above. Currently we have a `main.cpp` OpenGL program, we have two shaders (shader is small program running on GPU) that we want to use to render the triangle. They are different programs and we need to link them together.

4.1.5 ShaderProgram Class

We will write several functions to load, compile and link shaders to our program. before we write any code, let's create a folder `include` to store header files, and a folder `src` to store source files. Create a header file `shaderprogram.h` in `include` folder and a source file `shaderprogram.cpp` in `src` folder. Now your project structure should be:

```

YourProject/
├── include/
│   └── shaderprogram.h
├── src/
│   └── shaderprogram.cpp
├── main.cpp
└── CMakeLists.txt

```

Since we are introducing new files into our project, we need to update our `CMakeLists.txt`:

```

1  cmake_minimum_required(VERSION 3.27)
2  project(CGProject)
3
4  set(CMAKE_CXX_STANDARD 17)
5
6  #####updated#####
7  # Include GLFW headers
8  include_directories(${CMAKE_SOURCE_DIR}/glfw-3.4.bin.MACOS/include
9                      ${CMAKE_SOURCE_DIR}/glad/include

```

```

10         include)
11
12     # Link GLFW library
13     link_directories(${CMAKE_SOURCE_DIR}/glfw-3.4.bin.MACOS/lib-arm64)
14
15     # Add GLAD source file
16     add_library(glad ${CMAKE_SOURCE_DIR}/glad/src/glad.c)
17
18     #####updated#####
19     #easy to access shader files
20     add_compile_definitions(SHADER_DIR="${CMAKE_SOURCE_DIR}/shaders/")
21
22     #####updated#####
23     add_executable(CGProject main.cpp
24                     src/shaderprogram.cpp)
25
26     # Link GLFW
27     target_link_libraries(CGProject
28                           glad
29                           glfw3.dylib
30                           "-framework OpenGL"
31     )

```

Since we are writing C++, we should code in a OOP Paradigm. In the shaderprogram.h file, let's define a Shader-Program class:

```

1  #ifndef SHADERPROGRAM_H
2  #define SHADERPROGRAM_H
3
4  #include <string>
5  #include <GLFW/glfw3.h>
6  #include <glm.hpp>
7  #include <gtc/matrix_transform.hpp>
8  #include <gtc/type_ptr.hpp>
9
10
11 class ShaderProgram {
12 public:
13     ShaderProgram(const std::string& vertexPath, const std::string& fragmentPath);
14     ~ShaderProgram();
15
16     void use() const;
17     GLuint getID() const;
18
19     void destroy();
20
21 private:
22     GLuint ID;
23     bool isDeleted = false;
24
25     std::string loadShaderSource(const std::string& filePath);
26     static GLuint compileShader(const std::string& source, GLenum shaderType);
27     void linkProgram(GLuint vertexShader, GLuint fragmentShader);
28 };
29
30 #endif

```

The purpose of #ifndef and #define is:

- #ifndef CGPROJECT_SHADERPROGRAM_H: Checks if CGPROJECT_SHADERPROGRAM_H has not been defined yet.
- #define CGPROJECT_SHADERPROGRAM_H: Defines CGPROJECT_SHADERPROGRAM_H so the next time this file is included, the compiler skips it.

- `#endif`: Ends the conditional.

So, the first time the file is included, the contents are processed. On subsequent includes, the compiler sees that `CGPROJECT_SHADERPROGRAM_H` is already defined and skips the file.

In the `shaderprogram.cpp` file, let's implement all member functions (both public and private). The first one is used for loading shader file:

```

1  #include "shaderprogram.h"
2  #include <fstream>
3  #include <sstream>
4  #include <iostream>
5  std::string ShaderProgram::loadShaderSource(const std::string& filePath) {
6      //OpenGL requires the shader to be stored as a const string.
7      //Create an input file stream to read from the file.
8      std::ifstream shaderFile;
9      //bitwise OR operation combining two error state flags used with
10     //input/output streams, particularly std::ifstream for file input.
11     shaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
12     try {
13         shaderFile.open(filePath);
14         std::stringstream shaderStream;
15         //Reads the entire contents of the file into the string stream using the file buffer.
16         shaderStream << shaderFile.rdbuf();
17         shaderFile.close();
18         return shaderStream.str();
19     } catch (std::ifstream::failure& e) {
20         std::cerr << "ERROR::SHADER::FILE_NOT_SUCCESSFULLY_READ: " << filePath << "\n";
21         return "";
22     }
23 }
```

Using two stream objects—`std::ifstream` and `std::stringstream`—is a common and practical pattern for file reading in C++. `std::ifstream` is designed specifically for reading from files. It handles file opening, closing, and error checking. It reads raw data from disk. `std::stringstream` is used to accumulate and manipulate the file content in memory. It allows easy conversion to `std::string`.

The second one is used for compiling a shader:

```

1  //shaderType: GL_VERTEX_SHADER or GL_FRAGMENT_SHADER
2  GLuint ShaderProgram::compileShader(const std::string& source, GLenum shaderType) {
3      //Converts the std::string source code into a C-style string (const char*)
4      //because OpenGL expects shader source code in this format.
5      const char* shaderCode = source.c_str();
6      GLuint shader = glCreateShader(shaderType);
7      glShaderSource(shader, 1, &shaderCode, nullptr);
8      glCompileShader(shader);
9
10     GLint success;
11     GLchar infoLog[512];
12     glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
13     if (!success) {
14         glGetShaderInfoLog(shader, 512, nullptr, infoLog);
15         std::cerr << "ERROR::SHADER::COMPILATION_FAILED\n" << infoLog << "\n";
16     }
17
18     return shader;
19 }
```

The third function is for linking shader to our program:

```

1  void ShaderProgram::linkProgram(GLuint vertexShader, GLuint fragmentShader) {
2      //Creates a new shader program and returns its ID.
```

```

3     ID = glCreateProgram();
4     //Attach the compiled vertex and fragment shaders to the program.
5     glAttachShader(ID, vertexShader);
6     glAttachShader(ID, fragmentShader);
7     glLinkProgram(ID);
8     //After linking, the shaders are no longer needed separately, so it's safe to delete them.
9     glDeleteShader(vertexShader);
10    glDeleteShader(fragmentShader);
11
12    GLint success;
13    GLchar infoLog[512];
14    glGetProgramiv(ID, GL_LINK_STATUS, &success);
15    if (!success) {
16        glGetProgramInfoLog(ID, 512, nullptr, infoLog);
17        std::cerr << "ERROR::PROGRAM::LINKING_FAILED\n" << infoLog << "\n";
18    }
19 }

```

The program object's information log is updated and the program is generated at the time of the link operation. After the link operation, applications are free to modify attached shader objects, compile attached shader objects, detach shader objects, delete shader objects, and attach additional shader objects. None of these operations affects the information log or the program that is part of the program object.

The fourth one is the class constructor, it will be quite different from the one in Python, the constructor does not have a return type, neither a return value. When you use a constructor to create an object, the constructor is automatically called, and the object is stored in the variable you declare:

```

1 ShaderProgram::ShaderProgram(const std::string& vertexPath, const std::string& fragmentPath) {
2     std::string vertexCode = loadShaderSource(vertexPath);
3     std::string fragmentCode = loadShaderSource(fragmentPath);
4
5     //we call compileShader function by supplying different second parameter to indicate different shaders.
6     GLuint vertexShader = compileShader(vertexCode, GL_VERTEX_SHADER);
7     GLuint fragmentShader = compileShader(fragmentCode, GL_FRAGMENT_SHADER);
8
9     linkProgram(vertexShader, fragmentShader);
10 }

```

Some other helper functions that are useful for us to manage our ShaderProgram class objects:

```

1 //This is the destructor of the ShaderProgram class.
2 //It is automatically called when a ShaderProgram object goes out of scope or is deleted.
3 //It calls the destroy() method to clean up the OpenGL shader program.
4 ShaderProgram::~ShaderProgram() {
5     destroy();
6 }
7
8 //This method activates the shader program so that OpenGL uses it for rendering.
9 void ShaderProgram::use() const {
10    glUseProgram(ID);
11 }
12
13 //This is a getter method that returns the shader program's ID.
14 GLuint ShaderProgram::getID() const {
15    return ID;
16 }
17
18 //This method cleans up the shader program by deleting it from OpenGL.
19 !!isDeleted: to avoid deleting the program multiple times.
20 //ID != 0: to ensure there's a valid program to delete.
21
22 void ShaderProgram::destroy() {
23    if (!isDeleted && ID != 0) {

```

```

24         glDeleteProgram(ID);
25         ID = 0;
26         isDeleted = true;
27     }
28 }

```

Now we have our `ShaderProgram` class ready and we can use it in our `main` function:

```

1  ... ..
2  #include "shaderprogram.h"
3  ... ..
4
5  int main() {
6      ... ..
7      if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
8          std::cerr << "Failed to initialize GLAD\n";
9          return -1;
10     }
11     //create a ShaderProgram object shaderProgram
12     std::string vertPath = std::string(SHADER_DIR) + "vertex.vert";
13     std::string fragPath = std::string(SHADER_DIR) + "fragment.frag";
14     ShaderProgram shaderProgram(vertPath, fragPath);
15
16
17     float vertices[] = {
18         -0.5f, -0.5f, 0.0f,
19         0.5f, -0.5f, 0.0f,
20         0.0f, 0.5f, 0.0f
21     };
22     unsigned int VBO;
23     glGenBuffers(1, &VBO);
24     glBindBuffer(GL_ARRAY_BUFFER, VBO);
25     glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
26     ... ..
27     //eliminate the object after using it.
28     shaderProgram.destroy();
29     glfwTerminate();
30     return 0;
31 }
32
33 ... ..

```

Now we are ready to draw the orange triangle in the main render loop:

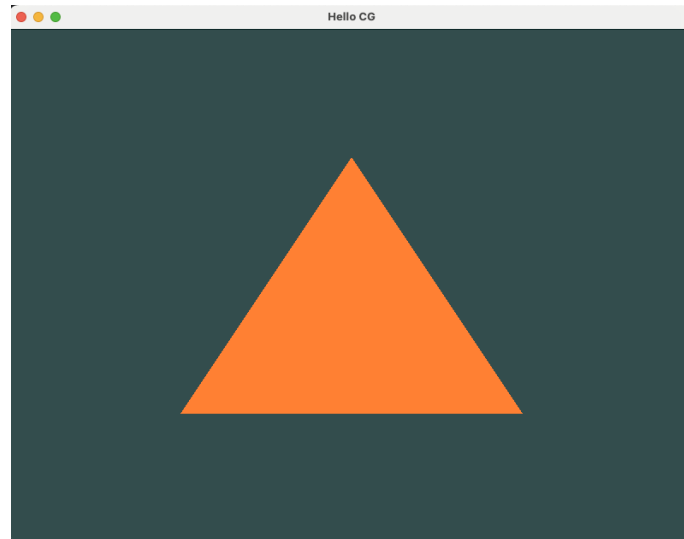
```

1  ... ..
2  while (!glfwWindowShouldClose(window)) {
3      if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
4          glfwSetWindowShouldClose(window, true);
5      glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
6      glClear(GL_COLOR_BUFFER_BIT);
7
8      //enable VAO that stores the specific model's info
9      shaderProgram.use();
10     glBindVertexArray(VAO);
11     glDrawArrays(GL_TRIANGLES, 0, 3);
12
13     glfwSwapBuffers(window);
14     glfwPollEvents();
15 }
16 ... ..

```

The `glDrawArrays` function takes as its first argument the OpenGL primitive type we would like to draw. The second

argument specifies the starting index of the vertex array we'd like to draw; we just leave this at 0. The last argument specifies how many vertices we want to draw, which is 3. Your program should produce a window below:



4.1.6 Element Buffer Object

There is one last thing we'd like to discuss when rendering vertices and that is element buffer objects abbreviated to EBO. To explain how element buffer objects work it's best to give an example: suppose we want to draw a rectangle instead of a triangle. We can draw a rectangle using two triangles (OpenGL mainly works with triangles). This will generate the following set of vertices:

```
1 float vertices[] = {
2     // first triangle
3     0.5f, 0.5f, 0.0f, // top right
4     0.5f, -0.5f, 0.0f, // bottom right
5     -0.5f, 0.5f, 0.0f, // top left
6     // second triangle
7     0.5f, -0.5f, 0.0f, // bottom right
8     -0.5f, -0.5f, 0.0f, // bottom left
9     -0.5f, 0.5f, 0.0f // top left
10 };
```

As you can see, there is some overlap on the vertices specified. We specify bottom right and top left twice! This is an overhead of 50% since the same rectangle could also be specified with only 4 vertices, instead of 6. This will only get worse as soon as we have more complex models that have over 1000s of triangles where there will be large chunks that overlap. **What would be a better solution is to store only the unique vertices and then specify the order at which we want to draw these vertices in.**

An element buffer object (EBO) is a buffer, just like a vertex buffer object, that stores indices that OpenGL uses to decide what vertices to draw.

```
1 float vertices[] = {
2     0.5f, 0.5f, 0.0f, // top right
3     0.5f, -0.5f, 0.0f, // bottom right
4     -0.5f, -0.5f, 0.0f, // bottom left
5     -0.5f, 0.5f, 0.0f // top left
6 };
7 unsigned int indices[] = { // note that we start from 0!
8     0, 1, 3, // first triangle
9     1, 2, 3 // second triangle
10 };
```

You can see that, when using indices, we only need 4 vertices instead of 6. Next we need to create the element buffer object:

```

1 unsigned int EBO;
2 glGenBuffers(1, &EBO);
3 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
4 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

```

Note that we're now giving `GL_ELEMENT_ARRAY_BUFFER` as the buffer target. The last thing left to do is replace the `glDrawArrays` call with `glDrawElements` to indicate we want to render the triangles from an index buffer. When using `glDrawElements` we're going to draw using indices provided in the element buffer object currently bound:

```

1 glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);

```

The first argument specifies the mode we want to draw in, similar to `glDrawArrays`. The second argument is the count or number of elements we'd like to draw. We specified 6 indices so we want to draw 6 vertices in total. The third argument is the type of the indices which is of type `GL_UNSIGNED_INT`. The last argument allows us to specify an offset in the EBO, but we're just going to leave this at 0.

Similar to VBO, we can also bind EBO to VAO, such that we only need to manage VAO when we want to draw a specific model:

```

1 ... ..
2 float vertices[] = {
3     0.5f, 0.5f, 0.0f, // top right
4     0.5f, -0.5f, 0.0f, // bottom right
5     -0.5f, -0.5f, 0.0f, // bottom left
6     -0.5f, 0.5f, 0.0f // top left
7 };
8 unsigned int indices[] = { // note that we start from 0!
9     0, 1, 3, // first triangle
10    1, 2, 3 // second triangle
11 };
12
13 unsigned int VBO, VAO, EBO;
14 glGenVertexArrays(1, &VAO);
15 glGenBuffers(1, &VBO);
16 glGenBuffers(1, &EBO);
17
18 glBindVertexArray(VAO);
19
20 glBindBuffer(GL_ARRAY_BUFFER, VBO);
21 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
22 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
23 glEnableVertexAttribArray(0);
24
25 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
26 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
27
28 glBindVertexArray(0);
29
30 while (!glfwWindowShouldClose(window)) {
31     if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
32         glfwSetWindowShouldClose(window, true);
33     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
34     glClear(GL_COLOR_BUFFER_BIT);
35     glBindVertexArray(VAO);
36     glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
37     glfwSwapBuffers(window);
38     glfwPollEvents();
39 }
40
41 //de-allocate all resources once they've outlived their purpose:
42 glDeleteVertexArrays(1, &VAO);
43 glDeleteBuffers(1, &VBO);

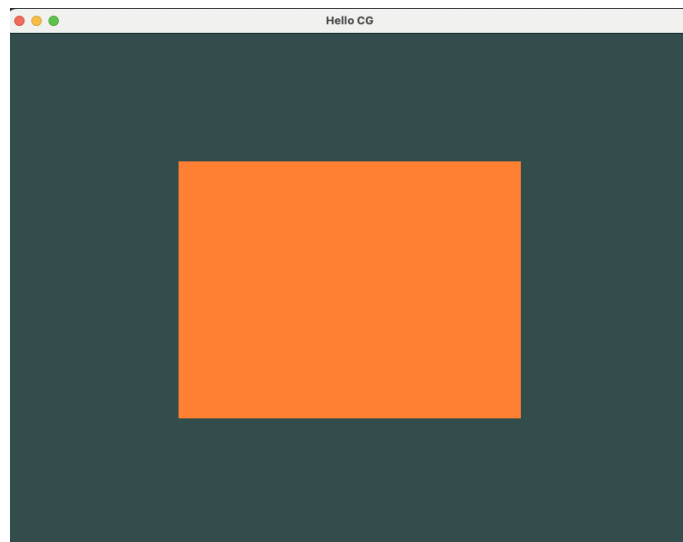
```

```

44     glDeleteBuffers(1, &EBO);
45     shaderProgram.destroy();
46     glfwTerminate();
47     return 0;
48 }
49 ... ...

```

Your program should produce a window below:



4.2 GLSL

Shaders are little programs that rest on the GPU. These programs are run for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs transforming inputs to outputs. 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. **Shaders are written in the C-like language GLSL.** GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

GLSL has, like any other programming language, data types for specifying what kind of variable we want to work with. GLSL has most of the default basic types we know from languages like C: **int**, **float**, **double**, **uint** and **bool**. GLSL also features two container types that we'll be using a lot, namely **vectors** and **matrices**.

4.2.1 Vector

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.
- **ivec**n: a vector of n integers.
- **uvec**n: a vector of n unsigned integers.
- **dvec**n: a vector of n double components.

Most of the time we will be using the basic vecn since floats are sufficient for most of our purposes.

```

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

```

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

```

1 // Accessing components using .xyzw
2 float red = color.x; // same as color.r
3 float green = color.y; // same as color.g
4 float blue = color.z; // same as color.b
5 float alpha = color.w; // same as color.a
6
7 // Accessing components using .rgba
8 vec3 rgb = color.rgb; // same as color.xyz
9
10 // Accessing components using .stpq (commonly used for texture coordinates)
11 vec4 texCoords = vec4(0.1, 0.2, 0.3, 0.4);
12 float s = texCoords.s; // same as texCoords.x
13 float t = texCoords.t; // same as texCoords.y
14 float p = texCoords.p; // same as texCoords.z
15 float q = texCoords.q; // same as texCoords.w

```

The vector datatype allows for some interesting and flexible component selection called **swizzling**. Swizzling allows us to use syntax like this:

```

1 vec4 color = vec4(1.0, 0.5, 0.25, 1.0); // RGBA
2
3 vec3 rgb = color.rgb; // Extracts the first three components: (1.0, 0.5, 0.25)
4 vec2 rg = color.rg; // Extracts red and green: (1.0, 0.5)
5 vec3 bgr = color.bgr; // Rearranges components: (0.25, 0.5, 1.0)
6 vec4 repeated = color.aaaa; // Repeats alpha: (1.0, 1.0, 1.0, 1.0)
7
8 // You can also assign using swizzling:
9 vec4 newColor = vec4(0.0);
10 newColor.rg = vec2(0.8, 0.3); // Sets red and green
11 newColor.ba = color.ba; // Copies blue and alpha from `color`

```

4.2.2 Ins & Outs

Shaders are nice little programs on their own, but they are part of a whole and for that reason we want to have inputs and outputs on the individual shaders so that we can move stuff around. So if we want to send data from one shader to the other we'd have 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).

Let's write a vertex shader and fragment shader to see how the data is transferred between them.

```

1 #version 410 core
2 layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
3
4 out vec4 vertexColor; // specify a color output to the fragment shader
5
6 void main(){
7     gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
8     vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
9 }

```

```

1 #version 410 core
2 out vec4 FragColor;
3
4 in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)
5
6 void main(){
7     FragColor = vertexColor;
8 }

```

4.2.3 layout

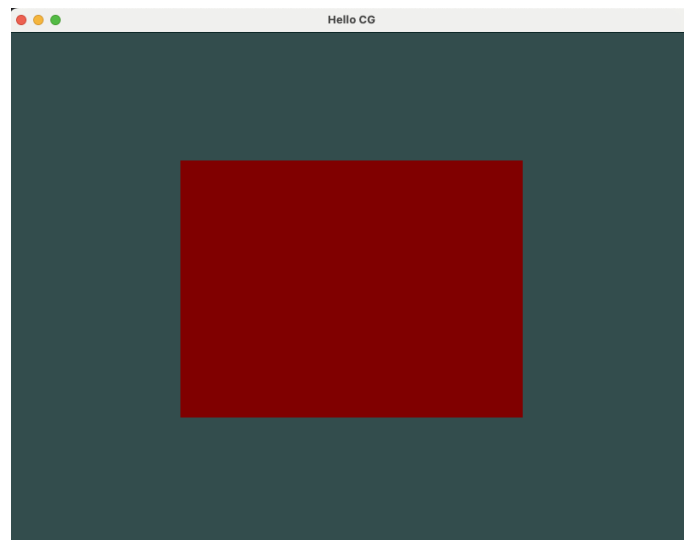
The `layout (location = 0)` syntax is used in GLSL shaders (especially in modern OpenGL) to explicitly specify the location of a shader input or output variable. If you use layout syntax, then you can directly use location value you defined in your main code to locate corresponding data sent to the shader. We use this syntax in our current code:

```
1 //because in the shader we define apos position is 0, which is the first paramter for the function below
2 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

The layout syntax is optional, which means you do not have to specify it in the shader program. If you do not use layout syntax, then you need to manually bind the attribute locations before linking the shader program by calling `glGetAttribLocation` function:

```
1 GLint posAttrib = glGetAttribLocation(shaderProgram.getID(), "aPos");
2 glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
3 glEnableVertexAttribArray(posAttrib);
```

Both ways are equal, you can choose the one you like. Now your window should be:



4.2.4 Uniforms

Uniforms are another way to pass data from our 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**.

```
1 #version 410 core
2 out vec4 FragColor;
3
4 uniform vec3 ourColor; // we set this variable in the OpenGL code.
5
6 void main(){
7     FragColor = vec4(ourColor, 1.0f);
8 }
```

We defined a uniform `vec3 ourColor` in the fragment shader and assigned it as the fragment's output color. This means all fragments will share the same color value. Now, let's add several functions in our `ShaderProgram` class to support uniform variable configuration:

```

1  class ShaderProgram {
2  public:
3      ... ..
4
5      // Set uniform variables of various types
6      void setUniform(const std::string& name, int value) const;
7      void setUniform(const std::string& name, float value) const;
8      void setUniform(const std::string& name, const glm::vec3& value) const;
9      void setUniform(const std::string& name, const glm::mat4& value) const;
10
11      ... ..
12 };

```

And implement these functions:

```

1  void ShaderProgram::setUniform(const std::string& name, int value) const {
2      GLint location = glGetUniformLocation(ID, name.c_str());
3      if (location == -1) {
4          std::cerr << "Warning: uniform '" << name << "' not found in shader.\n";
5          return;
6      }
7      glUniform1i(location, value);
8  }
9
10 void ShaderProgram::setUniform(const std::string& name, float value) const {
11     GLint location = glGetUniformLocation(ID, name.c_str());
12     if (location == -1) {
13         std::cerr << "Warning: uniform '" << name << "' not found in shader.\n";
14         return;
15     }
16     glUniform1f(location, value);
17 }
18
19 void ShaderProgram::setUniform(const std::string& name, const glm::vec3& value) const {
20     GLint location = glGetUniformLocation(ID, name.c_str());
21     if (location == -1) {
22         std::cerr << "Warning: uniform '" << name << "' not found in shader.\n";
23         return;
24     }
25     glUniform3fv(location, 1, glm::value_ptr(value));
26 }
27
28 void ShaderProgram::setUniform(const std::string& name, const glm::mat4& value) const {
29     GLint location = glGetUniformLocation(ID, name.c_str());
30     if (location == -1) {
31         std::cerr << "Warning: uniform '" << name << "' not found in shader.\n";
32         return;
33     }
34     glUniformMatrix4fv(location, 1, GL_FALSE, glm::value_ptr(value));
35 }

```

`glUniform` requires a specific postfix for the type of the uniform you want to 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.
- 4f: the function expects 4 floats as its value.
- fv: the function expects a float vector/array as its value.

You'll notice that the same color is applied to all vertices each frame. As time progresses, the color value recalculated per frame gradually changes.

```
1  ... ...
2      int vertexColorLocation = glGetUniformLocation(shaderProgram.getID(), "ourColor");
3
4      while (!glfwWindowShouldClose(window)) {
5          if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
6              glfwSetWindowShouldClose(window, true);
7          glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
8          glClear(GL_COLOR_BUFFER_BIT);
9
10         //enable VAO that stores the specific model's info
11         shaderProgram.use();
12         float timeValue = glfwGetTime();
13         float greenValue = (sin(timeValue) / 2.0f) + 0.5f;
14         shaderProgram.setUniform("ourColor", glm::vec3(0.0f, greenValue, 0.0f));
15         glBindVertexArray(VAO);
16         glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
17
18         glfwSwapBuffers(window);
19         glfwPollEvents();
20     }
21  ... ...
```

4.2.5 More Attributes

We saw in the previous sections how we can fill a VBO, configure vertex attribute pointers and store it all in a VAO. This time, we also want to add color data to the vertex data. We're going to add color data as 3 floats to the vertices array. We assign a red, green, blue and black color to each of the corners of our rectangular respectively:

```
1      float vertices[] = {
2          0.5f,  0.5f, 0.0f,  1.0f, 0.0f, 0.0f, // top right
3          0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f, // bottom right
4          -0.5f, -0.5f, 0.0f,  0.0f, 0.0f, 1.0f, // bottom left
5          -0.5f,  0.5f, 0.0f,  0.0f, 0.0f, 0.0f // top left
6      };
```

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

```
1  #version 410 core
2  in vec3 aPos; // the position variable has attribute position 0
3  in vec3 aColor; // the color variable has attribute position 1
4
5  out vec4 vertexColor; // specify a color output to the fragment shader
6
7  void main(){
8      gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
9      vertexColor = vec4(aColor, 1.0);
10 }
```

Since we no longer use a uniform for the fragment's color, but now use the ourColor output variable we'll have to change the fragment shader as well:

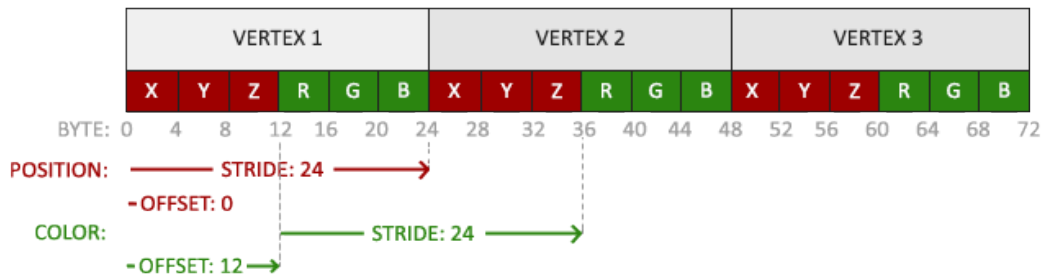
```
1  #version 410 core
2  out vec4 FragColor;
3
4  in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)
```

```

5 //uniform vec4 ourColor; // we set this variable in the OpenGL code.
6
7 void main(){
8     FragColor = vertexColor;
9     //FragColor = ourColor;
10 }

```

Because we added another vertex attribute and updated the VBO's memory we have to re-configure the vertex attribute pointers. The updated data in the VBO's memory now looks a bit like this:



Knowing the current layout we can update the vertex format with `glVertexAttribPointer`:

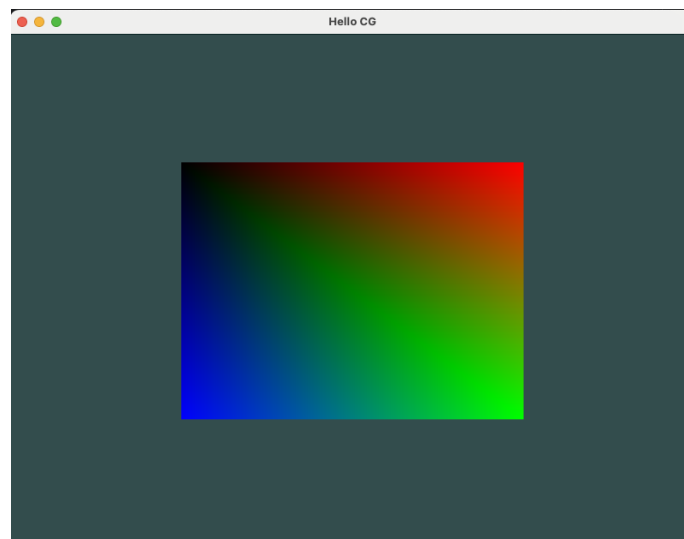
```

1 GLint posAttrib = glGetAttribLocation(shaderProgram.getID(), "aPos");
2 //per vertex size is changed from 3 to 6
3 glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
4 glEnableVertexAttribArray(posAttrib);
5
6 GLint colAttrib = glGetAttribLocation(shaderProgram.getID(), "aColor");
7 glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
8 glEnableVertexAttribArray(colAttrib);

```

Since we now have two vertex attributes we have to re-calculate the stride value. To get the next attribute value (e.g. the next x component of the position vector) in the data array we have to move 6 floats to the right, three for the position values and three for the color values. This gives us a stride value of 6 times the size of a float in bytes (= 24 bytes). Also, this time we have to specify an offset. For each vertex, the position vertex attribute is first so we declare an offset of 0. The color attribute starts after the position data so the offset is $3 * \text{sizeof(float)}$ in bytes (= 12 bytes).

Running the application should result in the following image:



4.3 Textures

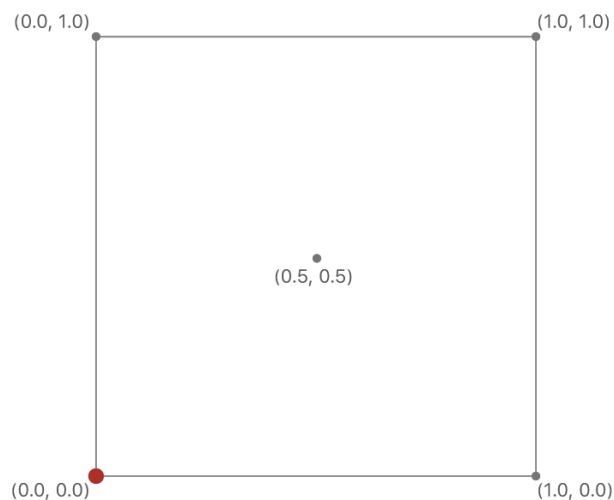
A texture is a 2D image (even 1D and 3D textures exist) used to add detail to an object; think of a texture as a piece of paper with a nice brick image (for example) on it neatly folded over your 3D house so it looks like your house has a stone exterior. Below is a texture image of a brick wall.



In order to map a texture to the rectangular 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.

4.3.1 Texture Coordinates

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.



We specify 4 texture coordinate points for the rectangular.

```
1 float texCoords[] = {  
2     0.0f, 0.0f,  
3     1.0f, 0.0f,  
4     0.0f, 1.0f,  
5     1.0f, 1.0f  
6 };
```

4.3.2 Texture Wrapping

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

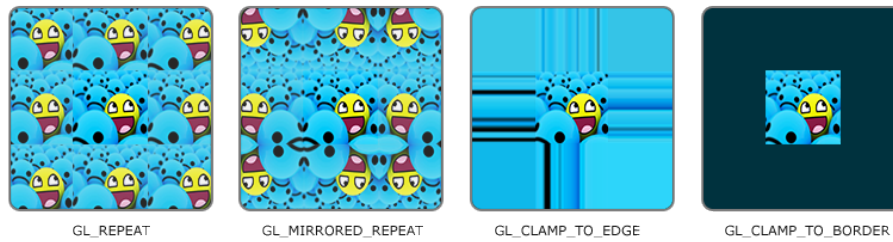


Figure 3: original image by Hólger Rezende

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:

```
1 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
2 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`.

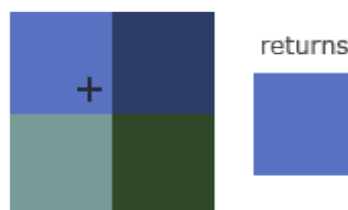
If we choose the `GL_CLAMP_TO_BORDER` option we should also specify a border color. This is done using the `f`v equivalent of the `glTexParameter` function with `GL_TEXTURE_BORDER_COLOR` as its option where we pass in a float array of the border's color value:

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

4.3.3 Texture Filtering

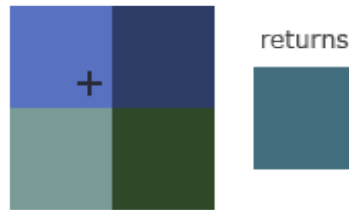
Texture coordinates do not depend on resolution but can be any floating point value, thus OpenGL has to figure out which texture pixel (also known as a texel) to map the texture coordinate to. This becomes especially important if you have a very large object and a low resolution texture. You probably guessed by now that OpenGL has options for this texture filtering as well. There are several options available but for now we'll discuss the most important options: `GL_NEAREST` and `GL_LINEAR`.

`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. Below 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. Below we can see that a mixed color of the neighboring pixels is returned:



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 (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.

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:

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

4.3.4 Mipmaps

Imagine we had a large room with 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 waste of memory bandwidth using high resolution textures on small objects.

To solve this issue OpenGL uses a concept called **mipmaps** that is basically a collection of texture images where each subsequent texture is twice as small compared to the previous one. The idea behind mipmaps should be easy to understand: **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. Let's take a closer look at what a mipmapped texture looks like:



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 `glGenerateMipmap` after we've created a texture.

When switching between mipmaps levels during rendering OpenGL may 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. 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.

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

```
1 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
2 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.

4.3.5 Loading And Creating Textures

After all of these, we need to load the image into our program such that we can further process it. `stb_image.h` is a very popular single header image loading library by Sean Barrett that is able to load most popular file formats and is easy to integrate in your project(s). `stb_image.h` can be downloaded from https://github.com/nothings/stb/blob/master/stb_image.h.

Then create a `stb_image.cpp` in your src folder and type the content below:

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

Remember to add this file to the CMakeLists.txt. Now download "wall.jpg" and create a folder `assets` in your project directory, we will use this folder to hold all of the assets we will use in this course. To make the file access more convenient, we define a constant representing the folder in our `CMakeLists.txt`:

```
1 add_compile_definitions(ASSET_DIR="${CMAKE_SOURCE_DIR}/assets/")
```

Then we can use `stbi_load` function to load the image:

```
1 #include "stb_image.h"
2 ...
3 int width, height, nrChannels;
4 std::string texPath = std::string(ASSET_DIR) + "wall.jpg";
5 unsigned char *data = stbi_load(texPath.c_str(), &width, &height, &nrChannels, 0);
```

Then we need to generate a texture object to hold it:

```
1 unsigned int texture;
2 glGenTextures(1, &texture);
3 glBindTexture(GL_TEXTURE_2D, texture);
```

Now we have the texture data and texture object ready, we only need to link the data to the object:

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

This is a large function with quite a few parameters so we'll walk through them step-by-step:

- 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.
- 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.

Once `glTexImage2D` is called, the currently bound texture object now has the texture image attached to it. However, currently it only has the base-level of the texture image loaded and 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. We will discuss mipmap later.

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

```
1 stbi_image_free(data);
```

Since we have three attributes and per-vertex data increases to 8 float numbers, we need to adjust the parameters of `glVertexAttribPointer` function for previous position and color attributes, and add code for texture coordinates:

```
1
2 float vertices[] = {
3     0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // top right
4     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // bottom right
5     -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom left
6     -0.5f, 0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f // top left
7 };
8 ...
9 GLint posAttrib = glGetAttribLocation(shaderProgram.getID(), "aPos");
10 glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
11 glEnableVertexAttribArray(posAttrib);
12
13 GLint colAttrib = glGetAttribLocation(shaderProgram.getID(), "aColor");
14 glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
15 glEnableVertexAttribArray(colAttrib);
16
17 GLint texAttrib = glGetAttribLocation(shaderProgram.getID(), "aTexCoord");
```

```

18 glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
19 glEnableVertexAttribArray(texAttrib);

```

Let's put all texture configuration code together:

```

1 unsigned int texture;
2 glGenTextures(1, &texture);
3 glActiveTexture(GL_TEXTURE0);
4 glBindTexture(GL_TEXTURE_2D, texture);
5 // set the texture wrapping/filtering options (on the currently bound texture object)
6 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
7 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
8 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
9 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
10 // load and generate the texture
11 int width, height, nrChannels;
12 std::string texPath = std::string(ASSETS_DIR) + "wall.jpg";
13 unsigned char *data = stbi_load(texPath.c_str(), &width, &height, &nrChannels, 0);
14 if (data){
15     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
16     glGenerateMipmap(GL_TEXTURE_2D);
17 }
18 else{
19     std::cout << "Failed to load texture" << std::endl;
20 }
21 stbi_image_free(data);

```

Currently we just processed texture data in OpenGL program, we need to give it to the shaders so the GPU running those small program can produce the results we want:

```

1 #version 410 core
2 in vec3 aPos; // the position variable has attribute position 0
3 in vec3 aColor; // the color variable has attribute position 1
4 in vec2 aTexCoord; // the texture coords has attribute position 2
5
6 out vec4 vertexColor; // specify a color output to the fragment shader
7 out vec2 TexCoord; // specify a texture coord output to the fragment shader
8
9 void main(){
10     gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
11     vertexColor = vec4(aColor, 1.0);
12     TexCoord = aTexCoord;
13 }

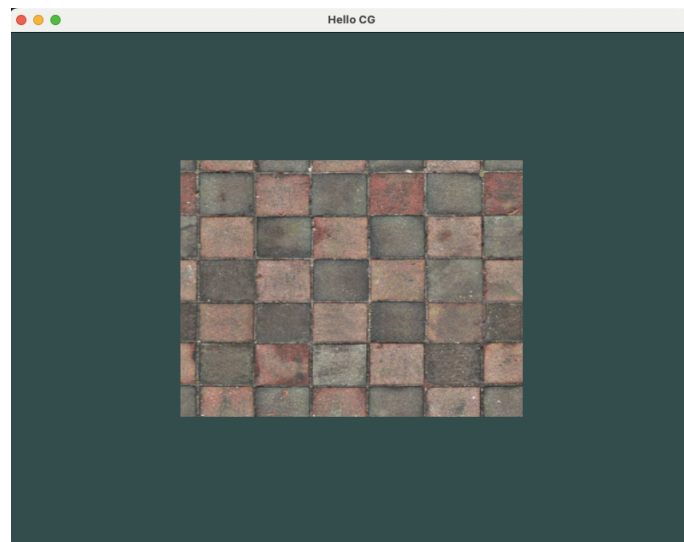
```

```

1 #version 410 core
2 out vec4 FragColor;
3
4 in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)
5 in vec2 TexCoord; // the input variable from the vertex shader (same name and same type)
6
7 uniform sampler2D ourTexture; // texture data we defined in main program
8
9 void main(){
10     FragColor = texture(ourTexture, TexCoord);
11 }

```

Now you window should be:



4.3.6 Texture Units

You probably wondered why the `sampler2D` variable in your fragment shader is a uniform if we didn't even assign it some value with `glUniform`. Using `glUniform1i` we can actually assign a location value to the texture sampler so we can set multiple textures at once in a fragment shader. This 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 so we didn't need to assign a location in the previous section; note that not all graphics drivers assign a default texture unit so the previous section may not have rendered for you.**

The main purpose of texture units is to allow us to use more than 1 texture in our shaders. 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 we didn't have to activate any texture units in the previous example when using `glBindTexture`.

OpenGL should have at least a minimum of 16 texture units for you to use which you can activate using `GL_TEXTURE0` to `GL_TEXTURE15`. They are defined in order so we could also get `GL_TEXTURE8` via `GL_TEXTURE0 + 8` for example, which is useful when we'd have to loop over several texture units.

We still however need to edit the fragment shader to accept another sampler.

```

1  #version 410 core
2  out vec4 FragColor;
3
4  in vec4 vertexColor;
5  in vec2 TexCoord;
6
7  uniform sampler2D texture1;
8  uniform sampler2D texture2;
9
10 void main()
11 {
12     FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2);
13 }
```

The final output color is now 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.

We now want to load and create another texture; you should be familiar with the steps now. Make sure to create another texture object, load the image and generate the final texture using `glTexImage2D`.

```

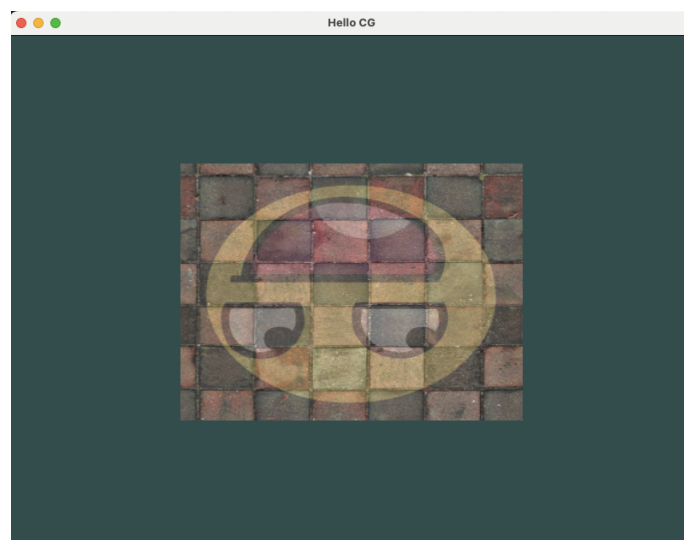
1  ... ..
2  shaderProgram.use(); // necessary
3  unsigned int texture1;
4  glGenTextures(1, &texture1);
```

```

5     glActiveTexture(GL_TEXTURE0);
6     glBindTexture(GL_TEXTURE_2D, texture1);
7     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
8     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
9     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
10    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
11    // load and generate the texture
12    int width, height, nrChannels;
13    std::string texPath = std::string(ASSETS_DIR) + "wall.jpg";
14    unsigned char *data = stbi_load(texPath.c_str(), &width, &height, &nrChannels, 0);
15    if (data){
16        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
17        glGenerateMipmap(GL_TEXTURE_2D);
18    }
19    else{
20        std::cout << "Failed to load texture" << std::endl;
21    }
22    glUniform1i(glGetUniformLocation(shaderProgram.getID(), "texture1"), 0);
23    stbi_image_free(data);
24
25    unsigned int texture2;
26    glGenTextures(1, &texture2);
27    glActiveTexture(GL_TEXTURE1);
28    glBindTexture(GL_TEXTURE_2D, texture2);
29    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
30    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
31    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
32    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
33    // load and generate the texture
34    texPath = std::string(ASSETS_DIR) + "awesomface.png";
35    data = stbi_load(texPath.c_str(), &width, &height, &nrChannels, 0);
36    if (data){
37        //Note that we now load a .png image that includes an alpha (transparency) channel.
38        //This means we now need to specify that the image data contains an alpha channel as well by using GL_RGBA.
39        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);
40        glGenerateMipmap(GL_TEXTURE_2D);
41    }
42    else{
43        std::cout << "Failed to load texture" << std::endl;
44    }
45    glUniform1i(glGetUniformLocation(shaderProgram.getID(), "texture2"), 1);
46    stbi_image_free(data);
47    ...

```

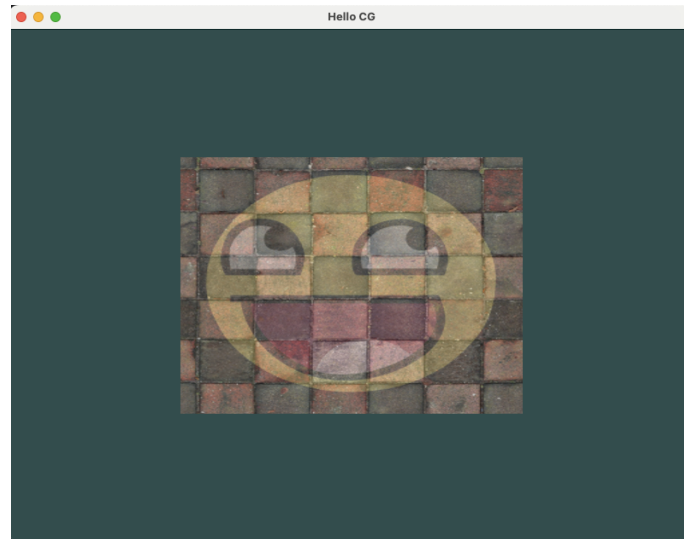
Then you will get a window below:



You probably noticed that the 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. Luckily for us, `stb_image.h` can flip the y-axis during image loading by adding the following statement before loading any image:

```
1 stbi_set_flip_vertically_on_load(true);
```

After telling `stb_image.h` to flip the y-axis when loading images you should get the following result:



Now in our fragment shader, we do not use vertex color attribute, we can slightly modify our fragment shader to produce a more funny output:

```
1 FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2) * vertexColor;
```

Then recompile your code you will get:



Since the texture is part of shader program, now lets add one more function to our `ShaderProgram` class to allow us to load textures:

```
1 #include <string>
2 #include "glad/glad.h"
3 #include "GLFW/glfw3.h"
4 #include <glm.hpp>
```

```

6  #include <gtc/matrix_transform.hpp>
7  #include <gtc/type_ptr.hpp>
8  #include "stb_image.h"
9  ... ..
10 // Loads an image, creates a GL texture, binds it to `textureUnit`,
11 // and sets `samplerName` in THIS program to that unit.
12 // Returns GL texture ID (0 on failure).
13 GLuint bindTexture2D(const std::string& samplerName,
14                     const std::string& filePath,
15                     GLint textureUnit,
16                     bool flipVertical = true,
17                     GLint wrap = GL_REPEAT,
18                     GLint minFilter = GL_LINEAR_MIPMAP_LINEAR,
19                     GLint magFilter = GL_LINEAR,
20                     bool generateMipmaps = true) const;
21 ... ..

```

Also implement this function:

```

1  ... ..
2  GLuint ShaderProgram::bindTexture2D(const std::string& samplerName,
3                                     const std::string& filePath,
4                                     GLint textureUnit,
5                                     bool flipVertical,
6                                     GLint wrap,
7                                     GLint minFilter,
8                                     GLint magFilter,
9                                     bool generateMipmaps) const
10 {
11     if (samplerName.empty() || filePath.empty()) {
12         std::cerr << "bindTexture2D: empty samplerName or filePath\n";
13         return 0;
14     }
15
16     // Ensure we set uniforms on the correct program
17     use();
18
19     stbi_set_flip_vertically_on_load(flipVertical ? 1 : 0);
20     int w = 0, h = 0, ch = 0;
21     unsigned char* data = stbi_load(filePath.c_str(), &w, &h, &ch, 0);
22     if (!data) {
23         std::cerr << "Failed to load texture: " << filePath << "\n";
24         return 0;
25     }
26
27     GLenum srcFormat = GL_RGB;
28     GLint internal = GL_RGB;
29     if (ch == 4) { srcFormat = GL_RGBA; internal = GL_RGBA; }
30     else if (ch == 3) { srcFormat = GL_RGB; internal = GL_RGB; }
31     else if (ch == 1) { srcFormat = GL_RED; internal = GL_RED; }
32
33     GLuint texID = 0;
34     glGenTextures(1, &texID);
35
36     glActiveTexture(GL_TEXTURE0 + textureUnit);
37     glBindTexture(GL_TEXTURE_2D, texID);
38
39     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap);
40     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, wrap);
41     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, minFilter);
42     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, magFilter);
43
44     glTexImage2D(GL_TEXTURE_2D, 0, internal, w, h, 0, srcFormat, GL_UNSIGNED_BYTE, data);
45     if (generateMipmaps) glGenerateMipmap(GL_TEXTURE_2D);

```

```

46     stbi_image_free(data);
47
48
49     // bind sampler uniform to this unit
50     GLint loc = glGetUniformLocation(ID, samplerName.c_str());
51     if (loc >= 0) {
52         glUniform1i(loc, textureUnit);
53     } else {
54         std::cerr << "Warning: sampler uniform not found: " << samplerName << "\n";
55     }
56
57     return texID;
58 }
59 ... ..

```

In the main function, we can drastically simplify the code, now we only need two lines to load two textures:

```

1 ... ..
2 shaderProgram.bindTexture2D("texture1", std::string(ASSETS_DIR) + "wall.jpg", 0, false);
3 shaderProgram.bindTexture2D("texture2", std::string(ASSETS_DIR) + "awesomeface.png", 1, true);
4 ... ..

```

4.4 Mesh Class

Looking at our code, we see that the main function contains many lines for interpreting vertex data, creating buffer objects, and performing cleanup after rendering. This is manageable when rendering only a single model, but as the number of models grows, the main function will quickly become unmanageable. From an OOP perspective, wouldn't it be better to create a class that handles all model-related tasks?

4.4.1 Mesh.h

This file declares the Mesh class and its interface. It includes:

- Constructor: Takes in vertex data, index data, and a list of vertex attributes (like position, color, etc.) and a list of texture specifications.
- Destructor: Automatically cleans up GPU resources when the object is destroyed.
- Public Methods:
 - draw(): Renders the mesh using `glDrawElements`.
 - bind() / unbind(): Binds or unbinds the VAO and textures.
 - cleanup(): Manually deletes VAO, VBO, and EBO (optional if you want to free memory before destruction) and textures.
- Private Members:
 - VAO, VBO, EBO: OpenGL buffer object IDs.
 - indexCount: Number of indices to draw.
 - createBuffers_ is a private function accessed by constructor to further simplify its structure.

The `VertexAttribute` struct is a flexible way to describe how vertex data is laid out in memory, which is essential when working with different models that may have different vertex formats:

- What each attribute is (position, color, normal, etc.)
- How to interpret the data (type, size, stride, offset)

Similar setup for the textures.

```

1 // Created by dengq on 8/29/25.
2 //
3
4 #ifndef MESH_H
5 #define MESH_H

```

```

6  #include <vector>
7  #include <string>
8  #include <glad/glad.h>
9  #include <glm.hpp>
10 #include <gtc/matrix_transform.hpp>
11 #include <gtc/type_ptr.hpp>
12 #include <cstdio>
13 #include <shaderprogram.h>
14
15 #include <assimp/Importer.hpp>
16 #include <assimp/scene.h>
17 #include <assimp/postprocess.h>
18
19 struct VertexAttribute {
20     GLuint pos;           // attribute location (index) in the shader
21     GLint size;           // number of components (e.g., 3 for vec3)
22     GLenum type;          // GL_FLOAT, etc.
23     GLboolean normalized;
24     GLsizei stride;        // byte stride of the vertex
25     size_t offset;        // byte offset to the first component
26 };
27
28 class Mesh {
29 public:
30     Mesh(std::vector<float> v, std::vector<unsigned int> idx, GLuint id);
31
32     ~Mesh();
33
34     // Drawing
35     // void draw() const;
36     void draw() const;
37     // Bind/unbind VAO and all registered textures
38     void bind() const;
39     void unbind() const;
40
41     // Resource cleanup (safe to call multiple times)
42     void cleanup();
43
44 private:
45     // Buffer setup helpers
46     void createBuffers_();
47
48     GLuint VAO = 0, VBO = 0, EBO = 0;
49     GLsizei indexCount = 0;
50
51     GLuint shaderProgramID;
52     // Transformation M, V, P;
53     std::vector<float> vertices = {};
54     std::vector<unsigned int> indices = {};
55
56 };
57
58 #endif

```

4.4.2 Mesh.cpp

Mesh class implementation is very straightforward, we slightly modify the corresponding parts and copy them to the correct positions:

```

1  // Created by dengq on 8/29/25.
2  //
3  // Created by dengq on 8/29/25.
4  //
5  #include "mesh.h"

```

```

6  #include "stb_image.h"
7  #include <iostream>
8  #include "tiny_obj_loader.h"
9  #include <unordered_set>
10
11 // Vertex layout: [px,py,pz, nx,ny,nz, u,v]
12
13 void Mesh::createBuffers_()
14 {
15
16     const GLint posLoc = glGetAttribLocation(shaderProgramID, "aPos");
17     const GLint norLoc = glGetAttribLocation(shaderProgramID, "aColor");
18     const GLint uvLoc = glGetAttribLocation(shaderProgramID, "aTexCoord");
19
20     GLsizei stride = 8 * sizeof(float);
21     std::vector<VertexAttribute> attributes = {
22         VertexAttribute{ static_cast<GLuint>(posLoc), 3, GL_FLOAT, GL_FALSE, stride, 0 },
23         VertexAttribute{ static_cast<GLuint>(norLoc), 3, GL_FLOAT, GL_FALSE, stride, 3 * sizeof(float)},
24         VertexAttribute{ static_cast<GLuint>(uvLoc), 2, GL_FLOAT, GL_FALSE, stride, 6 * sizeof(float) }};
25
26     indexCount = static_cast<GLsizei>(indices.size());
27
28     glGenVertexArrays(1, &VAO);
29     glGenBuffers(1, &VBO);
30     glGenBuffers(1, &EBO);
31
32     glBindVertexArray(VAO);
33
34     glBindBuffer(GL_ARRAY_BUFFER, VBO);
35     glBufferData(GL_ARRAY_BUFFER,
36         static_cast<GLsizeiptr>(vertices.size() * sizeof(float)),
37         vertices.data(),
38         GL_STATIC_DRAW);
39
40     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
41     glBufferData(GL_ELEMENT_ARRAY_BUFFER,
42         static_cast<GLsizeiptr>(indices.size() * sizeof(unsigned int)),
43         indices.data(),
44         GL_STATIC_DRAW);
45
46     for (const auto& attr : attributes) {
47         glVertexAttribPointer(attr.pos,
48             attr.size,
49             attr.type,
50             attr.normalized,
51             attr.stride,
52             reinterpret_cast<const void*>(attr.offset));
53         glEnableVertexAttribArray(attr.pos);
54     }
55
56     glBindVertexArray(0);
57 }
58
59 Mesh::Mesh(std::vector<float> v, std::vector<unsigned int> idx, GLuint id){
60     shaderProgramID = id;
61     vertices = v;
62     indices = idx;
63
64     createBuffers_();
65 }
66
67 Mesh::~Mesh() {
68     cleanup();
69 }
70
71 void Mesh::draw() const{

```

```

72     bind();
73     glDrawElements(GL_TRIANGLES, indexCount, GL_UNSIGNED_INT, nullptr);
74     unbind();
75 }
76
77 void Mesh::bind() const {
78     glBindVertexArray(VAO);
79 }
80
81 void Mesh::unbind() const {
82     glBindVertexArray(0);
83 }
84
85 void Mesh::cleanup() {
86     if (VAO) { glDeleteVertexArrays(1, &VAO); VAO = 0; }
87     if (VBO) { glDeleteBuffers(1, &VBO); VBO = 0; }
88     if (EBO) { glDeleteBuffers(1, &EBO); EBO = 0; }
89 }

```

4.4.3 Rewrite Main Function

We use `std::vector` instead of C-style arrays here. Vector container is much more convenient compared to C-style arrays.

```

1  #include <glad/glad.h>
2  #include <GLFW/glfw3.h>
3  #include "shaderprogram.h"
4  #include "stb_image.h"
5  #include "mesh.h"
6  #include <iostream>
7
8  void framebuffer_size_callback(GLFWwindow* window, int width, int height);
9
10 int main() {
11     ... ..
12     // change the type from c array to cpp vector
13     const std::vector<float> vertices = {
14         // pos          // color          // uv
15         0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 1.0f, // top right
16         0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // bottom right
17         -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom left
18         -0.5f, 0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f // top left
19     };
20     const std::vector<unsigned int> indices = {
21         0, 1, 3,
22         1, 2, 3
23     };
24
25     Mesh quad(vertices, indices, shaderProgram.getID());
26
27     while (!glfwWindowShouldClose(window)) {
28         if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
29             glfwSetWindowShouldClose(window, true);
30         glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
31         glClear(GL_COLOR_BUFFER_BIT);
32
33         //enable VAO that stores the specific model's info
34         shaderProgram.use();
35         shaderProgram.bindTexture2D("texture1", std::string(ASSETS_DIR) + "wall.jpg", 0, false);
36         shaderProgram.bindTexture2D("texture2", std::string(ASSETS_DIR) + "awesomeface.png", 1, true);
37         quad.draw();
38
39         glfwSwapBuffers(window);
40         glfwPollEvents();

```

```

41     }
42
43     quad.cleanup();
44     shaderProgram.destroy();
45     glfwTerminate();
46     return 0;
47 }
48
49 void framebuffer_size_callback(GLFWwindow* window, int width, int height)
50 {
51     glViewport(0, 0, width, height);
52     std::cout << width << " " << height << std::endl;
53 }

```

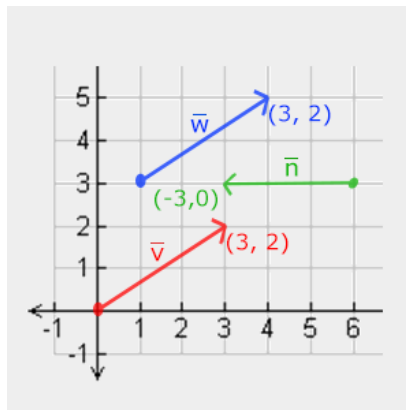
5 Transformations

So far, we've learned how to create objects, color them, and even add textures for more detail. But they're still not very exciting because they don't move. One option would be to adjust their vertices and reconfigure buffers every frame, but that's complicated and inefficient. A much better approach is to transform objects using matrices.

Matrices might look intimidating at first, but they're powerful tools that make transformations much easier once you get used to them. To fully understand how they work, we'll need to take a short detour into some math.

5.1 Vector Basics

In its most basic definition, vectors are directions and nothing more. A vector has a direction and a magnitude (also known as its strength or length). Below you'll see 3 vectors where each vector is represented with (x,y) as arrows in a 2D graph. Because it is more intuitive to display vectors in 2D (rather than 3D) you can think of the 2D vectors as 3D vectors with a z coordinate of 0. Since vectors represent directions, the origin of the vector does not change its value. In the graph below we can see that the vectors $\vec{v}$ and $\vec{w}$ are equal even though their origin is different:



When displaying vectors in formulas they are generally displayed as follows:

$$\vec{v} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

A **scalar** is a single number. When adding, subtracting, multiplying, or dividing a vector with a scalar we simply add, subtract, multiply, or divide each element of the vector by the scalar. For addition it would look like this:

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + x \Rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} x \\ x \\ x \end{pmatrix} = \begin{pmatrix} 1+x \\ 2+x \\ 3+x \end{pmatrix}$$

Where operators can be $+$, $-$, $\times$, $\div$.

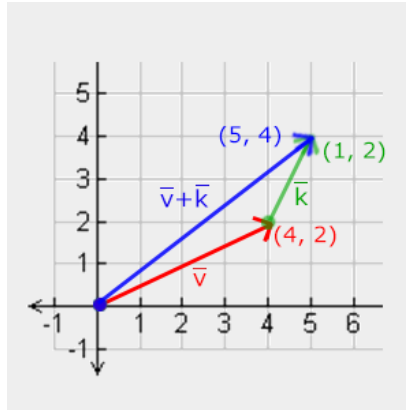
Negating a vector results in a vector in the reversed direction. A vector pointing north-east would point south-west after negation. To negate a vector we add a minus sign to each component (you can also represent it as a scalar-vector multiplication with a scalar value of -1):

$$-\vec{v} = - \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -v_x \\ -v_y \\ -v_z \end{pmatrix}$$

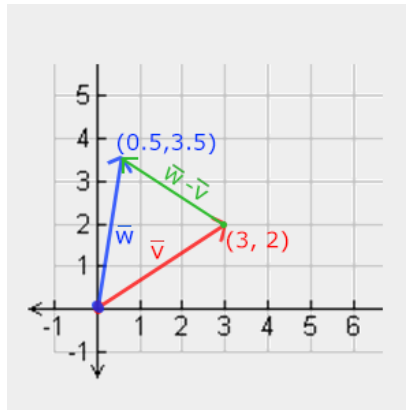
Addition of two vectors is defined as **component-wise** addition, that is, each component of one vector is added to the same component of the other vector like so:

$$\vec{v} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \quad \vec{k} = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \Rightarrow \vec{v} + \vec{k} = \begin{pmatrix} 1+4 \\ 2+5 \\ 3+6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$

Visually, it looks like this on vectors $\vec{v} = (4, 2)$ and $\vec{k} = (1, 2)$, where the second vector is added on top of the first vector's end to find the endpoint of the resulting vector (**head-to-tail method**).



Subtracting two vectors from each other results in a vector that's the difference of the positions both vectors are pointing at.



To retrieve the **length** (or **magnitude**) of a vector we use the **Pythagoras theorem** that you may remember from your math classes. A vector forms a triangle when you visualize its individual x and y components as two sides of a triangle.

Since the length of the two sides (x, y) are known and we want to know the length of the tilted side $\vec{v}$, we can calculate it using the Pythagoras theorem as:

$$\|\vec{v}\| = \sqrt{x^2 + y^2}$$

Where $\|\vec{v}\|$ is denoted as the *length of vector $\vec{v}$* . This is easily extended to 3D by adding z^2 to the equation.

In this case the length of vector $\vec{v} = (4, 2)$ equals:

$$\|\vec{v}\| = \sqrt{4^2 + 2^2} = \sqrt{16 + 4} = \sqrt{20} \approx 4.47$$

Which is 4.47.

There is also a special type of vector that we call a **unit vector**. A unit vector has the extra property that its length is exactly 1. We can calculate a unit vector $\hat{n}$ from any vector by dividing each of the vector's components by its length:

$$\hat{n} = \frac{\vec{v}}{\|\vec{v}\|}$$

We call this **normalizing a vector**. Unit vectors are displayed with a little roof over their head and are generally easier to work with, especially when we only care about their *directions* (the direction does not change if we change a vector's length).

5.1.1 Vector-Vector Multiplication

Multiplying two vectors is not as straightforward as multiplying numbers, since regular multiplication does not carry a direct geometric meaning for vectors. Instead, there are two special cases that are commonly used:

- The **dot product**, denoted as $\vec{v} \cdot \vec{k}$.
- The **cross product**, denoted as $\vec{v} \times \vec{k}$.

The dot product of two vectors is defined as the product of their magnitudes times the cosine of the angle between them:

$$\vec{v} \cdot \vec{k} = \|\vec{v}\| \cdot \|\vec{k}\| \cdot \cos \theta$$

where θ is the angle between $\vec{v}$ and $\vec{k}$.

If $\vec{v}$ and $\vec{k}$ are both **unit vectors** (i.e., $\|\vec{v}\| = \|\vec{k}\| = 1$), this reduces to:

$$\hat{v} \cdot \hat{k} = \cos \theta$$

Thus, the dot product provides a way to measure the angle between two vectors. For example:

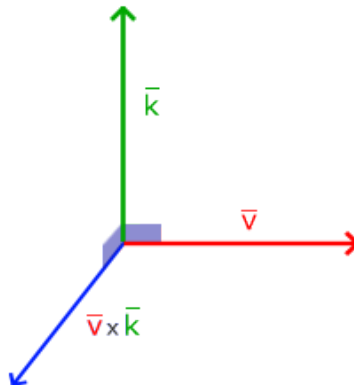
$$\begin{pmatrix} 0.6 \\ -0.8 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = (0.6 \cdot 0) + (-0.8 \cdot 1) + (0 \cdot 0) = -0.8$$

To find the angle θ , we apply the inverse cosine:

$$\theta = \cos^{-1}(-0.8) \approx 143.1^\circ$$

The dot product is very useful in computer graphics and physics, for example in lighting calculations, because it relates vector orientation and angles.

The cross product is defined only in 3D space. It takes two non-parallel vectors and produces a new vector that is orthogonal (perpendicular) to both. Geometrically, the result $\vec{v} \times \vec{k}$ is perpendicular to the plane spanned by $\vec{v}$ and $\vec{k}$.



Given two vectors $A = (A_x, A_y, A_z)$ and $B = (B_x, B_y, B_z)$, the cross product is:

$$A \times B = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix} = \begin{pmatrix} A_y B_z - A_z B_y \\ A_z B_x - A_x B_z \\ A_x B_y - A_y B_x \end{pmatrix}$$

The resulting vector is orthogonal to both A and B . This operation is less intuitive than the dot product, but its usefulness lies in generating perpendicular directions, which play a fundamental role in geometry, physics, and graphics.

5.2 Matrices

We now move from vectors to **matrices**. A matrix is a rectangular array of numbers, symbols, or mathematical expressions. Each entry is called an **element**. For example, a 2×3 matrix is shown below:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

A matrix is indexed by (i, j) , where i is the row index and j is the column index. For instance, the element 4 lies at position $(2, 1)$ (second row, first column). Thus, the above example is called a 2×3 matrix (2 rows, 3 columns).

Matrix addition and subtraction are performed element-wise. That is, corresponding elements from two matrices of the same dimension are added (or subtracted). This means addition/subtraction is only defined for matrices of equal dimensions.

Example of matrix addition:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+6 \\ 3+7 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Example of matrix subtraction:

$$\begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 4-2 & 2-4 \\ 1-0 & 6-1 \end{bmatrix} = \begin{bmatrix} 2 & -2 \\ 1 & 5 \end{bmatrix}$$

A **matrix-scalar product** multiplies each element of a matrix by a scalar:

$$2 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

This shows why numbers are called scalars: they scale every entry of the matrix.

Matrix multiplication is less intuitive than addition but follows specific rules.

1. You can multiply two matrices only if the number of columns of the left matrix equals the number of rows of the right matrix.
2. Matrix multiplication is not commutative: $A \cdot B \neq B \cdot A$.

Example: Multiply two 2×2 matrices.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

To compute each element, take a row from the left matrix and a column from the right matrix, multiply corresponding entries, and sum them up. For example:

- Top-left element: $(1 \cdot 5) + (2 \cdot 7) = 19$
- Bottom-right element: $(3 \cdot 6) + (4 \cdot 8) = 50$

The resulting matrix has dimensions (n, m) where:

- n = number of rows of the left matrix
- m = number of columns of the right matrix

Matrix multiplication is a blend of multiplication and addition applied row-by-column, and it is one of the most fundamental operations in linear algebra.

5.2.1 Matrix-Vector Multiplication

So far, we have worked with vectors to represent positions, colors, and texture coordinates. A vector can be thought of as a special case of a matrix: an $N \times 1$ matrix, where N is the number of components. Thus, vectors are essentially matrices with one column.

If we have an $M \times N$ matrix and a corresponding $N \times 1$ vector, then matrix-vector multiplication is well defined. This operation is extremely useful since many geometric transformations (2D and 3D) can be represented as matrix multiplications.

In OpenGL and many other applications, we often use 4×4 transformation matrices. The simplest of these is the **identity matrix**, which is an $N \times N$ matrix containing 1s on the diagonal and 0s elsewhere:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When multiplied with any vector, it leaves the vector unchanged:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

This happens because each row of the identity matrix isolates exactly one component of the vector. Although the identity matrix does not transform a vector, it serves as the foundation for building more complex transformation matrices.

We can express scaling as a matrix transformation. Starting from the identity matrix, we replace the diagonal entries with the scaling factors (S_x, S_y, S_z):

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying this to a vector $(x, y, z, 1)$:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x \cdot x \\ S_y \cdot y \\ S_z \cdot z \\ 1 \end{bmatrix}$$

Here, the fourth component (the w coordinate) is kept as 1, since it serves other purposes in homogeneous coordinates. This scaling matrix provides a compact and systematic way of enlarging or shrinking vectors along each axis independently.

Translation is the process of shifting a vector to a new position by adding another vector to it. In other words, we move the vector based on a translation vector. This is conceptually the same as vector addition, but applied in matrix form.

Just like scaling, translation can be expressed using a 4×4 transformation matrix. The top three entries of the fourth column represent the translation vector (T_x, T_y, T_z) . The translation matrix is defined as:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{bmatrix}$$

This works because the translation values are multiplied by the w component of the vector (which is usually 1) and added to the original coordinates (x, y, z) . Notice that this requires a 4×4 matrix; a 3×3 matrix would not suffice.

The w component of a vector is called the **homogeneous coordinate**. To retrieve the original 3D vector, we divide (x, y, z) by w . In practice, $w = 1$ in most cases, so this division is invisible.

Advantages of homogeneous coordinates:

- They allow us to apply translations using matrix multiplication (without w , translation is not possible).
- They provide a unified way to represent transformations (scaling, rotation, translation, perspective).

A special case occurs when $w = 0$: in this case, the vector is called a **direction vector**, because it cannot be translated (only positioned vectors with $w = 1$ can be moved).

The previous transformations (scaling, translation) were relatively easy to understand. Rotation, however, is a bit more involved, especially in 3D. A **rotation** is defined by an angle. The angle may be measured in degrees (360° for a full circle) or in radians (2π for a full circle).

Conversion between degrees and radians:

$$\text{angle in degrees} = \text{angle in radians} \cdot \frac{180}{\pi}, \quad \text{angle in radians} = \text{angle in degrees} \cdot \frac{\pi}{180}$$

In 3D, a rotation is specified by:

- an **angle** θ , and
- a **rotation axis**.

For instance, rotating around the z -axis in 3D has the same effect as rotating in 2D. Rotations are expressed using a combination of trigonometric functions $\sin$ and $\cos$.

Each principal axis has its own rotation matrix. Given a vector $(x, y, z, 1)^T$, the rotations are as follows:

Rotation around the x -axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{bmatrix}$$

Rotation around the y -axis:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta \cdot x + \sin \theta \cdot z \\ y \\ -\sin \theta \cdot x + \cos \theta \cdot z \\ 1 \end{bmatrix}$$

Rotation around the z -axis:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta \cdot x - \sin \theta \cdot y \\ \sin \theta \cdot x + \cos \theta \cdot y \\ z \\ 1 \end{bmatrix}$$

To rotate around an arbitrary axis $\vec{R} = (R_x, R_y, R_z)$, normalized to unit length, we use a general rotation matrix:

$$\begin{bmatrix} \cos \theta + R_x^2(1 - \cos \theta) & R_x R_y(1 - \cos \theta) - R_z \sin \theta & R_x R_z(1 - \cos \theta) + R_y \sin \theta & 0 \\ R_y R_x(1 - \cos \theta) + R_z \sin \theta & \cos \theta + R_y^2(1 - \cos \theta) & R_y R_z(1 - \cos \theta) - R_x \sin \theta & 0 \\ R_z R_x(1 - \cos \theta) - R_y \sin \theta & R_z R_y(1 - \cos \theta) + R_x \sin \theta & \cos \theta + R_z^2(1 - \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This formula allows for rotation about any axis, not just the x , y , or z axes.

When combining rotations around multiple axes, one can encounter **Gimbal lock**, where degrees of freedom are lost. A more robust solution is to use **quaternions**, which avoid gimbal lock and are computationally efficient. However, the discussion of quaternions lies outside the scope of this chapter.

5.2.2 Combining Matrices

The real power of using matrices for transformations is that multiple transformations can be combined into a single matrix through **matrix-matrix multiplication**.

Suppose we have a vector (x, y, z) that we want to scale by a factor of 2, and then translate by $(1, 2, 3)$. We need a scaling matrix and a translation matrix, and then multiply them together.

The combined transformation matrix is:

$$\text{Trans} \cdot \text{Scale} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that the order of multiplication matters. Matrix multiplication is **not commutative**, so the result of translation followed by scaling is different from scaling followed by translation.

- The right-most matrix is applied first.
- It is generally best to apply transformations in the order: **scale** → **rotate** → **translate**.

Otherwise, transformations can interfere. For instance, if translation comes before scaling, the translation vector itself will also be scaled.

Applying the combined matrix to a vector $(x, y, z, 1)^T$:

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 2x + 1 \\ 2y + 2 \\ 2z + 3 \\ 1 \end{bmatrix}$$

5.3 GLM

GLM stands for OpenGL Mathematics and is a header-only library, which means that we only have to include the proper header files and we're done; no linking and compiling necessary. GLM can be downloaded from their website <https://github.com/g-truc/glm/tags>. Unzip the file into your project directory and update the CMakeLists.txt file.

```
1 include_directories(${CMAKE_SOURCE_DIR}/glfw-3.4.bin.MACOS/include
2   ${CMAKE_SOURCE_DIR}/glad/include
3   include
4   ${CMAKE_SOURCE_DIR}/glm-1.0.1/glm)
```

The most popular header files we will use:

```
1 #include <glm.hpp>
2 #include <gtc/matrix_transform.hpp>
3 #include <gtc/type_ptr.hpp>
```

Let's see if we can put our transformation knowledge to good use by translating a vector of (1,0,0) by (1,1,0) (note that we define it as a `glm::vec4` with its homogeneous coordinate set to 1.0):

```
1 glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
2 glm::mat4 trans = glm::mat4(1.0f);
3 trans = glm::translate(trans, glm::vec3(1.0f, 1.0f, 0.0f));
4 vec = trans * vec;
5 std::cout << vec.x << " " << vec.y << " " << vec.z << std::endl;
```

We first define a vector named `vec` using GLM's built-in vector class. Next we define a `mat4` and explicitly initialize it to the identity matrix by initializing the matrix's diagonals to 1.0; if we do not initialize it to the identity matrix the matrix would be a null matrix (all elements 0) and all subsequent matrix operations would end up a null matrix as well.

The next step is to create a transformation matrix by passing our identity matrix to the `glm::translate` function, together with a translation vector (**the given matrix is then multiplied with a translation matrix and the resulting matrix is returned**).

Then we multiply our vector by the transformation matrix and output the result. If we still remember how matrix translation works then the resulting vector should be (1+1,0+1,0+0) which is (2,1,0). This snippet of code outputs 2 1 0 so the translation matrix did its job.

Let's do something more interesting and scale and rotate the container object from the previous sections:

```
1 glm::mat4 trans = glm::mat4(1.0f);
2 trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
3 trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
```

First we scale the container by 0.5 on each axis and then rotate the container 90 degrees around the Z-axis. GLM expects its angles in radians so we convert the degrees to radians using `glm::radians`. Note that the textured rectangle is on the XY plane so we want to rotate around the Z-axis. Keep in mind that the axis that we rotate around should be a unit vector, so be sure to normalize the vector first if you're not rotating around the X, Y, or Z axis. Because we pass the matrix to each of GLM's functions, GLM automatically multiplies the matrices together, resulting in a transformation matrix that combines all the transformations.

We'll adapt the vertex shader to accept a mat4 uniform variable and multiply the position vector by the matrix uniform:

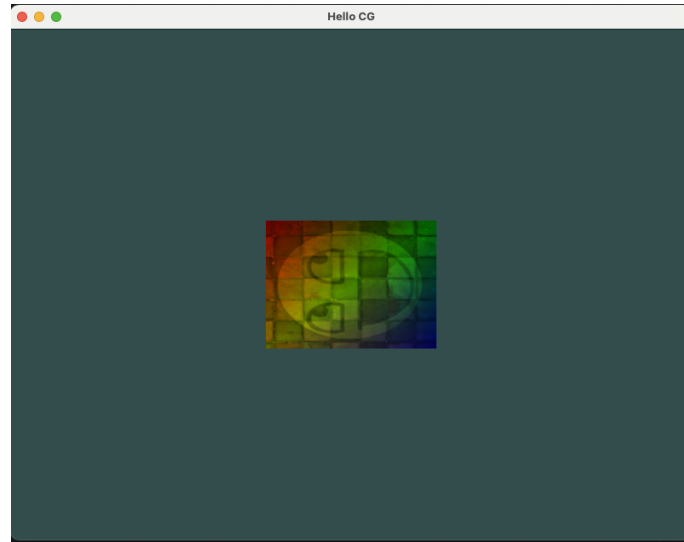
```
1 #version 410 core
2 in vec3 aPos;
3 in vec2 aTexCoord;
4
5 out vec2 TexCoord;
6
7 uniform mat4 transform;
8
9 void main()
10 {
11     gl_Position = transform * vec4(aPos, 1.0f);
12     TexCoord = vec2(aTexCoord.x, aTexCoord.y);
13 }
```

Our container should now be twice as small and rotated 90 degrees (tilted to the left). We still need to pass the transformation matrix to the shader though:

```
1 glm::mat4 trans = glm::mat4(1.0f);
2 trans = glm::rotate(trans, glm::radians(90.0f), glm::vec3(0.0, 0.0, 1.0));
3 trans = glm::scale(trans, glm::vec3(0.5, 0.5, 0.5));
4 shaderProgram.setUniform("transform", trans);
```

We first query the location of the uniform variable and then send the matrix data to the shaders using `glUniform` with `Matrix4fv` as its postfix.

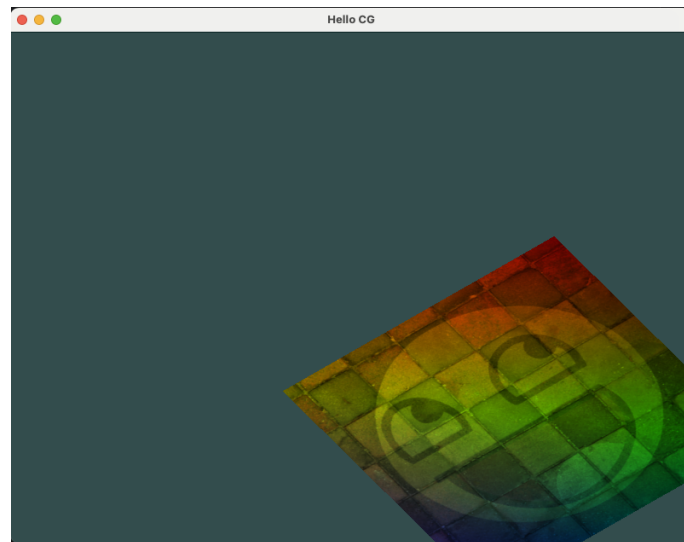
- The first argument should be familiar by now which is the uniform's location.
- The second argument tells OpenGL how many matrices we'd like to send, which is 1.
- The third argument asks us if we want to transpose our matrix, that is to swap the columns and rows. OpenGL developers often use an internal matrix layout called column-major ordering which is the default matrix layout in GLM so there is no need to transpose the matrices; we can keep it at `GL_FALSE`.
- The last parameter is the actual matrix data, but GLM stores their matrices' data in a way that doesn't always match OpenGL's expectations so we first convert the data with GLM's built-in function `value_ptr`.



To rotate the container over time we have to update the transformation matrix in the render loop because it needs to update each frame. We use GLFW's time function to get an angle over time:

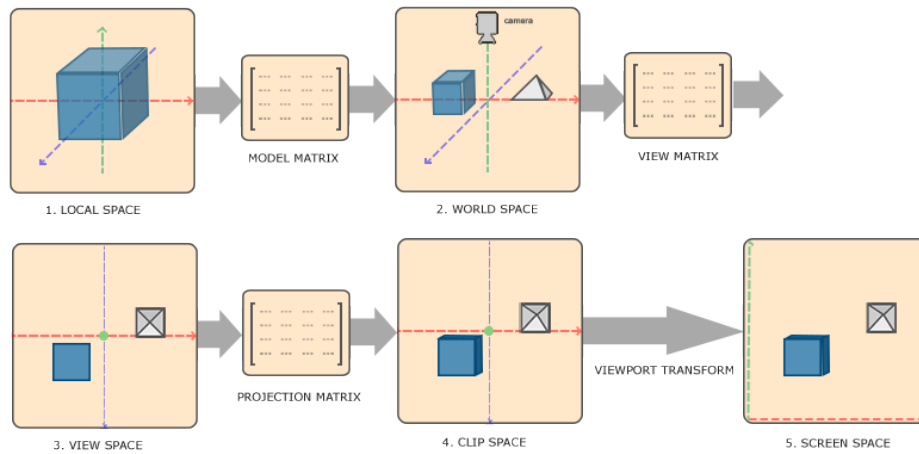
```
1 glm::mat4 trans = glm::mat4(1.0f);
2 trans = glm::translate(trans, glm::vec3(0.5f, -0.5f, 0.0f));
3 trans = glm::rotate(trans, (float)glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
```

Here we first rotate the container around the origin (0,0,0) and once it's rotated, we translate its rotated version to the bottom-right corner of the screen. **Remember that the actual transformation order should be read in reverse: even though in code we first translate and then later rotate, the actual transformations first apply a rotation and then a translation.**



5.4 Coordinate Systems

To transform the coordinates from one space to the next coordinate space we'll use several transformation matrices of which the most important are the model, view and projection matrix. Our vertex coordinates first start in local space as local coordinates and are then further processed to world coordinates, view coordinates, clip coordinates and eventually end up as screen coordinates. The following image displays the process and shows what each transformation does:

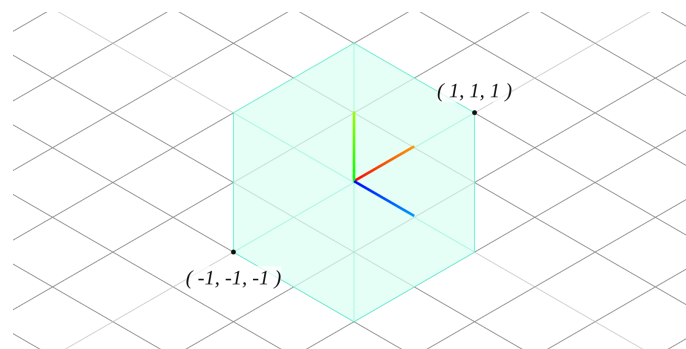


- Local coordinates are the coordinates of your object relative to its local origin; they're the coordinates your object begins in.
- The next step is to transform the local coordinates to world-space coordinates which are coordinates in respect of a larger world. These coordinates are relative to some global origin of the world, together with many other objects also placed relative to this world's origin.
- Next we transform the world coordinates to view-space coordinates in such a way that each coordinate is as seen from the camera or viewer's point of view.
- After the coordinates are in view space we want to project them to clip coordinates. Clip coordinates are processed to the -1.0 and 1.0 range and determine which vertices will end up on the screen. Projection to clip-space coordinates can add perspective if using perspective projection.
- And lastly we transform the clip coordinates to screen coordinates in a process we call viewport transform that transforms the coordinates from -1.0 and 1.0 to the coordinate range defined by glViewport. The resulting coordinates are then sent to the rasterizer to turn them into fragments.

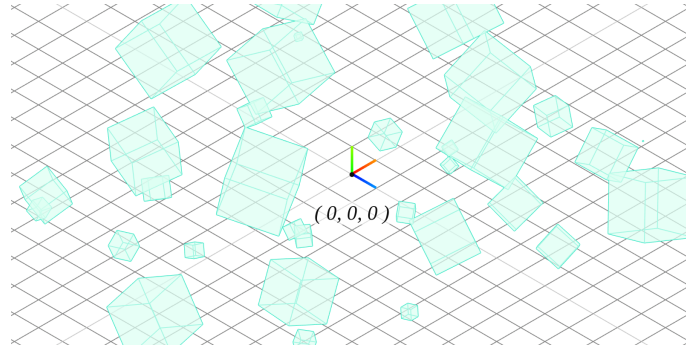
5.4.1 From Local Space to World Space

Local space is the coordinate space that is local to your object, i.e. where your object begins in. Imagine that you've created your cube in a modeling software package (like Blender). The origin of your cube is probably at (0,0,0) even though your cube may end up at a different location in your final application. Probably all the models you've created all have (0,0,0) as their initial position. All the vertices of your model are therefore in local space: they are all local to your object.

The vertices of the container we've been using were specified as coordinates between -0.5 and 0.5 with 0.0 as its origin. These are local coordinates.



The model matrix transforms vertices from model space to world space. A 2x2x2 cube centered at (0,0,0) can be resized, twisted and placed anywhere when combined with a model matrix.



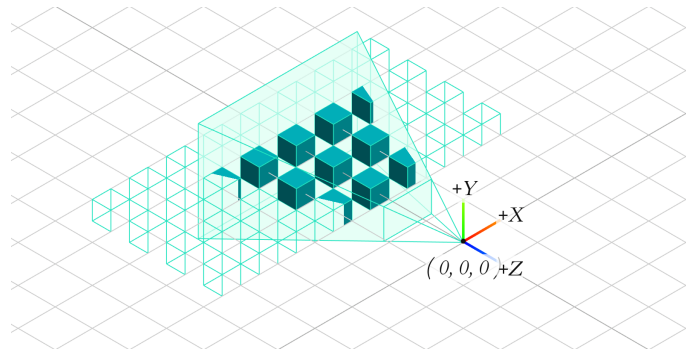
A model matrix M is composed of an object's translation transform T , rotation transform R , and scale transform S . Multiplying a vertex position v_{model} by this model matrix transforms the vector into world space:

$$M = T \cdot R \cdot S$$

$$v_{\text{world}} = M \cdot v_{\text{model}}$$

5.4.2 From World Space to View Space

As all renders are from some camera's perspective, all vertices must be defined relative to the camera. View space is the coordinate system defined as the camera at $(0,0,0)$, facing down its $-Z$ axis. The camera also has a model matrix defining its position in world space. The inverse of the camera's model matrix is the view matrix, and it transforms vertices from world space to view space.



The view matrix is shared by all objects in the scene, since they are all rendered from the same camera. Given a camera's model matrix C , any vector v can be transformed from model space, to world space, and then to view space:

$$V = C^{-1}$$

$$v_{\text{camera}} = V \cdot M \cdot v_{\text{model}}$$

In an OpenGL system where the camera faces down the $-Z$ axis, any vertex that will be rendered must lie in front of the camera. Thus, in camera space, the vertex will have a negative Z value.

A camera can be conceptually thought of as just another object in the world — it has:

- A **position** (translation)
- An **orientation** (rotation)

If the camera were a visible object, its model matrix M_{camera} would transform it from the camera's *local coordinate system* to the *world coordinate system*:

$$\text{world position of a camera point} = M_{\text{camera}} \cdot \text{camera local position}$$

However, when rendering, we don't move the camera itself — instead, we move the world in the opposite direction of the camera.

We want to transform all vertices from *world space* into the *camera's local space*, i.e., what the camera “sees.”

Thus, we need the reverse of what the camera's model matrix does:

$$V = M_{\text{camera}}^{-1}$$

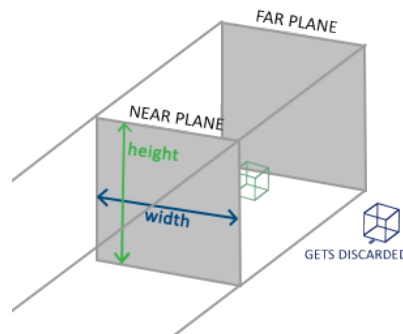
That's why the view matrix is the inverse of the camera's model matrix.

5.4.3 From View Space to Clip Space

Once vertices are in camera space, they can finally be transformed into clip space by applying a projection transformation. The projection matrix encodes how much of the scene is captured in a render by defining the extents of the camera's view. The two most common types of projection are perspective and orthographic.

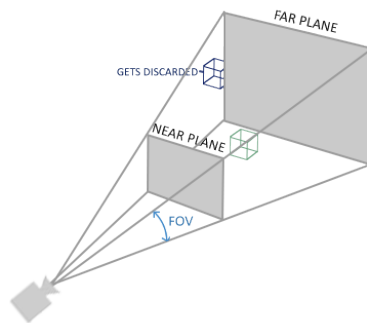
This viewing box a projection matrix creates is called a **frustum** and each coordinate that ends up inside this frustum will end up on the user's screen.

An **orthographic projection matrix** defines a cube-like frustum box that defines the clipping space where each vertex outside this box is clipped. When creating an orthographic projection matrix we specify the width, height and length of the visible frustum. All the coordinates inside this frustum will end up within the NDC range after transformed by its matrix and thus won't be clipped. The frustum looks a bit like a container:



The frustum defines the visible coordinates and is specified by a width, a height and a near and far plane. Any coordinate in front of the near plane is clipped and the same applies to coordinates behind the far plane. The orthographic frustum directly maps all coordinates inside the frustum to normalized device coordinates without any special side effects since it won't touch the w component of the transformed vector; if the w component remains equal to 1.0 perspective division won't change the coordinates. But in reality a direct projection produces unrealistic results since the projection doesn't take perspective into account.

If you ever were to enjoy the graphics the real life has to offer you'll notice that objects that are farther away appear much smaller. This weird effect is something we call **perspective**. The **projection matrix** maps a given frustum range to clip space, but also manipulates the w value of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes. Once the coordinates are transformed to clip space they are in the range -w to w (anything outside this range is clipped).



5.4.4 From Clip Space to Normalize Device Space

OpenGL requires that the visible coordinates fall within the range $[-1.0, 1.0]$ as the final vertex shader output. Thus, once the coordinates are in clip space, **perspective division** is applied to the clip space coordinates:

$$\text{out} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}$$

Each component of the vertex coordinate is divided by its w component, causing vertices that are further away from the viewer to have smaller normalized coordinates. This is one of the key roles of the w component, as it enables perspective projection. The resulting coordinates are then in **normalized device space**.

5.4.5 Putting It All Together

We create a transformation matrix for each of the aforementioned steps: the model matrix, the view matrix, and the projection matrix. A vertex coordinate is then transformed to clip coordinates as follows:

$$V_{\text{clip}} = M_{\text{projection}} \cdot M_{\text{view}} \cdot M_{\text{model}} \cdot V_{\text{local}}$$

Note that the order of matrix multiplication is reversed (remember that we read matrix multiplication from right to left). The resulting vertex should then be assigned to `gl_Position` in the vertex shader, after which OpenGL will automatically perform perspective division and clipping.

Next, OpenGL uses the parameters from `glViewport` to map the normalized device coordinates to **screen coordinates**, where each coordinate corresponds to a point on the screen (in our case an 800×600 window). This process is called the **viewport transform**.

5.4.6 An Example

To start drawing in the window we'll first create a model matrix. The model matrix consists of translations, scaling and/or rotations we'd like to apply to transform all object's vertices to the global world space. Let's transform our plane a bit by rotating it on the x-axis so it looks like it's laying on the floor. The model matrix then looks like this:

```
1 glm::mat4 model = glm::mat4(1.0f);
2 model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
```

By multiplying the vertex coordinates with this model matrix we're transforming the vertex coordinates to world coordinates. Our plane that is slightly on the floor thus represents the plane in the global world.

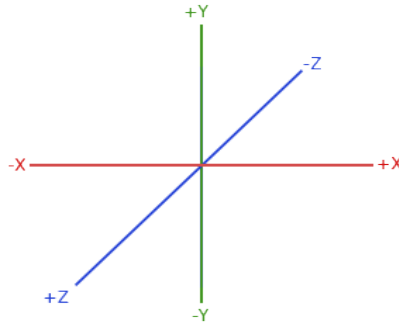
Next we need to create a view matrix. We want to move slightly backwards in the scene so the object becomes visible (when in world space we're located at the origin (0,0,0)). To move around the scene, think about the following:

To move a camera backwards, is the same as moving the entire scene forward.

That is exactly what a view matrix does, we move the entire scene around inversed to where we want the camera to move.

Because we want to move backwards and since OpenGL is a right-handed system we have to move in the positive z-axis. We do this by translating the scene towards the negative z-axis. This gives the impression that we are moving backwards.

By convention, OpenGL is a right-handed system. What this basically says is that the positive x-axis is to your right, the positive y-axis is up and the positive z-axis is backwards. Think of your screen being the center of the 3 axes and the positive z-axis going through your screen towards you. The axes are drawn as follows:



For now the view matrix looks like this:

```
1 glm::mat4 view = glm::mat4(1.0f);
2 // note that we're translating the scene in the reverse direction of where we want to move
3 view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
```

The last thing we need to define is the projection matrix. We want to use perspective projection for our scene so we'll declare the projection matrix like this:

```
1 glm::mat4 projection;
2 projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
```

Now that we have created the transformation matrices, the next step is to pass them to our shaders. The Model-View-Projection (MVP) matrix can be computed either on the CPU side in the OpenGL program (C++/Python/etc.) or on the GPU side within the vertex shader. In our earlier approach, we calculated the MVP on the CPU, but this becomes inefficient for large scenes. Since each vertex can be processed independently, the computation is highly parallelizable, making the GPU the more suitable and efficient choice.

Let's declare the transformation matrices as uniforms in the vertex shader and multiply them with the vertex coordinates because every vertex will share the transformation matrices that means we only need one copy:

```
1 ...
2 uniform mat4 model;
3 uniform mat4 view;
4 uniform mat4 projection;
5
6 void main(){
7     // note that we read the multiplication from right to left
8     gl_Position = projection * view * model * vec4(aPos, 1.0);
9     ...
10 }
```

The updated main rendering loop should be:

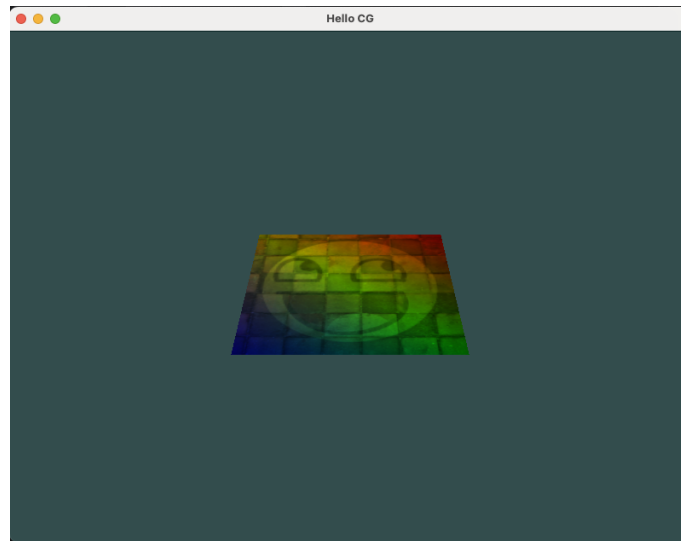
```
1 while (!glfwWindowShouldClose(window)) {
2     if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
3         glfwSetWindowShouldClose(window, true);
4     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
5     glClear(GL_COLOR_BUFFER_BIT);
6
7     //enable VAO that stores the specific model's info
8     shaderProgram.use();
9     shaderProgram.bindTexture2D("texture1", std::string(ASSETS_DIR) + "wall.jpg", 0, false);
10    shaderProgram.bindTexture2D("texture2", std::string(ASSETS_DIR) + "awesomeface.png", 1, true);
11    glm::mat4 model = glm::mat4(1.0f);
12    model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));
13    glm::mat4 view = glm::mat4(1.0f);
14    // note that we're translating the scene in the reverse direction of where we want to move
```

```

15     view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
16     glm::mat4 projection;
17     projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
18     shaderProgram.setUniform("model", model);
19     shaderProgram.setUniform("view", view);
20     shaderProgram.setUniform("projection", projection);
21
22     quad.draw();
23
24     glfwSwapBuffers(window);
25     glfwPollEvents();
26 }

```

The output should be:



5.5 Going 3D

2D shapes are not fun anymore; let's render a 3D model instead of a 2D shape. Well, 3D model means a lot more vertices than a 2D shape. Instead of writing the vertices and indices data in the main function, we should write a function such that we can load the model from the external file.

5.5.1 Tinyobjloader Library

tinyobjloader is a lightweight, single-header C++ library for loading 3D models stored in the Wavefront .obj format. Designed to be simple and dependency-free, it efficiently parses vertex positions, normals, texture coordinates, materials, and groups from OBJ and MTL files into easy-to-use data structures. Because it's header-only, **tinyobjloader** can be integrated into any OpenGL or graphics project with minimal setup—just include the header, define one implementation macro, and call its API to read model data directly into your own rendering pipeline. Put the header file `tiny_obj_loader.h` in your include folder, and create an executable file `tiny_obj_loader.cpp` in your src folder, and put the following code into it:

```

1  #define TINYOBJLOADER_IMPLEMENTATION
2  #include "tiny_obj_loader.h"

```

Remember to update your CMakeLists.txt file:

```

1  add_executable(demo main.cpp
2      src/shaderprogram.cpp
3      src/stb_image.cpp
4      src/mesh.cpp
5      src/tiny_obj_loader.cpp)

```

Let's define a function to load obj file in Mesh class:

```
1 bool loadOBJ_(const std::string& path);
```

Then let's do corresponding modification in the source file:

```
1 bool Mesh::loadOBJ_(const std::string& path)
2 {
3     vertices.clear();
4     indices.clear();
5
6     tinyobj::ObjReaderConfig config;
7     config.triangulate = true;    // let tinyobj make all faces triangles
8     config.vertex_color = false;
9
10    tinyobj::ObjReader reader;
11    if (!reader.ParseFromFile(path, config)) {
12        if (!reader.Error().empty())
13            std::cerr << "tinyobj error: " << reader.Error() << "\n";
14        return false;
15    }
16    if (!reader.Warning().empty()) {
17        std::cerr << "tinyobj warning: " << reader.Warning() << "\n";
18    }
19
20    const tinyobj::attrib_t& attrib = reader.GetAttrib();
21    const std::vector<tinyobj::shape_t>& shapes = reader.GetShapes();
22
23    auto getPos = [&](int vi) -> glm::vec3 {
24        const float* v = attrib.vertices.data() + 3 * vi;
25        return glm::vec3(v[0], v[1], v[2]);
26    };
27
28    auto getUV = [&](int ti) -> glm::vec2 {
29        const float* t = attrib.texcoords.data() + 2 * ti;
30        return glm::vec2(t[0], t[1]);
31    };
32
33    uint32_t baseIndex = 0;
34
35    for (const auto& shape : shapes) {
36        const auto& mesh = shape.mesh;
37
38        size_t index_offset = 0;
39        for (size_t f = 0; f < mesh.num_face_vertices.size(); ++f) {
40            int fv = mesh.num_face_vertices[f];
41            if (fv != 3) { // should be 3 due to triangulate=true
42                index_offset += fv;
43                continue;
44            }
45
46            // emit 3 vertices (pos + uv only)
47            for (int v = 0; v < 3; ++v) {
48                const tinyobj::index_t idx = mesh.indices[index_offset + v];
49
50                // position
51                const glm::vec3 p = getPos(idx.vertex_index);
52                vertices.push_back(p.x);
53                vertices.push_back(p.y);
54                vertices.push_back(p.z);
55
56                // uv (default 0,0 if missing)
57                float u = 0.0f, vv = 0.0f;
58                if (idx.texcoord_index >= 0 &&
59                    static_cast<size_t>(2 * idx.texcoord_index + 1) < attrib.texcoords.size())
60                {
```

```

61         const glm::vec2 uv = getUV(idx.texcoord_index);
62         u = uv.x; vv = uv.y;
63     }
64     vertices.push_back(u);
65     vertices.push_back(vv);
66
67     indices.push_back(baseIndex++);
68 }
69
70     index_offset += fv;
71 }
72 }
73
74     return !vertices.empty() && !indices.empty();
75 }

```

Because load model should be one step of the Mesh object initialization process, let's define a different Mesh constructor in mesh.h:

```

1 Mesh(const std::string& path, GLuint id);

```

and implement it in mesh.cpp:

```

1 Mesh::Mesh(const std::string& path, GLuint id){
2     shaderProgramID = id;
3
4     loadOBJ_(path);
5     createBuffers_();
6 }

```

Modify our createBuffers_ function because we do not have per-vertex color anymore:

```

1 void Mesh::createBuffers_()
2 {
3
4     const GLint posLoc = glGetAttribLocation(shaderProgramID, "aPos");
5     const GLint uvLoc = glGetAttribLocation(shaderProgramID, "aTexCoord");
6
7     GLsizei stride = 5 * sizeof(float);
8     std::vector<VertexAttribute> attributes = {
9         VertexAttribute{ static_cast<GLuint>(posLoc), 3, GL_FLOAT, GL_FALSE, stride, 0 },
10        VertexAttribute{ static_cast<GLuint>(uvLoc), 2, GL_FLOAT, GL_FALSE, stride, 3 * sizeof(float) } };
11    ... ...
12 }

```

Now download the obj file box.obj from Canvas, modify our main function a little bit:

```

1 Mesh box(std::string(ASSETS_DIR) + "box.obj", shaderProgram.getID());
2
3 while (!glfwWindowShouldClose(window)) {
4     if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
5         glfwSetWindowShouldClose(window, true);
6     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
7     glClear(GL_COLOR_BUFFER_BIT);
8
9     //enable VAO that stores the specific model's info
10    shaderProgram.use();
11    shaderProgram.bindTexture2D("texture1", std::string(ASSETS_DIR) + "wall.jpg", 0, false);
12    shaderProgram.bindTexture2D("texture2", std::string(ASSETS_DIR) + "awesomeface.png", 1, true);

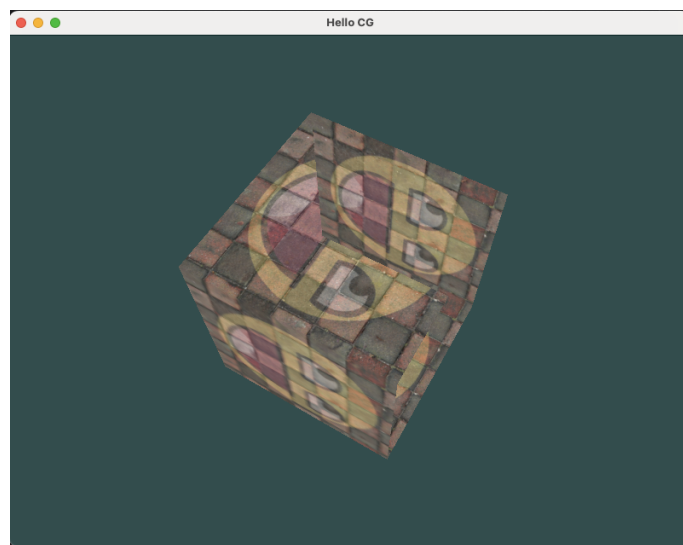
```

```

13
14     glm::mat4 model = glm::mat4(1.0f);
15     model = glm::rotate(model, (float)glfwGetTime() * glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));
16     glm::mat4 view = glm::mat4(1.0f);
17     view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
18     glm::mat4 projection;
19     projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
20
21     shaderProgram.setUniform("model", model);
22     shaderProgram.setUniform("view", view);
23     shaderProgram.setUniform("projection", projection);
24
25     box.draw();
26
27     glfwSwapBuffers(window);
28     glfwPollEvents();
29 }

```

You should get a similar result below:



5.5.2 Z-buffer

OpenGL stores all its depth information in a z-buffer, also known as a depth buffer. GLFW automatically creates such a buffer for you. The depth is stored within each fragment (as the fragment's z value) and whenever the fragment wants to output its color, OpenGL compares its depth values with the z-buffer. If the current fragment is behind the other fragment it is discarded, otherwise overwritten. This process is called depth testing and is done automatically by OpenGL.

However, if we want to make sure OpenGL actually performs the depth testing we first need to tell OpenGL we want to enable depth testing; it is **disabled by default**. We can enable depth testing using `glEnable`. The `glEnable` and `glDisable` functions allow us to enable/disable certain functionality in OpenGL. That functionality is then enabled/disabled until another call is made to disable/enable it. Right now we want to enable depth testing by enabling `GL_DEPTH_TEST`:

```

1     glEnable(GL_DEPTH_TEST);

```

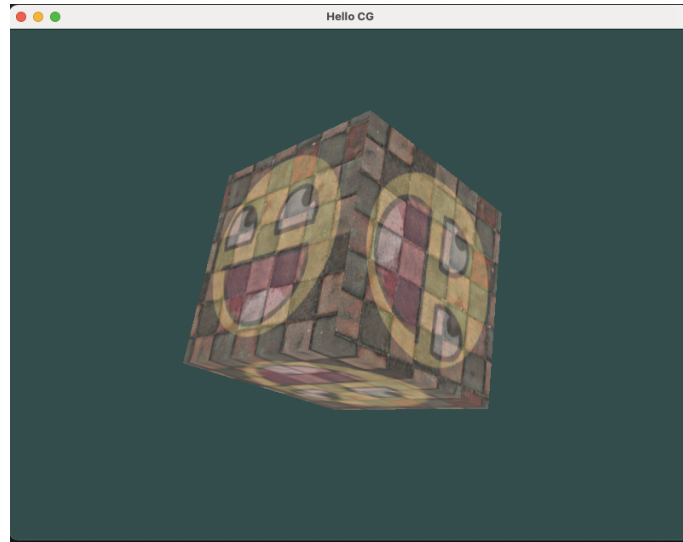
Since we're using a depth buffer we also want to clear the depth buffer before each render iteration (otherwise the depth information of the previous frame stays in the buffer). Just like clearing the color buffer, we can clear the depth buffer by specifying the `DEPTH_BUFFER_BIT` bit in the `glClear` function:

```

1     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

```

Let's re-run our program and see if OpenGL now performs depth testing:



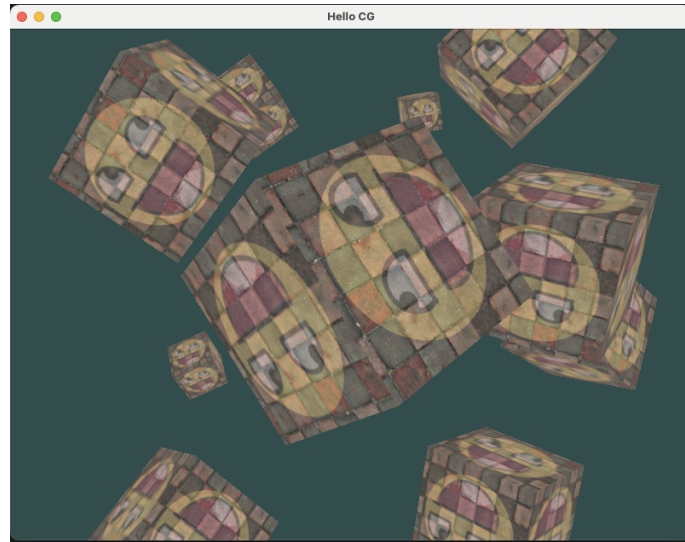
One more interesting thing is we can reuse the same model multiple times in a single frame. Lets render the box for 10 times and translate it into different positions:

```

1  glm::vec3 cubePositions[] = {
2      glm::vec3( 0.0f,  0.0f,  0.0f),
3      glm::vec3( 2.0f,  5.0f, -15.0f),
4      glm::vec3(-1.5f, -2.2f, -2.5f),
5      glm::vec3(-3.8f, -2.0f, -12.3f),
6      glm::vec3( 2.4f, -0.4f, -3.5f),
7      glm::vec3(-1.7f,  3.0f, -7.5f),
8      glm::vec3( 1.3f, -2.0f, -2.5f),
9      glm::vec3( 1.5f,  2.0f, -2.5f),
10     glm::vec3( 1.5f,  0.2f, -1.5f),
11     glm::vec3(-1.3f,  1.0f, -1.5f)
12 };
13
14 while (!glfwWindowShouldClose(window)) {
15     if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
16         glfwSetWindowShouldClose(window, true);
17     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
18     glEnable(GL_DEPTH_TEST);
19     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
20     //enable VAO that stores the specific model's info
21     shaderProgram.use();
22     shaderProgram.bindTexture2D("texture1", std::string(ASSETS_DIR) + "wall.jpg", 0, false);
23     shaderProgram.bindTexture2D("texture2", std::string(ASSETS_DIR) + "awesomeface.png", 1, true);
24
25     for(unsigned int i = 1; i < 11; i++) {
26         glm::mat4 model = glm::mat4(1.0f);
27         model = glm::translate(model, cubePositions[i-1]);
28         model = glm::rotate(model, (float) glfwGetTime() * glm::radians(20.0f * i), glm::vec3(1.0f, 0.3f, 0.5f));
29         glm::mat4 view = glm::mat4(1.0f);
30         view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
31         glm::mat4 projection;
32         projection = glm::perspective(glm::radians(45.0f), 800.0f / 600.0f, 0.1f, 100.0f);
33         shaderProgram.setUniform("model", model);
34         shaderProgram.setUniform("view", view);
35         shaderProgram.setUniform("projection", projection);
36
37         box.draw();
38     }

```

Now you will get 10 randomly rotated boxes:



5.6 Cameras

In this chapter we'll discuss how we can set up a camera in OpenGL. We will discuss a fly style camera that allows you to freely move around in a 3D scene. We'll also discuss keyboard and mouse input and finish with a custom camera class.

When we're talking about camera/view space we're talking about all the vertex coordinates as seen from the camera's perspective as the origin of the scene: the view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction. To define a camera we need:

- its position in world space
- the direction it's looking at
- a vector pointing to the right
- a vector pointing upwards from the camera

A careful reader may notice that we're actually going to create a coordinate system with 3 perpendicular unit axes with the camera's position as the origin.

5.6.1 Camera Position

Getting the camera position is easy. The camera position is a vector in world space that points to the camera's position. We set the camera at the same position we've set the camera in the previous chapter:

```
1 glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
```

5.6.2 Camera Direction

The next vector we need is the camera's direction, that is, the direction in which it is pointing. For now, let's assume the camera points toward the origin of our scene at (0, 0, 0). Recall that subtracting one vector from another gives the difference between them. If we subtract the camera's position vector from the origin vector, we obtain the direction vector we want. However, in the view matrix's coordinate system, we define the z-axis as positive, while in OpenGL the camera is conventionally oriented toward the negative z-axis. Therefore, we must negate this direction vector.

```
1 glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.0f);
2 glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);
```

5.6.3 Right Axis

The next vector we need is the right vector, which defines the positive x-axis of the camera space. To calculate it, we start by defining an up vector that points upward in world space. We then take the cross product of this up vector with the direction vector from step 2. Because the cross product produces a vector perpendicular to both inputs, the

result gives us the right vector aligned with the positive x-axis. Note that if we were to reverse the order of the cross product, the resulting vector would instead point along the negative x-axis.

```
1 glm::vec3 up = glm::vec3(0.0f, 1.0f, 0.0f);
2 glm::vec3 cameraRight = glm::normalize(glm::cross(up, cameraDirection));
```

5.6.4 Up axis

Now that we have both the x-axis vector and the z-axis vector, retrieving the vector that points to the camera's positive y-axis is relatively easy: we take the cross product of the right and direction vector:

```
1 glm::vec3 cameraUp = glm::cross(cameraDirection, cameraRight);
```

5.6.5 Why do we recompute the Up vector?

Even though we already have a global/world *up axis* (e.g., $(0, 1, 0)$), this axis does not necessarily align with the camera's orientation. If we directly used the world up as the camera's up vector, the result could be "tilted" or inconsistent, especially when the camera rotates or moves.

To avoid this, we proceed as follows:

1. Start with a fixed world up vector to serve as a reference:

$$\mathbf{up} = (0, 1, 0).$$

2. Compute the right vector using the cross product of the world up and the camera's direction:

$$\mathbf{cameraRight} = \frac{\mathbf{up} \times \mathbf{cameraDirection}}{\|\mathbf{up} \times \mathbf{cameraDirection}\|}.$$

This ensures the right vector is perpendicular to both the camera's forward direction and the global up. If the order of the cross product is reversed, the resulting vector points along the negative x -axis.

3. Recompute the camera's up vector as the cross product of the camera's direction and the right vector:

$$\mathbf{cameraUp} = \mathbf{cameraDirection} \times \mathbf{cameraRight}.$$

This guarantees that **cameraUp** is truly orthogonal to both **cameraDirection** and **cameraRight**.

This procedure ensures that the three basis vectors (direction, right, up) are mutually perpendicular, forming an orthonormal basis for the camera space. Mathematically, this is an application of the **Gram-Schmidt orthogonalization process**.

Giving an example to demonstrate the process: Now consider

$$\mathbf{cameraPos} = (0, 2, 3), \quad \mathbf{target} = (0, 0, 0), \quad \mathbf{up} = (0, 1, 0).$$

The camera direction becomes

$$\mathbf{cameraDirection} = \frac{(0, 2, 3)}{\sqrt{2^2 + 3^2}} \approx (0, 0.5547, 0.8320).$$

The dot product is

$$\mathbf{cameraDirection} \cdot \mathbf{up} \approx 0.5547 \neq 0,$$

so the world up is not orthogonal to the camera direction.

If we directly set

$$\mathbf{cameraUp} = \mathbf{up},$$

then **cameraUp** is not perpendicular to **cameraDirection**. The resulting basis vectors (Right, Up, Direction) are no longer orthogonal, leading to a **skewed view matrix**. This skew manifests visually as stretching or tilting of the rendered scene.

5.6.6 LookAt Matrix

A powerful property of matrices is that, if you define a coordinate space using three perpendicular (or at least non-collinear) axes, you can create a matrix from these three axes together with a translation vector. Multiplying any vector by this matrix will transform it into the new coordinate space.

This is exactly what the **LookAt** matrix does. Now that we have three perpendicular axes and a position vector defining the camera space, we can construct our own **LookAt** matrix:

$$\text{LookAt} = \begin{bmatrix} R_x & U_x & D_x & 0 \\ R_y & U_y & D_y & 0 \\ R_z & U_z & D_z & 0 \\ -P_x & -P_y & -P_z & 1 \end{bmatrix}$$

where

$R = (R_x, R_y, R_z)$ is the right vector,

$U = (U_x, U_y, U_z)$ is the up vector,

$D = (D_x, D_y, D_z)$ is the direction (forward) vector,

$P = (P_x, P_y, P_z)$ is the camera's position vector.

By applying this **LookAt** matrix as the view matrix, all world coordinates are transformed into the view space defined by the camera. In other words, the **LookAt** matrix constructs a view that “looks at” a given target from a given position.

Luckily for us, GLM already does all this work for us. We only have to specify a camera position, a target position and a vector that represents the up vector in world space (the up vector we used for calculating the right vector). GLM then creates the **LookAt** matrix that we can use as our view matrix:

```
1 glm::mat4 view;  
2 view = glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f),  
3                   glm::vec3(0.0f, 0.0f, 0.0f),  
4                   glm::vec3(0.0f, 1.0f, 0.0f));
```

The `glm::LookAt` function requires a position, target and up vector respectively. Before delving into user input, let's get a little funky first by rotating the camera around our scene. We keep the target of the scene at (0,0,0). We use a little bit of trigonometry to create an x and z coordinate each frame that represents a point on a circle and we'll use these for our camera position. By re-calculating the x and y coordinate over time we're traversing all the points in a circle and thus the camera rotates around the scene. We enlarge this circle by a pre-defined radius and create a new view matrix each frame using GLFW's `glfwGetTime` function:

```
1 const float radius = 10.0f;  
2 float camX = sin(glfwGetTime()) * radius;  
3 float camZ = cos(glfwGetTime()) * radius;  
4 glm::mat4 view;  
5 view = glm::lookAt(glm::vec3(camX, 0.0, camZ), glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
```

5.6.7 Move Around

It will be much more fun if we can control the position and the view point of the camera using keyboard and mouse just like playing a video game!

```
1 glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);  
2 glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);  
3 glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
```

The **LookAt** function now becomes:

```
1 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
```

First we set the camera position to the previously defined cameraPos. The direction is the current position + the direction vector, this is because **we want the camera to always look in its front direction no matter where it moves**.

Then let's define a function to manage all GLFW's keyboard inputs:

```
1 void processInput(GLFWwindow *window)
2 {
3     if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
4         glfwSetWindowShouldClose(window, true);
5     const float cameraSpeed = 0.05f; // adjust accordingly
6     if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
7         cameraPos += cameraSpeed * cameraFront;
8     if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
9         cameraPos -= cameraSpeed * cameraFront;
10    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
11        cameraPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
12    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
13        cameraPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
14 }
```

Whenever we press one of the WASD keys, the camera's position is updated accordingly. If we want to move forward or backwards we add or subtract the direction vector from the position vector scaled by some speed value. If we want to move sideways we do a cross product to create a right vector and we move along the right vector accordingly.

Note that we normalize the resulting right vector. If we wouldn't normalize this vector, the resulting cross product may return differently sized vectors based on the cameraFront variable. If we would not normalize the vector we would move slow or fast based on the camera's orientation instead of at a consistent movement speed.

To apply the function processInput, we need to call it in our render loop:

```
1 while (!glfwWindowShouldClose(window)) {
2     processInput(window);
3     ...
4     glm::mat4 view;
5     view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
6     ...
7 }
```

Currently we used a constant value for movement speed when walking around. In theory this seems fine, but in practice people's machines have different processing powers and the result of that is that some people are able to render much more frames than others each second. Whenever a user renders more frames than another user he also calls processInput more often. The result is that some people move really fast and some really slow depending on their setup. When shipping your application you want to make sure it runs the same on all kinds of hardware.

Graphics applications and games usually keep track of a deltaTime variable that stores the time it took to render the last frame. We then multiply all velocities with this deltaTime value. The result is that when we have a large deltaTime in a frame, meaning that the last frame took longer than average, the velocity for that frame will also be a bit higher to balance it all out. When using this approach it does not matter if you have a very fast or slow pc, the velocity of the camera will be balanced out accordingly so each user will have the same experience.

To calculate the deltaTime value we keep track of 2 global variables:

```
1 float deltaTime = 0.0f; // Time between current frame and last frame
2 float lastFrame = 0.0f; // Time of last frame
```

Within each frame we then calculate the new deltaTime value for later use:

```

1 float currentFrame = glfwGetTime();
2 deltaTime = currentFrame - lastFrame;
3 lastFrame = currentFrame;

```

Now that we have deltaTime we can take it into account when calculating the velocities:

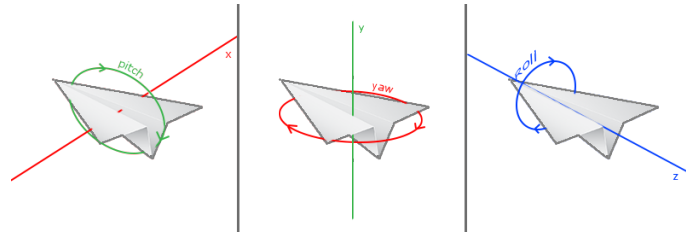
```

1 void processInput(GLFWwindow *window)
2 {
3     float cameraSpeed = 2.5f * deltaTime;
4     ...
5 }

```

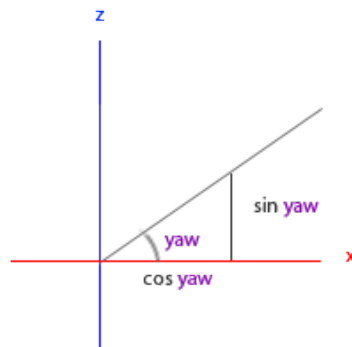
5.6.8 Look Around

To look around the scene we have to change the cameraFront vector based on the input of the mouse. However, changing the direction vector based on mouse rotations is a little complicated and requires some trigonometry. Euler angles are 3 values that can represent any rotation in 3D and there are 3 Euler angles: pitch, yaw and roll. The following image gives them a visual meaning:



The pitch is the angle that depicts how much we're looking up or down as seen in the first image. The second image shows the yaw value which represents the magnitude we're looking to the left or to the right. The roll represents how much we roll as mostly used in space-flight cameras. Each of the Euler angles are represented by a single value and with the combination of all 3 of them we can calculate any rotation vector in 3D.

For our camera system we only care about the yaw and pitch values so we won't discuss the roll value here. Given a pitch and a yaw value we can convert them into a 3D vector that represents a new direction vector. The process of converting yaw and pitch values to a direction vector requires a bit of trigonometry. Let's imagine this triangle, but now looking at it from a top perspective with the adjacent and opposite sides being parallel to the scene's x and z axis (as if looking down the y-axis).

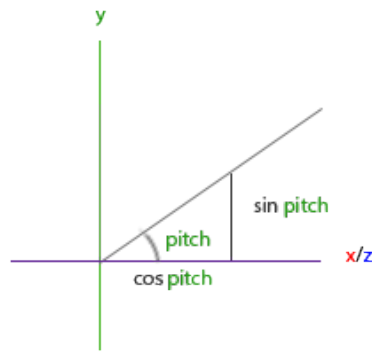


If we visualize the yaw angle to be the counter-clockwise angle starting from the x side we can see that the length of the x side relates to $\cos(\text{yaw})$. And similarly how the length of the z side relates to $\sin(\text{yaw})$. If we take this knowledge and a given yaw value we can use it to create a camera direction vector:

```

1 glm::vec3 direction;
2 direction.x = cos(glm::radians(yaw)); // Note that we convert the angle to radians first
3 direction.z = sin(glm::radians(yaw));

```



This solves how we can get a 3D direction vector from a yaw value, but pitch needs to be included as well. Let's now look at the y axis side as if we're sitting on the xz plane:

Similarly, from this triangle we can see that the direction's y component equals $\sin(\text{pitch})$ so let's fill that in:

```
1 direction.y = sin(glm::radians(pitch));
```

But now, the projection to x/z hypotenuse is no longer 1, instead, it is $\cos(\text{pitch})$ so we need to make sure this is also part of the direction vector. With this included we get the final direction vector as translated from yaw and pitch Euler angles:

```
1 direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
2 direction.y = sin(glm::radians(pitch));
3 direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
```

To make sure the camera points towards the negative z-axis by default we can give the yaw a default value of a 90 degree clockwise rotation. Positive degrees rotate counter-clockwise so we set the default yaw value to:

```
1 yaw = -90.0f;
```

The yaw and pitch values are obtained from mouse (or controller/joystick) movement where horizontal mouse-movement affects the yaw and vertical mouse-movement affects the pitch. The idea is to store the last frame's mouse positions and calculate in the current frame how much the mouse values changed. The higher the horizontal or vertical difference, the more we update the pitch or yaw value and thus the more the camera should move.

First we will tell GLFW that it should hide the cursor and capture it. Capturing a cursor means that, once the application has focus, the mouse cursor stays within the center of the window (unless the application loses focus or quits). We can do this with one simple configuration call:

```
1 glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

After this call, wherever we move the mouse it won't be visible and it should not leave the window. This is perfect for an FPS camera system. To calculate the pitch and yaw values we need to tell GLFW to listen to mouse-movement events. We do this by creating a callback function with the following prototype:

```
1 void mouse_callback(GLFWwindow* window, double xpos, double ypos);
```

Here `xpos` and `ypos` represent the current mouse positions. As soon as we register the callback function with GLFW each time the mouse moves, the `mouse_callback` function is called:

```
1 glfwSetCursorPosCallback(window, mouse_callback);
```

The first step is to calculate the offset of the mouse since last frame. We first have to store the last mouse positions in the application, which we initialize to be in the center of the screen (screen size is 800 by 600) initially:

```
1 float lastX = 400, lastY = 300;
```

Then in the mouse's callback function we calculate the offset movement between the last and current frame:

```
1 float xoffset = xpos - lastX;
2 float yoffset = lastY - ypos; // reversed since y-coordinates range from bottom to top
3 lastX = xpos;
4 lastY = ypos;
5
6 const float sensitivity = 0.1f;
7 xoffset *= sensitivity;
8 yoffset *= sensitivity;
```

Next we add the offset values to the globally declared pitch and yaw values:

```
1 yaw += xoffset;
2 pitch += yoffset;
```

In the third step we'd like to add some constraints to the camera so users won't be able to make weird camera movements (also causes a LookAt flip once direction vector is parallel to the world up direction). The pitch needs to be constrained in such a way that users won't be able to look higher than 89 degrees (at 90 degrees we get the LookAt flip) and also not below -89 degrees. This ensures the user will be able to look up to the sky or below to his feet but not further. The constraints work by replacing the Euler value with its constraint value whenever it breaches the constraint:

```
1 if(pitch > 89.0f)
2     pitch = 89.0f;
3 if(pitch < -89.0f)
4     pitch = -89.0f;
```

The fourth and last step is to calculate the actual direction vector using the formula from the previous section:

```
1 glm::vec3 direction;
2 direction.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
3 direction.y = sin(glm::radians(pitch));
4 direction.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
5 cameraFront = glm::normalize(direction);
```

If you'd now run the code you'll notice the camera makes a large sudden jump whenever the window first receives focus of your mouse cursor. The cause for this sudden jump is that as soon as your cursor enters the window the mouse callback function is called with an xpos and ypos position equal to the location your mouse entered the screen from. This is often a position that is significantly far away from the center of the screen, resulting in large offsets and thus a large movement jump. We can circumvent this issue by defining a global bool variable to check if this is the first time we receive mouse input. If it is the first time, we update the initial mouse positions to the new xpos and ypos values. The resulting mouse movements will then use the newly entered mouse's position coordinates to calculate the offsets:

```
1 if (firstMouse) // initially set to true
2 {
```

```

3     lastX = xpos;
4     lastY = ypos;
5     firstMouse = false;
6 }

```

5.6.9 Zoom

As a little extra to the camera system we'll also implement a zooming interface. In the previous chapter we said the Field of view or fov largely defines how much we can see of the scene. When the field of view becomes smaller, the scene's projected space gets smaller. This smaller space is projected over the same NDC, giving the illusion of zooming in. To zoom in, we're going to use the mouse's scroll wheel. Similar to mouse movement and keyboard input we have a callback function for mouse scrolling:

```

1 void scroll_callback(GLFWwindow* window, double xoffset, double yoffset){
2     fov -= (float)yoffset;
3     if (fov < 1.0f)
4         fov = 1.0f;
5     if (fov > 45.0f)
6         fov = 45.0f;
7 }

```

When scrolling, the yoffset value tells us the amount we scrolled vertically. When the `scroll_callback` function is called we change the content of the globally declared fov variable. Since 45.0 is the default fov value we want to constrain the zoom level between 1.0 and 45.0.

We now have to upload the perspective projection matrix to the GPU each frame, but this time with the fov variable as its field of view:

```

1 projection = glm::perspective(glm::radians(fov), 800.0f / 600.0f, 0.1f, 100.0f);

```

And lastly don't forget to register the scroll callback function:

```

1 glfwSetScrollCallback(window, scroll_callback);

```

5.6.10 Camera Class

For the header file, we need to define all its attributes and functions:

```

1 #ifndef CAMERA_H
2 #define CAMERA_H
3
4 #include <glm.hpp>
5 #include <gtc/matrix_transform.hpp>
6 #include <algorithm>
7 #include <GLFW/glfw3.h>
8
9 class Camera {
10 public:
11     // --- Public state ---
12     glm::vec3 Position{0.0f, 0.0f, 3.0f};
13     glm::vec3 Front{0.0f, 0.0f, -1.0f};
14     glm::vec3 Up{0.0f, 1.0f, 0.0f};
15     glm::vec3 Right{1.0f, 0.0f, 0.0f};
16     glm::vec3 WorldUp{0.0f, 1.0f, 0.0f};
17
18     // Euler angles (degrees)
19     float Yaw = -90.0f; // facing -Z by default
20     float Pitch = 0.0f;

```

```

21     float lastX;
22     float lastY;
23     bool firstMouse = true;
24
25     // Tunables
26     float MovementSpeed = 3.0f; // units / second
27     float MouseSensitivity = 0.1f; // degrees / pixel
28     float Zoom = 45.0f; // FOV (deg)
29
30     // --- Ctors ---
31     Camera();
32     Camera(glm::vec3 position, glm::vec3 worldUp, float yawDeg = -90.0f, float pitchDeg = 0.0f);
33
34     // --- Matrices ---
35     glm::mat4 GetViewMatrix() const;
36     glm::mat4 GetProjection(float aspect, float nearPlane = 0.1f, float farPlane = 100.0f) const;
37
38     // key* are true if pressed; deltaTime in seconds.
39     void ProcessKeyboard(GLFWwindow *window, float deltaTime);
40     // Mouse deltas in pixels since last frame; positive x = right, positive y = up.
41     void ProcessMouseMovement(float xoffset, float yoffset, bool constrainPitch = true);
42     // Mouse wheel delta (typical GLFW sign: positive = scroll up = zoom in).
43     void ProcessMouseScroll(float yoffset);
44
45 private:
46     void updateVectors(); // recompute Front/Right/Up from Yaw/Pitch + WorldUp
47 };
48
49 #endif

```

In the source file, we just need to implement all functions:

```

1  #include "camera.h"
2  #include <cmath>
3
4  Camera::Camera() {
5      updateVectors();
6  }
7
8
9  Camera::Camera(glm::vec3 position, glm::vec3 worldUp, float yawDeg, float pitchDeg)
10      : Position(position), WorldUp(worldUp), Yaw(yawDeg), Pitch(pitchDeg) {
11      updateVectors();
12  }
13
14  glm::mat4 Camera::GetViewMatrix() const {
15      return glm::lookAt(Position, Position + Front, Up);
16  }
17
18  glm::mat4 Camera::GetProjection(float aspect, float nearPlane, float farPlane) const {
19      return glm::perspective(glm::radians(Zoom), aspect, nearPlane, farPlane);
20  }
21
22  void Camera::ProcessKeyboard(GLFWwindow *window, float deltaTime) {
23      if(glwfGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
24          glfwSetWindowShouldClose(window, true);
25      const float velocity = MovementSpeed * deltaTime;
26      if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
27          Position += velocity * Front;
28      if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
29          Position -= velocity * Front;
30      if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
31          Position -= glm::normalize(glm::cross(Front, Up)) * velocity;
32      if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)

```

```

33     Position += glm::normalize(glm::cross(Front, Up)) * velocity;
34 }
35
36 void Camera::ProcessMouseMove(float xoffset, float yoffset, bool constrainPitch) {
37     xoffset *= MouseSensitivity;
38     yoffset *= MouseSensitivity;
39
40     Yaw    += xoffset;
41     Pitch += yoffset;
42
43     if (constrainPitch) {
44         Pitch = std::clamp(Pitch, -89.9f, 89.9f);
45     }
46
47     updateVectors();
48 }
49
50 void Camera::ProcessMouseScroll(float yoffset) {
51     Zoom -= yoffset;
52     Zoom = std::clamp(Zoom, 1.0f, 90.0f);
53 }
54
55 void Camera::updateVectors() {
56     // Convert yaw/pitch to a forward vector (right-handed, OpenGL-style)
57     const float yawRad    = glm::radians(Yaw);
58     const float pitchRad = glm::radians(Pitch);
59
60     glm::vec3 f;
61     f.x = std::cos(yawRad) * std::cos(pitchRad);
62     f.y = std::sin(pitchRad);
63     f.z = std::sin(yawRad) * std::cos(pitchRad);
64     Front = glm::normalize(f);
65
66     // Derive Right and Up to form an orthonormal basis (Gram{Schmidt)
67     Right = glm::normalize(glm::cross(Front, WorldUp));
68     // If Front is nearly parallel to WorldUp, Right can become tiny; clamping Pitch above avoids this.
69     Up    = glm::normalize(glm::cross(Right, Front));
70 }

```

We need to further modify our main function to adopt our Camera class:

First, we need to declare a global Camera object:

```

1 float deltaTime = 0.0f;           // Time between current frame and last frame
2 float lastFrame = 0.0f; // Time of last frame
3
4 int SCR_WIDTH = 800;
5 int SCR_LENGTH = 600;
6 Camera camera;

```

Also remember to update your CMakeLists.txt file:

```

1 add_executable(demo main.cpp
2                 src/shaderprogram.cpp
3                 src/stb_image.cpp
4                 src/mesh.cpp
5                 src/tiny_obj_loader.cpp
6                 src/camera.cpp)

```

Then in the render loop:

```

1 while (!glfwWindowShouldClose(window)) {
2     float currentTime = glfwGetTime();
3     deltaTime = currentTime - lastFrame;
4     lastFrame = currentTime;
5     camera.ProcessKeyboard(window, deltaTime);
6
7     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
8     glEnable(GL_DEPTH_TEST);
9     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
10
11     for(unsigned int i = 1; i < 11; i++) {
12         glm::mat4 model = glm::mat4(1.0f);
13         model = glm::translate(model, cubePositions[i-1]);
14         model = glm::rotate(model, (float) glfwGetTime() * glm::radians(20.0f * i), glm::vec3(1.0f, 0.3f, 0.5f));
15
16         glm::mat4 view;
17         view = camera.GetViewMatrix();
18
19         glm::mat4 projection;
20         projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
21         quad.draw(model, view, projection);
22     }
23     glfwSwapBuffers(window);
24     glfwPollEvents();
25 }

```

Also, because we calculate Yaw and Pitch in the class, we need to modify `mouse_callback`:

```

1 void mouse_callback(GLFWwindow* window, double xposIn, double yposIn)
2 {
3     float xpos = static_cast<float>(xposIn);
4     float ypos = static_cast<float>(yposIn);
5
6     if (camera.firstMouse)
7     {
8         camera.lastX = xpos;
9         camera.lastY = ypos;
10        camera.firstMouse = false;
11    }
12
13    float xoffset = xpos - camera.lastX;
14    float yoffset = camera.lastY - ypos; // reversed since y-coordinates go from bottom to top
15
16    camera.lastX = xpos;
17    camera.lastY = ypos;
18
19    camera.ProcessMouseMovement(xoffset, yoffset);
20 }

```

Similarly, we modify our `scroll_callback` function:

```

1 void scroll_callback(GLFWwindow* window, double xoffset, double yoffset){
2     camera.ProcessMouseScroll(static_cast<float>(yoffset));
3 }

```

6 Basic Lighting

First, we need an object for the light to shine on, so we'll use the familiar container cube from earlier chapters. We'll also add a light object to mark the position of the light source in the 3D scene, and for simplicity, we'll represent it with another cube since we already have it. Finally, we'll create a vertex shader to draw the container; the cube's vertex positions stay the same, though we won't use texture coordinates this time, and we'll use a simplified version of the vertex shader from before.

6.1 A lighting Scene

Let's first create two cubes, one is for the container object, and the other one is the light source cube. Since they are different objects used for different purposes, we need to write shader programs for both items separately and create shader program objects separately. For the vertex shader, since we will use the same cube, thus it remains same (cube_vertex.vert):

```
1  #version 410 core
2  layout (location = 0) in vec3 aPos;
3
4  uniform mat4 model;
5  uniform mat4 view;
6  uniform mat4 projection;
7
8  void main(){
9      gl_Position = projection * view * model * vec4(aPos, 1.0);
10 }
```

for the fragment shaders, we need two fragment shader because lighting computation for the lighting source and the object in the scene are different:

For the container object in the scene (cube_fragment.frag):

```
1  #version 410 core
2  out vec4 FragColor;
3
4  void main(){
5      vec3 objectColor = vec3(1.0f, 0.5f, 0.31f);
6      vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);
7      FragColor = vec4(lightColor * objectColor, 1.0);
8  }
```

for the lighting source cube (lighting_cube_fragment.frag):

```
1  #version 410 core
2  out vec4 FragColor;
3
4  void main(){
5      FragColor = vec4(1.0);
6  }
```

When rendering the scene, we use the lighting shader to draw the container object (and any others) and switch to the light source's shader to render the light itself. The light source cube mainly indicates the light's position in the scene—since a light's position is normally invisible, we draw a small white cube at that location using its own shader to visually show where the light comes from. Throughout the lighting chapters, we'll gradually refine these shaders to create more realistic lighting effects:

```
1  //container cube
2  std::string cube_vertPath = std::string(SHADER_DIR) + "cube_vertex.vert";
3  std::string cube_fragPath = std::string(SHADER_DIR) + "cube_fragment.frag";
4  ShaderProgram cubeShader(cube_vertPath, cube_fragPath);
5
```

```

6 Mesh cube(std::string(ASSET_DIR) + "box.obj", cubeShader.getID());
7
8 //lighting source cube
9 std::string lighting_cube_vertPath = std::string(SHADER_DIR) + "cube_vertex.vert";
10 std::string lighting_cube_fragPath = std::string(SHADER_DIR) + "lighting_cube_fragment.frag";
11 ShaderProgram lightingCubeShader(lighting_cube_vertPath, lighting_cube_fragPath);
12
13
14 Mesh lightingCube(std::string(ASSET_DIR) + "box.obj", lightingCubeShader.getID());

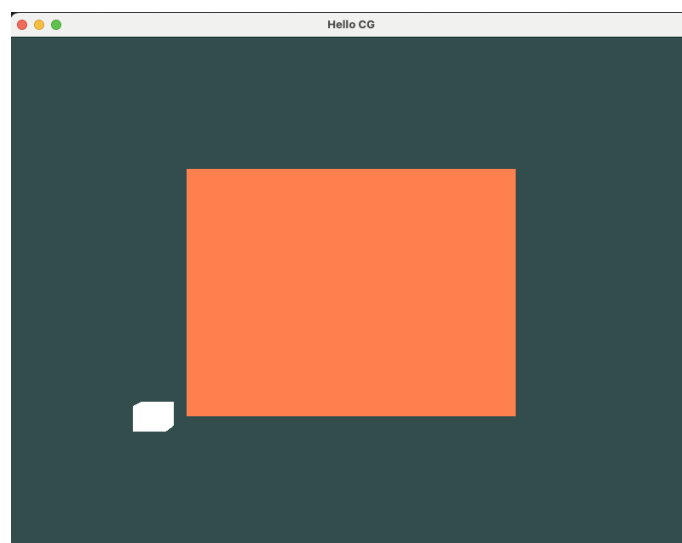
```

In the main loop:

```

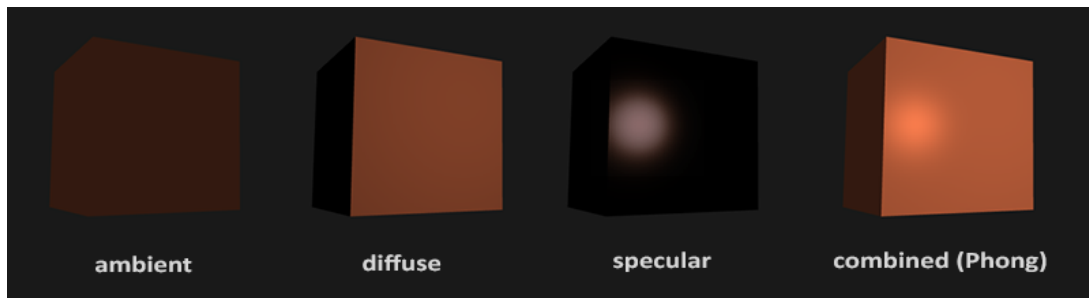
1 while (!glfwWindowShouldClose(window)) {
2     float currentFrame = glfwGetTime();
3     deltaTime = currentFrame - lastFrame;
4     lastFrame = currentFrame;
5     camera.ProcessKeyboard(window, deltaTime);
6
7     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
8     glEnable(GL_DEPTH_TEST);
9     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
10
11     cubeShader.use();
12     glm::mat4 model = glm::mat4(1.0f);
13     glm::mat4 view;
14     view = camera.GetViewMatrix();
15     glm::mat4 projection;
16     projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
17     cube.draw(model, view, projection);
18
19     lightingCubeShader.use();
20     glm::vec3 lightPos(-1.2f, -1.0f, -2.0f);
21     model = glm::mat4(1.0f);
22     model = glm::translate(model, lightPos);
23     model = glm::scale(model, glm::vec3(0.2f));
24     lightingCube.draw(model, view, projection);
25
26     glfwSwapBuffers(window);
27     glfwPollEvents();
28 }

```



6.2 Basic Lighting Concepts

Real-world lighting is incredibly complex and influenced by many factors, far beyond what our limited processing power can handle. In OpenGL, lighting is simplified using models that approximate real physics but are much easier to compute while still looking realistic. One common model is the Phong lighting model, which includes three key components: ambient, diffuse, and specular lighting. The image below shows how each component looks individually and when combined.



- **Ambient lighting:** Even in darkness, there's usually some light in the environment (like moonlight or distant sources), so objects are rarely completely black. To mimic this, we add a constant ambient term that ensures the object always retains a bit of color.
- **Diffuse lighting:** This simulates how light directionally affects an object's surface. It's the most noticeable part of the lighting model—areas facing the light appear brighter, while those angled away look dimmer.
- **Specular lighting:** This represents the shiny highlight seen on reflective surfaces. These bright spots depend more on the color of the light source than the object itself, creating the illusion of gloss or polish.

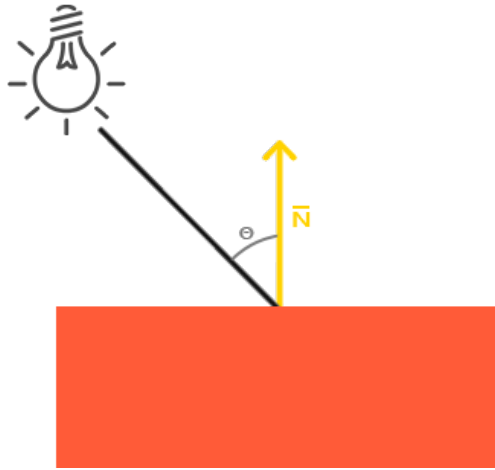
6.2.1 Ambient Lighting

In reality, light comes from many sources and often bounces off surfaces, indirectly illuminating areas not directly hit by light. Simulating this behavior accurately requires complex and costly global illumination algorithms. To keep things simple, we approximate this effect using ambient lighting, which adds a small constant light color to simulate scattered light. Implementing ambient lighting is straightforward: we multiply the light's color by a small ambient factor, then by the object's color, and use the result as the fragment color in the cube's shader.

```
1  #version 410 core
2  out vec4 FragColor;
3
4  void main()
5  {
6      vec3 objectColor = vec3(1.0f, 0.5f, 0.31f);
7      vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);
8
9      float ambientStrength = 0.1;
10     vec3 ambient = ambientStrength * lightColor;
11
12     vec3 result = ambient * objectColor;
13     FragColor = vec4(result, 1.0);
14 }
```

6.2.2 Diffuse Lighting

Diffuse lighting gives the object more brightness the closer its fragments are aligned to the light rays from a light source. To give you a better understanding of diffuse lighting take a look at the following image:



When a light ray hits a fragment of an object, the brightness depends on the angle between the light's direction and the fragment's surface. We use a normal vector—a perpendicular vector to the surface—to measure this angle. The dot product of the light direction and the normal vector gives the cosine of that angle, showing how much light affects the surface: a perpendicular hit gives maximum brightness (*dotproduct* ≈ 1), while a 90-degree angle gives no light (dot product = 0). To ensure accurate results, **both vectors must be normalized** (length 1). This value determines how strongly each fragment is lit, creating realistic shading based on its orientation to the light.

To calculate diffuse lighting, we need two main components:

- **Normal vector** – a vector perpendicular to the surface at a vertex.
- **Light direction vector** – obtained by subtracting the fragment's position from the light's position, giving the direction of the incoming light toward the fragment.

A normal vector is a (unit) vector that is perpendicular to the surface of a vertex. Since a vertex by itself has no surface (it's just a single point in space) we retrieve a normal vector by using its surrounding vertices to figure out the surface of the vertex. Now, we update our `loadOBJ_` function to compute per-vertex normal:

```

1  bool Mesh::loadOBJ_(const std::string& path)
2  {
3      vertices.clear();
4      indices.clear();
5
6      tinyobj::ObjReaderConfig config;
7      config.triangulate = false; // we'll triangulate ourselves (fan) to match your logic
8      config.vertex_color = false;
9
10     tinyobj::ObjReader reader;
11     if (!reader.ParseFromFile(path, config)) {
12         if (!reader.Error().empty())
13             std::cerr << "tinyobj error: " << reader.Error() << "\n";
14         return false;
15     }
16     if (!reader.Warning().empty()) {
17         std::cerr << "tinyobj warning: " << reader.Warning() << "\n";
18     }
19
20     const tinyobj::attrib_t& attrib = reader.GetAttrib();
21     const std::vector<tinyobj::shape_t>& shapes = reader.GetShapes();
22
23     auto getPos = [&](int vi) -> glm::vec3 {
24         const float* v = attrib.vertices.data() + 3 * vi;
25         return glm::vec3(v[0], v[1], v[2]);
26     };
27
28     auto getUV = [&](int ti) -> glm::vec2 {
29         const float* t = attrib.texcoords.data() + 2 * ti;
30         return glm::vec2(t[0], t[1]);
31     };

```

```

32
33     uint32_t baseIndex = 0;
34
35     for (const auto& shape : shapes) {
36         // --- 1) Shape centroid over unique referenced vertices
37         glm::dvec3 centroid(0.0, 0.0, 0.0);
38         size_t count = 0;
39         std::unordered_set<int> used;
40         used.reserve(shape.mesh.indices.size());
41         for (const auto& idx : shape.mesh.indices) {
42             if (idx.vertex_index < 0) continue;
43             if (used.insert(idx.vertex_index).second) {
44                 glm::vec3 p = getPos(idx.vertex_index);
45                 centroid += glm::dvec3(p);
46                 ++count;
47             }
48         }
49         if (count > 0) centroid /= static_cast<double>(count);
50
51         // --- 2) Iterate faces and triangulate (fan) if needed
52         size_t index_offset = 0;
53         for (size_t f = 0; f < shape.mesh.num_face_vertices.size(); ++f) {
54             int fv = shape.mesh.num_face_vertices[f];
55             if (fv < 3) { index_offset += fv; continue; }
56
57             // Collect the whole polygon's positions (for normal + fan)
58             std::vector<glm::vec3> facePos(fv);
59             for (int v = 0; v < fv; ++v) {
60                 const auto& idx = shape.mesh.indices[index_offset + v];
61                 if (idx.vertex_index < 0) { /* malformed face */ continue; }
62                 facePos[v] = getPos(idx.vertex_index);
63             }
64
65             // Fan triangulation: (0, k, k+1)
66             for (int k = 1; k < fv - 1; ++k) {
67                 const glm::vec3& p0 = facePos[0];
68                 const glm::vec3& p1 = facePos[k];
69                 const glm::vec3& p2 = facePos[k + 1];
70
71                 // --- 3) Flat normal
72                 glm::vec3 e1 = p1 - p0;
73                 glm::vec3 e2 = p2 - p0;
74                 glm::vec3 n = glm::cross(e1, e2);
75                 float len = glm::length(n);
76                 if (len > 0.0f) n /= len; else n = glm::vec3(0, 0, 1);
77
78                 // --- 4) Flip if pointing inward (toward centroid),
79                 //         but handle the "centroid in the plane" case
80                 glm::vec3 faceCenter = (p0 + p1 + p2) / 3.0f;
81                 glm::vec3 outDir = faceCenter - glm::vec3(centroid);
82                 float outLen2 = glm::dot(outDir, outDir);
83                 float d = glm::dot(n, outDir);
84
85                 if (outLen2 < 1e-5f) {
86                     // Degenerate: centroid basically lies in the same plane.
87                     // For big flat things (like your floor), force Y-up.
88                     if (n.y < 0.0f)
89                         n = -n;
90                 } else if (d < 0.0f) {
91                     n = -n;
92                 }
93
94                 // Helper to emit a single vertex (pos, n, uv) + push index
95                 auto emit = [&](int localFaceVertex) {
96                     const auto& idx = shape.mesh.indices[index_offset + localFaceVertex];
97                     const glm::vec3 p = getPos(idx.vertex_index);

```

```

98         vertices.push_back(p.x);
99         vertices.push_back(p.y);
100        vertices.push_back(p.z);
101
102
103        vertices.push_back(n.x);
104        vertices.push_back(n.y);
105        vertices.push_back(n.z);
106
107        float u = 0.0f, v = 0.0f;
108        if (idx.texcoord_index >= 0 &&
109            static_cast<size_t>(2 * idx.texcoord_index + 1) < attrib.texcoords.size())
110        {
111            glm::vec2 uv = getUV(idx.texcoord_index);
112            u = uv.x; v = uv.y;
113        }
114        vertices.push_back(u);
115        vertices.push_back(v);
116
117        indices.push_back(baseIndex++);
118    };
119
120    // Triangle: (0, k, k+1)
121    emit(0);
122    emit(k);
123    emit(k + 1);
124 }
125
126 index_offset += fv;
127 }
128 }
129
130 return !vertices.empty() && !indices.empty();
131 }
132

```

Flip normals outward: The first face at $z = -0.5$ will compute $n = (0,0,\pm 1)$. Since its center is “more negative z ” than the centroid $(0,0,0)$, outDir has negative z ; if n was $(0,0,1)$ the dot-product is negative $\rightarrow$ we flip to $(0,0,-1)$ to match your reference normals.

Now, there are 3 per-vertex attributes, position, normal, and UV. thus, we need to update our shader program by introducing a new input variable and we need to forward the normal vectors from the vertex shader to the fragment shader:

```

1  #version 410 core
2  in vec3 aPos;
3  in vec3 aNor;
4  in vec2 aTexCoord;
5
6  uniform mat4 model;
7  uniform mat4 view;
8  uniform mat4 projection;
9
10 out vec3 Normal;
11 out vec3 FragPos;
12 void main(){
13     gl_Position = projection * view * model * vec4(aPos, 1.0);
14     FragPos = vec3(model * vec4(aPos, 1.0));
15     Normal = aNor;
16 }

```

Now that we added a normal vector to each of the vertices and updated the vertex shader we should update the vertex attribute pointers as well.

```

1 void Mesh::createBuffers_()
2 {
3
4     const GLint posLoc = glGetAttribLocation(shaderProgramID, "aPos");
5     const GLint norLoc = glGetAttribLocation(shaderProgramID, "aNor");
6     const GLint uvLoc  = glGetAttribLocation(shaderProgramID, "aTexCoord");
7
8     GLsizei stride = 8 * sizeof(float);
9     std::vector<VertexAttribute> attributes = {
10         VertexAttribute{ static_cast<GLuint>(posLoc), 3, GL_FLOAT, GL_FALSE, stride, 0 },
11         VertexAttribute{ static_cast<GLuint>(norLoc), 3, GL_FLOAT, GL_FALSE, stride, 3 * sizeof(float)},
12         VertexAttribute{ static_cast<GLuint>(uvLoc), 2, GL_FLOAT, GL_FALSE, stride, 6 * sizeof(float) } };
13     ... ..
14
15 }

```

In the fragment shader program, we need to accept normal from vertex shader program and use it to compute the diffuse lighting:

```

1 #version 410 core
2 out vec4 FragColor;
3 in vec3 Normal;
4 in vec3 FragPos;
5
6
7 uniform vec3 objectColor;
8 uniform vec3 lightColor;
9 uniform vec3 lightPos;
10
11 void main()
12 {
13
14     float ambientStrength = 0.1;
15     vec3 ambient = ambientStrength * lightColor;
16
17     vec3 norm = normalize(Normal);
18     vec3 lightDir = normalize(lightPos - FragPos);
19
20     float diff = max(dot(norm, lightDir), 0.0);
21     vec3 diffuse = diff * lightColor;
22
23     vec3 result = (ambient + diffuse) * objectColor;
24     FragColor = vec4(result, 1.0);
25 }

```

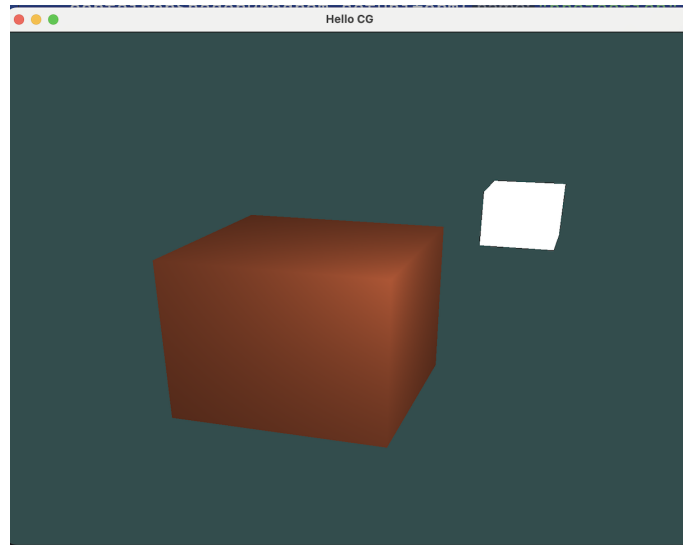
In the main function,

```

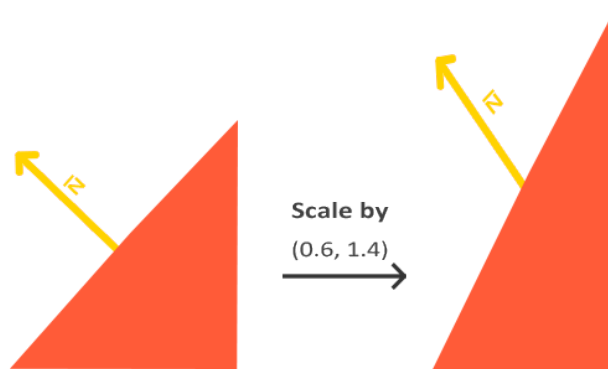
1 containerShaderProgram.use();
2 glm::vec3 lightPos(1.0f, 1.0f, 1.0f);
3 glm::vec3 objectColor(1.0f, 0.5f, 0.31f);
4 glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
5
6 glm::mat4 model = glm::mat4(1.0f);
7 glm::mat4 view = camera.GetViewMatrix();
8 glm::mat4 projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
9
10 containerShaderProgram.setUniform("lightPos", lightPos);
11 containerShaderProgram.setUniform("objectColor", objectColor);
12 containerShaderProgram.setUniform("lightColor", lightColor);
13
14 containerShaderProgram.setUniform("model", model);
15 containerShaderProgram.setUniform("view", view);

```

```
16 containerShaderProgram.setUniform("projection", projection);
```



In the previous section, we passed the normal vector directly from the vertex shader to the fragment shader, but since lighting calculations occur in world space, the normals also need to be transformed to world coordinates. However, this isn't as simple as multiplying by the model matrix because normals are direction vectors, not positions, and should not be affected by translation. Therefore, we use only the upper-left 3×3 portion of the model matrix to transform them. Additionally, when a model matrix includes non-uniform scaling, it can distort normals so they no longer remain perpendicular to the surface, which must be corrected using an appropriate transformation matrix.

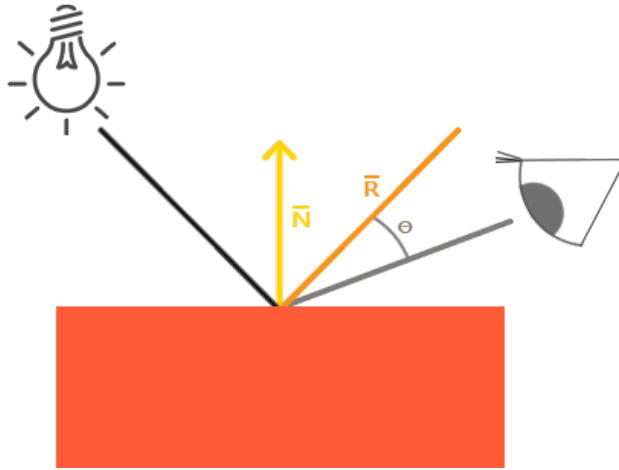


When we apply a non-uniform scale, the normal vectors are no longer perpendicular to the surface, which causes lighting distortions. To correct this, we use a special matrix called the normal matrix, designed to fix the scaling effect on normals. The normal matrix is defined as the transpose of the inverse of the upper-left 3×3 part of the model matrix. In practice, we can compute it directly in the vertex shader using the built-in `inverse()` and `transpose()` functions, after casting the model matrix to a 3×3 form so it properly multiplies with the normal vectors.

```
1 Normal = mat3(transpose(inverse(model))) * aNor;
```

6.2.3 Specular Lighting

Like diffuse lighting, specular lighting depends on the light's direction and the surface normal, but it also considers the view direction—the angle from which the viewer looks at the surface. It represents the reflective property of materials: if we imagine the surface as a mirror, the brightest specular spot appears where the reflected light aligns with the viewer's eye.



To calculate specular lighting, we first find a reflection vector by reflecting the light direction around the surface normal, then measure its angle with the view direction—the smaller the angle, the stronger the specular highlight. The view vector is obtained from the viewer's (camera's) world position and the fragment's position, and the specular intensity is multiplied by the light color and added to the ambient and diffuse components. While lighting can be calculated in either world or view space, world space is often easier to understand conceptually. In view space, the viewer's position is always at (0,0,0), but you must transform all relevant vectors and normals by the view matrix. To pass the viewer's position for world-space lighting, simply add a uniform for the camera position in the fragment shader.

```
1 uniform vec3 viewPos;
```

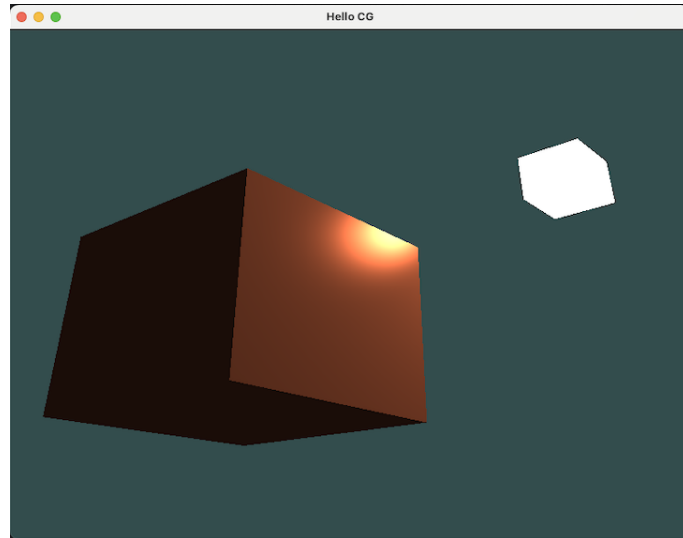
In the rendering loop, set up its value:

```
1 cubeShader.setUniform("viewPos", camera.Position);
```

inside our cube fragment shader, we need to do several steps:

- define a specular intensity value
- calculate the view direction vector and the corresponding reflect vector along the normal axis
- calculate the specular component
- add it to the ambient and diffuse components and multiply the combined result with the object's color

```
1 ... ..
2
3 float specularStrength = 2.0;
4 vec3 viewDir = normalize(viewPos - FragPos);
5 vec3 reflectDir = reflect(-lightDir, norm);
6 float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
7 vec3 specular = specularStrength * spec * lightColor;
8
9 vec3 result = (ambient + diffuse + specular) * objectColor;
10 FragColor = vec4(result, 1.0);
11 ... ..
```



6.3 Materials

In the real world, different materials respond to light in unique ways—steel shines more than clay, and wood reflects light differently than metal. Some surfaces reflect light sharply, creating small bright spots, while others scatter it broadly, producing softer highlights. To recreate these effects in OpenGL, we assign material properties to each surface. Unlike the previous chapter, where we used only object and light colors with ambient and specular intensity, we now define separate colors for the ambient, diffuse, and specular components, along with a shininess value, giving us full control over how each material interacts with light.

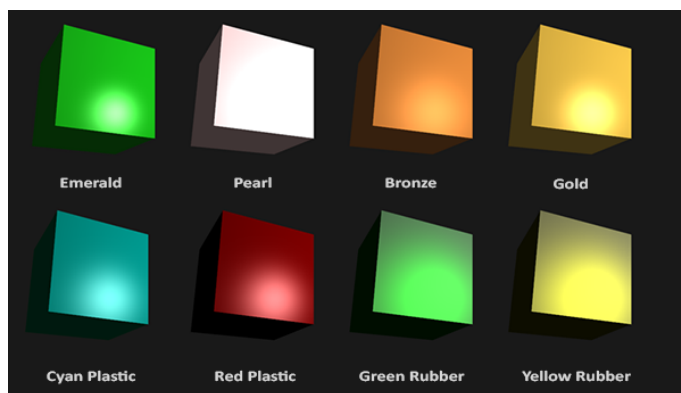
```
1 struct Material {  
2     vec3 ambient;  
3     vec3 diffuse;  
4     float shininess;  
5 };  
6  
7 uniform Material material;
```

In the fragment shader, we define a struct to organize the material properties of a surface instead of using separate uniform values. The struct includes color vectors for the Phong lighting components: ambient (the surface's color under ambient light), diffuse (the color under direct light), and specular (the color of the highlight). It also includes a shininess value that controls how concentrated the highlight appears. Together, these four properties allow us to simulate various real-world materials:

Name	Ambient			Diffuse			Specular			Shininess
	R	G	B	R	G	B	R	G	B	
emerald	0.0215	0.1745	0.0215	0.07568	0.61424	0.07568	0.633	0.727811	0.633	0.6
jade	0.135	0.2225	0.1575	0.54	0.89	0.63	0.316228	0.316228	0.316228	0.1
obsidian	0.05375	0.05	0.06625	0.18275	0.17	0.22525	0.332741	0.328634	0.346435	0.3
pearl	0.25	0.20725	0.20725	1.0	0.829	0.829	0.296648	0.296648	0.296648	0.088
ruby	0.1745	0.01175	0.01175	0.61424	0.04136	0.04136	0.727811	0.626959	0.626959	0.6
turquoise	0.1	0.18725	0.1745	0.396	0.74151	0.69102	0.297254	0.30829	0.306678	0.1
brass	0.329412	0.223529	0.027451	0.780392	0.568627	0.113725	0.992157	0.941176	0.807843	0.217949
bronze	0.2125	0.1275	0.054	0.714	0.4284	0.18144	0.393548	0.271906	0.166721	0.2
chrome	0.25	0.25	0.25	0.4	0.4	0.4	0.774597	0.774597	0.774597	0.6
copper	0.19125	0.0735	0.0225	0.7038	0.27048	0.0828	0.256777	0.137622	0.086014	0.1
gold	0.24725	0.1995	0.0745	0.75164	0.60648	0.22648	0.628281	0.555802	0.366065	0.4
silver	0.19225	0.19225	0.19225	0.50754	0.50754	0.50754	0.508273	0.508273	0.508273	0.4
black plastic	0.0	0.0	0.0	0.01	0.01	0.01	0.5	0.5	0.5	0.25
cyan plastic	0.0	0.1	0.06	0.0	0.509804	0.509804	0.501961	0.501961	0.501961	0.25
green plastic	0.0	0.0	0.0	0.1	0.35	0.1	0.45	0.55	0.45	0.25
red plastic	0.0	0.0	0.0	0.5	0.0	0.0	0.7	0.6	0.6	0.25
white plastic	0.0	0.0	0.0	0.55	0.55	0.55	0.7	0.7	0.7	0.25
yellow plastic	0.0	0.0	0.0	0.5	0.5	0.0	0.6	0.6	0.5	0.25
black rubber	0.02	0.02	0.02	0.01	0.01	0.01	0.4	0.4	0.4	0.0781
cyan rubber	0.0	0.05	0.05	0.4	0.5	0.5	0.04	0.7	0.7	0.0781
green rubber	0.0	0.05	0.0	0.4	0.5	0.4	0.04	0.7	0.04	0.0781
red rubber	0.05	0.0	0.0	0.5	0.4	0.4	0.7	0.04	0.04	0.0781
white rubber	0.05	0.05	0.05	0.5	0.5	0.5	0.7	0.7	0.7	0.0781
yellow rubber	0.05	0.05	0.0	0.5	0.5	0.4	0.7	0.7	0.04	0.0781

Table 1: Material properties for Phong lighting model.

As you can see, by correctly specifying the material properties of a surface it seems to change the perception we have of the object.



Next we want to change the lighting calculations to comply with the new material properties. Since all the material variables are stored in a struct we can access them from the material uniform:

```

1 void main(){
2     // ambient
3     vec3 ambient = lightColor * material.ambient;
4
5     // diffuse
6     vec3 norm = normalize(Normal);
7     vec3 lightDir = normalize(lightPos - FragPos);
8     float diff = max(dot(norm, lightDir), 0.0);
9     vec3 diffuse = lightColor * (diff * material.diffuse);
10
11    // specular
12    vec3 viewDir = normalize(viewPos - FragPos);
13    vec3 reflectDir = reflect(-lightDir, norm);
14    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
15    vec3 specular = lightColor * (spec * material.specular);

```

```

16
17     vec3 result = ambient + diffuse + specular;
18     FragColor = vec4(result, 1.0);
19 }

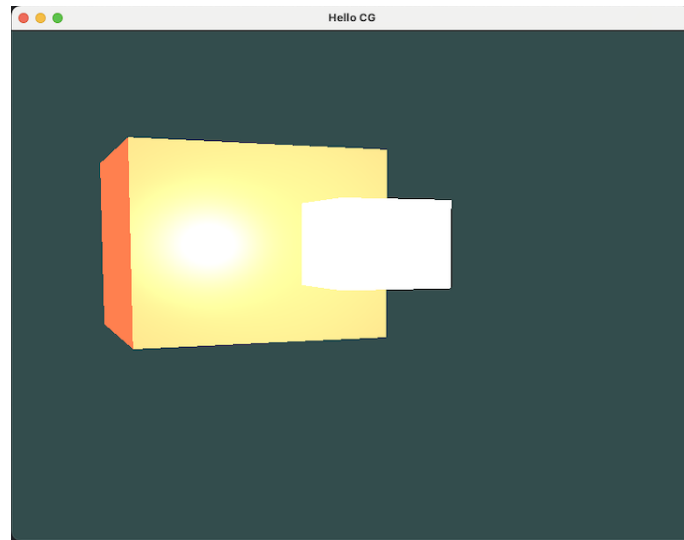
```

As you can see we now access all of the material struct's properties wherever we need them and this time calculate the resulting output color with the help of the material's colors. Each of the object's material attributes are multiplied with their respective lighting components.

```

1  while (!glfwWindowShouldClose(window)) {
2      float currentFrame = glfwGetTime();
3      deltaTime = currentFrame - lastFrame;
4      lastFrame = currentFrame;
5      camera.ProcessKeyboard(window, deltaTime);
6
7      glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
8      glEnable(GL_DEPTH_TEST);
9      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
10
11
12     cubeShader.use();
13     glm::vec3 lightPos(1.5f, 0.0f, 0.0f);
14
15     glm::mat4 model = glm::mat4(1.0f);
16     glm::mat4 view;
17     view = camera.GetViewMatrix();
18     glm::mat4 projection;
19     projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
20     cubeShader.setUniform("model", model);
21     cubeShader.setUniform("view", view);
22     cubeShader.setUniform("projection", projection);
23
24     // cubeShader.setUniform("objectColor", glm::vec3(1.0f, 0.5f, 0.31f));
25     cubeShader.setUniform("lightColor", glm::vec3(1.0f, 1.0f, 1.0f));
26     cubeShader.setUniform("lightPos", lightPos);
27     cubeShader.setUniform("viewPos", camera.Position);
28
29     cubeShader.setUniform("material.ambient", glm::vec3(1.0f, 0.5f, 0.31f));
30     cubeShader.setUniform("material.diffuse", glm::vec3(1.0f, 0.5f, 0.31f));
31     cubeShader.setUniform("material.specular", glm::vec3(0.5f, 0.5f, 0.5f));
32     cubeShader.setUniform("material.shininess", 32.0f);
33     cube.draw();
34
35     lightingCubeShader.use();
36
37     model = glm::mat4(1.0f);
38     model = glm::translate(model, lightPos);
39     model = glm::scale(model, glm::vec3(0.3f));
40
41     lightingCubeShader.setUniform("model", model);
42     lightingCubeShader.setUniform("view", view);
43     lightingCubeShader.setUniform("projection", projection);
44
45
46
47     lightingCube.draw();
48
49     glfwSwapBuffers(window);
50     glfwPollEvents();
51 }

```



The object is way too bright. The reason for the object being too bright is that the ambient, diffuse and specular colors are reflected with full force from any light source. Light sources also have different intensities for their ambient, diffuse and specular components respectively. We'll want to create something similar to the material struct for the light properties:

```
1 struct Light {
2     vec3 position;
3
4     vec3 ambient;
5     vec3 diffuse;
6     vec3 specular;
7 };
8
9 uniform Light light;
```

Also we need to modify the main function to adopt Light struct:

```
1 void main(){
2     // ambient
3     vec3 ambient = light.ambient * material.ambient;
4
5     // diffuse
6     vec3 norm = normalize(Normal);
7     vec3 lightDir = normalize(light.position - FragPos);
8     float diff = max(dot(norm, lightDir), 0.0);
9     vec3 diffuse = light.diffuse * (diff * material.diffuse);
10
11    // specular
12    vec3 viewDir = normalize(viewPos - FragPos);
13    vec3 reflectDir = reflect(-lightDir, norm);
14    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
15    vec3 specular = light.specular * (spec * material.specular);
16
17    vec3 result = ambient + diffuse + specular;
18    FragColor = vec4(result, 1.0);
19 }
```

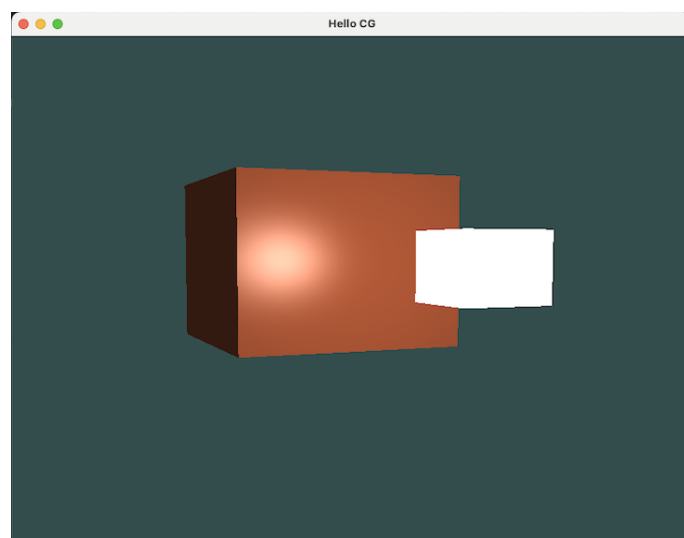
Finally, update the main rendering loop:

```
1 while (!glfwWindowShouldClose(window)) {
2     float currentFrame = glfwGetTime();
3     deltaTime = currentFrame - lastFrame;
4     lastFrame = currentFrame;
```

```

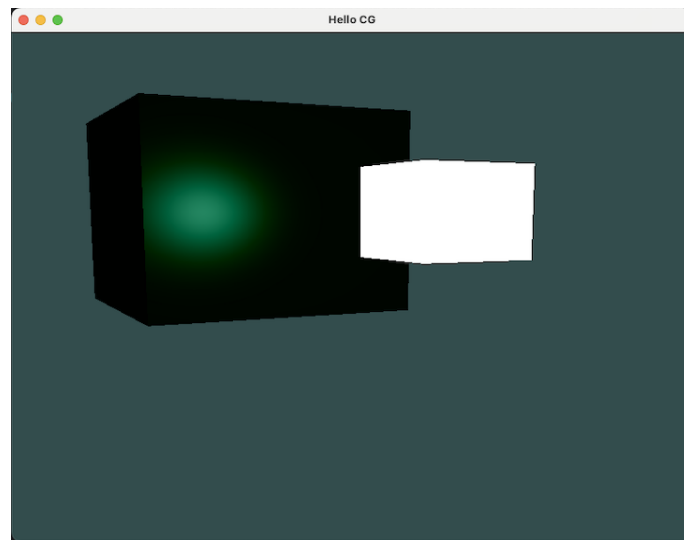
5     camera.ProcessKeyboard(window, deltaTime);
6
7     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
8     glEnable(GL_DEPTH_TEST);
9     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
10
11    cubeShader.use();
12    glm::vec3 lightPos(1.5f, 0.0f, 0.0f);
13
14    glm::mat4 model = glm::mat4(1.0f);
15    glm::mat4 view;
16    view = camera.GetViewMatrix();
17    glm::mat4 projection;
18    projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
19    cubeShader.setUniform("model", model);
20    cubeShader.setUniform("view", view);
21    cubeShader.setUniform("projection", projection);
22
23    cubeShader.setUniform("light.ambient", glm::vec3(0.2f, 0.2f, 0.2f));
24    cubeShader.setUniform("light.diffuse", glm::vec3(0.5f, 0.5f, 0.5f)); // darken diffuse light a bit
25    cubeShader.setUniform("light.specular", glm::vec3(1.0f, 1.0f, 1.0f));
26    cubeShader.setUniform("light.position", lightPos);
27    cubeShader.setUniform("viewPos", camera.Position);
28
29    cubeShader.setUniform("material.ambient", glm::vec3(1.0f, 0.5f, 0.31f));
30    cubeShader.setUniform("material.diffuse", glm::vec3(1.0f, 0.5f, 0.31f));
31    cubeShader.setUniform("material.specular", glm::vec3(0.5f, 0.5f, 0.5f));
32    cubeShader.setUniform("material.shininess", 32.0f);
33    cube.draw();
34
35    lightingCubeShader.use();
36
37    model = glm::mat4(1.0f);
38    model = glm::translate(model, lightPos);
39    model = glm::scale(model, glm::vec3(0.3f));
40
41    lightingCubeShader.setUniform("model", model);
42    lightingCubeShader.setUniform("view", view);
43    lightingCubeShader.setUniform("projection", projection);
44
45    lightingCube.draw();
46
47    glfwSwapBuffers(window);
48    glfwPollEvents();
49 }

```



Since we now have easy access to the light's properties we can change their colors over time to get some really interesting effects. Since everything is already set up in the fragment shader, changing the light's colors is easy and immediately creates some funky effects. We can easily change the light's colors over time by changing the light's ambient and diffuse colors via `sin` and `glfwGetTime`:

```
1 glm::vec3 lightColor;
2 lightColor.x = sin(glfwGetTime() * 2.0f);
3 lightColor.y = sin(glfwGetTime() * 0.7f);
4 lightColor.z = sin(glfwGetTime() * 1.3f);
5
6 glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f);
7 glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f);
8
9 cubeShader.setUniform("light.ambient", ambientColor);
10 cubeShader.setUniform("light.diffuse", diffuseColor); // darken diffuse light a bit
11 cubeShader.setUniform("light.specular", glm::vec3(1.0f, 1.0f, 1.0f));
12 cubeShader.setUniform("light.position", lightPos);
13 cubeShader.setUniform("viewPos", camera.Position);
```



6.4 Lighting Maps

Previously, we gave each object a single material that determined how it reacted to light, which allowed for basic visual differences but limited realism. In reality, most objects consist of multiple materials—for example, a car has shiny paint, reflective windows, dull tires, and glossy rims, each with distinct ambient, diffuse, and specular properties. To better capture this complexity, we extend our material system by introducing diffuse and specular maps, which provide finer control over an object's lighting response and make it possible to simulate the varied surface properties of real-world materials.

6.4.1 Diffuse Map

To control the diffuse color of each fragment on an object, we use a system that assigns color values based on fragment position—essentially, a texture. This is the same concept discussed earlier: an image is wrapped around the object, allowing each fragment to access its own color. In lighting contexts, this texture is called a diffuse map (a common term used by 3D artists before physically based rendering), as it defines the object's surface colors under diffuse lighting. To illustrate this, we'll use a texture of a wooden container with a steel border.



Using a diffuse map in shaders is exactly like we showed in the texture chapter. This time however we store the texture as a `sampler2D` inside the `Material` struct. We replace the earlier defined `vec3` diffuse color vector with the diffuse map.

```
1  in vec2 TexCoords;
2  struct Material {
3      sampler2D diffuse;
4      vec3      specular;
5      float     shininess;
6  };
```

Also, don't forget to set the ambient material's color equal to the diffuse material's color as well:

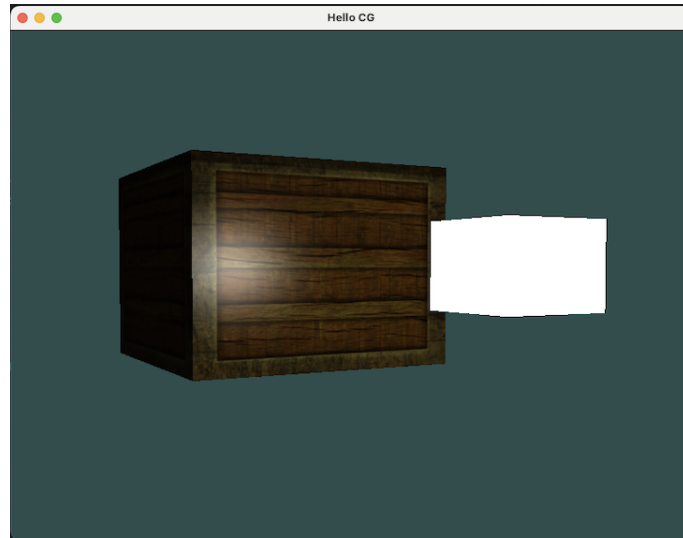
```
1  vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
2  vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
```

For the vertex shader, we need to pass texture coordinates to the fragment shader:

```
1  out vec2 TexCoords;
2
3  void main()
4  {
5      gl_Position = projection * view * model * vec4(aPos, 1.0);
6      FragPos = vec3(model * vec4(aPos, 1.0));
7      Normal = mat3(transpose(inverse(model))) * aNor;
8      TexCoords = aTexCoord;
9  }
```

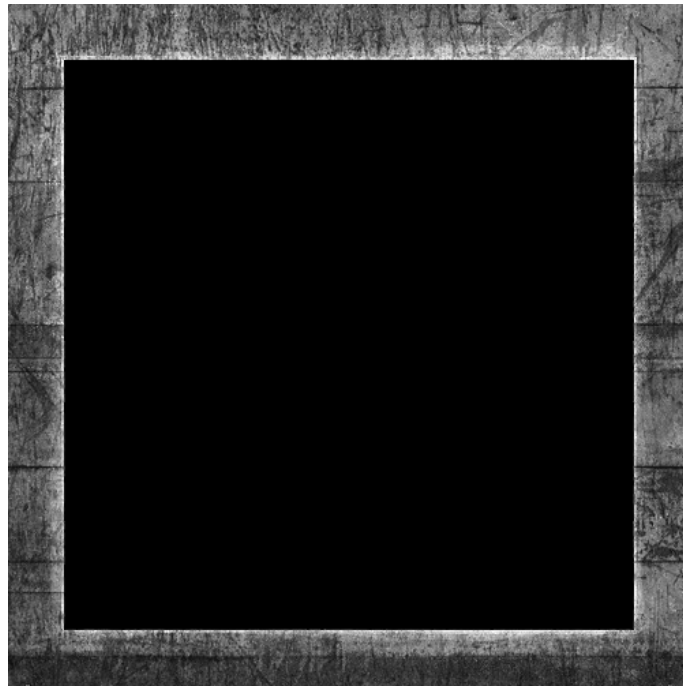
In the main function, we only need to bind the texture:

```
1  cubeShader.setUniform("material.diffuse", glm::vec3(1.0f, 0.5f, 0.31f));
2  cubeShader.setUniform("material.specular", glm::vec3(0.5f, 0.5f, 0.5f));
3  cubeShader.setUniform("material.shininess", 32.0f);
4
5  cubeShader.bindTexture2D("material.diffuse", std::string(ASSETS_DIR) + "container2.png", 0, false);
6  cube.draw();
```



6.4.2 Specular Map

You may notice the specular highlight looks unnatural because the container is mostly wood, which doesn't naturally reflect light that way. Setting the specular material to `vec3(0.0)` would remove the highlights entirely, but that would also eliminate the metallic shine on the steel edges, which should remain reflective. To control which parts of the object show specular highlights and how strong they appear, we can use a specular map—a black-and-white (or colored) texture that defines the intensity of specular reflections across different areas of the surface.



The specular highlight intensity depends on the brightness of each pixel in a specular map, where darker pixels (e.g., black as `vec3(0.0)`) produce little to no reflection and lighter pixels (e.g., white or gray) create stronger highlights. In the fragment shader, we sample the pixel's color and multiply it by the light's specular intensity—brighter pixels yield a more pronounced shine. For example, in a container made of wood and steel, the wooden areas are black in the specular map to indicate no reflection, while the steel edges show varying brightness to represent stronger highlights. Although real wood does have faint, scattered reflections, we simplify by ignoring them here. Specular maps like this can easily be created in image editors such as Photoshop or GIMP by converting a diffuse texture to grayscale and adjusting brightness and contrast.

Then update the material properties of the fragment shader to accept a `sampler2D` as its specular component instead of a `vec3`:

```
1 struct Material {
2     sampler2D diffuse;
```

```

3     sampler2D specular;
4     float      shininess;
5 };
6
7 vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));

```

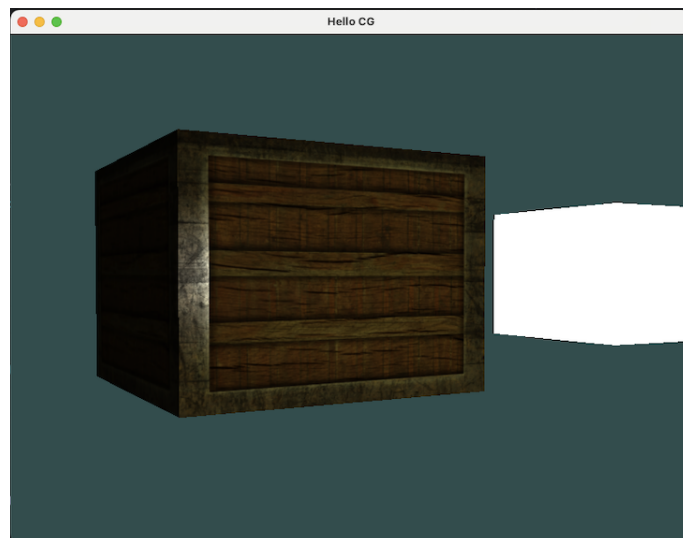
In the main function we need to bind the specular texture:

```

1 cubeShader.bindTexture2D("material.specular", std::string(ASSETS_DIR) + "container2_specular.png", 1, false);

```

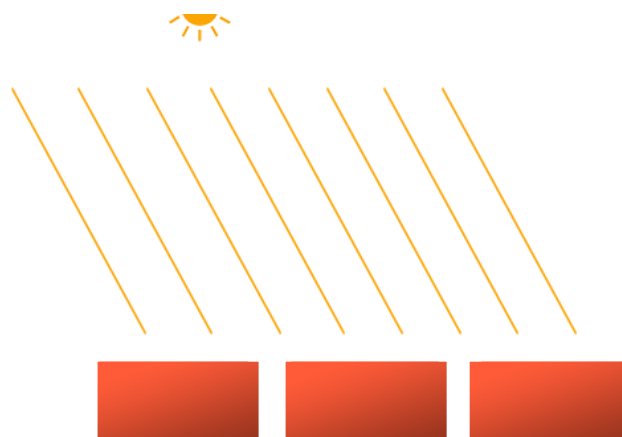
By using a specular map we can specify with enormous detail what parts of an object have shiny properties and we can even control the corresponding intensity. Specular maps give us an added layer of control over lighting on top of the diffuse map.



6.5 Light Casters

So far, our lighting has come from a single point source, which works well but doesn't fully reflect how light behaves in the real world. In reality, different types of lights interact with objects in unique ways. A light source that illuminates objects is called a light caster, and in this chapter, we'll explore three main types: directional lights, point lights (an extension of our previous setup), and spotlights. Understanding how to simulate these different light sources will help you create richer and more realistic environments.

6.5.1 Directional Light



When a light source is very far away, its rays appear nearly parallel, making it seem as though all light comes from the same direction regardless of the object's or viewer's position. Such a source is modeled as a directional light, meaning its rays share a constant direction rather than originating from a specific point. **The sun is a classic example**—though

not infinitely distant, it is far enough that its rays can be treated as parallel in lighting calculations. Because the light direction is uniform, every object in the scene experiences the same lighting behavior. In shaders, this type of light is represented by a direction vector instead of a position vector, simplifying calculations since the light direction is already known.

```

1  struct DirLight {
2      vec3 direction;
3
4      vec3 ambient;
5      vec3 diffuse;
6      vec3 specular;
7  };
8  uniform DirLight dirLight;
9  vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
10
11  ... ...
12  void main()
13  {
14      // properties
15      vec3 norm = normalize(Normal);
16      vec3 viewDir = normalize(viewPos - FragPos);
17
18      //directional lighting
19      vec3 result = CalcDirLight(dirLight, norm, viewDir);
20      FragColor = vec4(result, 1.0);
21  }
22
23  vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
24  {
25      vec3 lightDir = normalize(-light.direction);
26      // diffuse shading
27      float diff = max(dot(normal, lightDir), 0.0);
28      // specular shading
29      vec3 reflectDir = reflect(-lightDir, normal);
30      float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
31      // combine results
32      vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
33      vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
34      vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
35      return (ambient + diffuse + specular);
36  }

```

We negate the light direction vector because lighting calculations expect a direction from the fragment toward the light source, while a directional light is usually defined as pointing outward from the source. Negating it reverses the direction so it correctly points toward the light. We also normalize the vector to ensure it has unit length, as relying on an unnormalized vector can cause incorrect results. The resulting lightDir vector is then used in the diffuse and specular lighting calculations. To show that a directional light affects multiple objects equally, we revisit the container scene with ten containers, each having its own model matrix for proper local-to-world transformations.

```

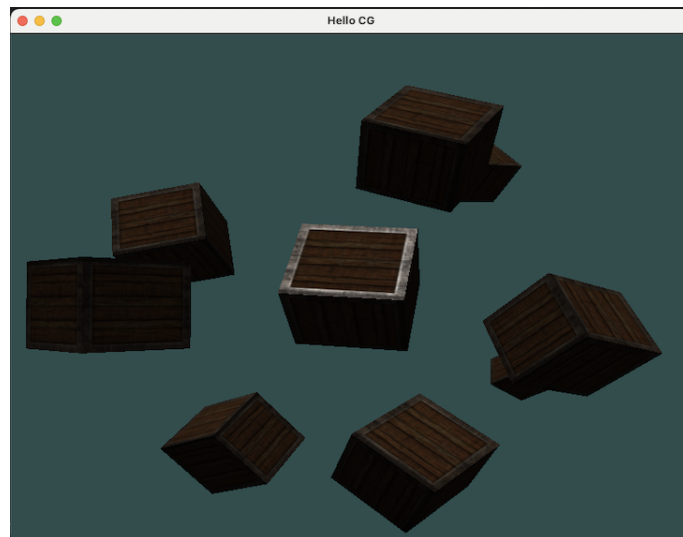
1
2  glm::vec3 cubePositions[] = {
3      glm::vec3( 0.0f,  0.0f,  0.0f),
4      glm::vec3( 2.0f,  5.0f, -15.0f),
5      glm::vec3(-1.5f, -2.2f, -2.5f),
6      glm::vec3(-3.8f, -2.0f, -12.3f),
7      glm::vec3( 2.4f, -0.4f, -3.5f),
8      glm::vec3(-1.7f,  3.0f, -7.5f),
9      glm::vec3( 1.3f, -2.0f, -2.5f),
10     glm::vec3( 1.5f,  2.0f, -2.5f),
11     glm::vec3( 1.5f,  0.2f, -1.5f),
12     glm::vec3(-1.3f,  1.0f, -1.5f)
13 };
14 while (!glfwWindowShouldClose(window)) {
15     ... ...

```

```

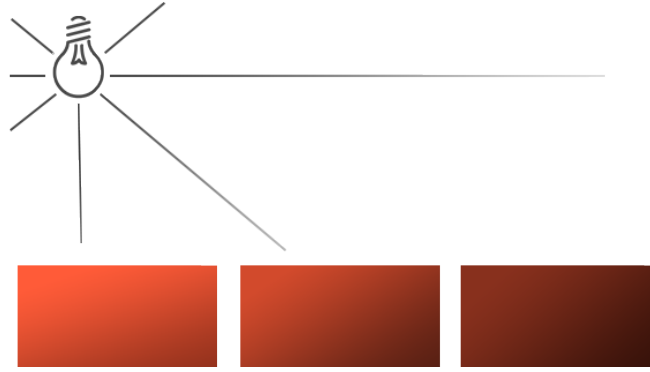
16     cubeShader.use();
17
18     glm::mat4 model = glm::mat4(1.0f);
19     glm::mat4 view = camera.GetViewMatrix();
20     glm::mat4 projection = camera.GetProjection(SCR_WIDTH/SCR_LENGTH);
21     cubeShader.setUniform("model", model);
22     cubeShader.setUniform("view", view);
23     cubeShader.setUniform("projection", projection);
24
25     //     cubeShader.setUniform("light.position", lightPos);
26     //     cubeShader.setUniform("light.direction", glm::vec3(-0.2f, -1.0f, -0.3f));
27     cubeShader.setUniform("viewPos", camera.Position);
28
29     cubeShader.setUniform("dirLight.direction", glm::vec3(-0.2f, -1.0f, -0.3f));
30     cubeShader.setUniform("dirLight.ambient", glm::vec3(0.05f, 0.05f, 0.05f));
31     cubeShader.setUniform("dirLight.diffuse", glm::vec3(0.4f, 0.4f, 0.4f));
32     cubeShader.setUniform("dirLight.specular", glm::vec3(0.5f, 0.5f, 0.5f));
33
34     cubeShader.setUniform("material.shininess", 32.0f);
35     cubeShader.bindTexture2D("material.diffuse", std::string(ASSETS_DIR) + "container2.png", 0, false);
36     cubeShader.bindTexture2D("material.specular", std::string(ASSETS_DIR) + "container2_specular.png", 1, false);
37
38     for (unsigned int i = 0; i < 10; i++){
39         // calculate the model matrix for each object and pass it to shader before drawing
40         glm::mat4 model = glm::mat4(1.0f);
41         model = glm::translate(model, cubePositions[i]);
42         float angle = 20.0f * i;
43         model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
44         cubeShader.setUniform("model", model);
45         cube.draw();
46     }
47     ... ...
48 }

```



6.5.2 Point Light

A point light is a light source positioned at a specific location in the scene that emits light uniformly in all directions, with its intensity diminishing over distance. Common examples include light bulbs or torches, which serve as typical point light sources.



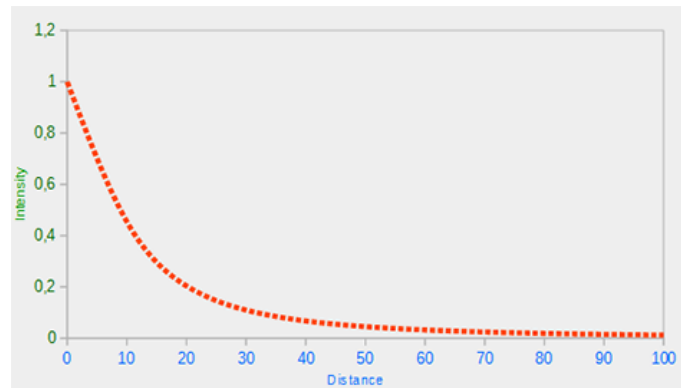
Previously, we used a simple point light that emitted light equally in all directions without fading, making it appear unrealistically strong. In real 3D scenes, however, a light source should only illuminate nearby objects while distant ones appear dimmer. For example, if you add multiple containers to the scene, the ones farther away should receive less light than those close to the source, but our current setup lights them all with equal intensity.

Attenuation refers to the gradual reduction of light intensity as it travels over a distance. A simple linear equation can model this effect by decreasing brightness proportionally with distance, but such a method appears unrealistic. In reality, light is very bright near its source, fades rapidly at first, and then decreases more slowly as the distance increases. To simulate this natural behavior, we use a more accurate attenuation formula:

$$F_{att} = \frac{1.0}{K_c + K_l \cdot d + K_q \cdot d^2}$$

where d is the distance from the fragment to the light source, and K_c , K_l , and K_q are the constant, linear, and quadratic terms that control how quickly the light intensity diminishes.

The constant term, usually set to 1.0, prevents the denominator from becoming less than one and artificially increasing the light intensity. The linear term scales with distance, causing intensity to fade linearly, while the quadratic term scales with the square of the distance, producing a faster, nonlinear decay as distance increases. At close range, the linear term dominates and light diminishes gradually, but as the distance grows, the quadratic term becomes more influential, leading to a sharper falloff before eventually leveling out at far distances.



How should we choose the values for these three terms? Their settings depend on several factors such as the environment, the desired lighting range, and the type of light source. In practice, selecting suitable values often involves experience and some experimentation. The table below provides example values, adapted from Ogre3D's wiki, that simulate realistic light sources over various distances. The constant term K_c remains fixed at 1.0, while the linear term K_l and quadratic term K_q decrease for lights that need to reach farther. Adjusting these parameters helps fine-tune how quickly light intensity fades, and in most environments, a range between 32 and 100 units works well for typical point lights.

Distance	K_c	K_l	K_q
7	1.0	0.7	1.8
13	1.0	0.35	0.44
20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002
3250	1.0	0.0014	0.000007

To implement attenuation, we need three additional values in the fragment shader: the constant, linear, and quadratic terms of the attenuation equation. These parameters are best stored in the `Light` struct defined earlier. Since this is a point light rather than a directional light, we must recalculate the `lightDir` vector using the light's position, as demonstrated in the previous chapter. In your fragment shader program, let's first modify our light struct:

```

1  struct PointLight {
2      vec3 position;
3
4      float constant;
5      float linear;
6      float quadratic;
7
8      vec3 ambient;
9      vec3 diffuse;
10     vec3 specular;
11 };
12
13 uniform PointLight pointLight;
14
15 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
16
17 void main(){
18     // properties
19     vec3 norm = normalize(Normal);
20     vec3 viewDir = normalize(viewPos - FragPos);
21     //pointlight
22     vec3 result = CalcPointLight(pointLight, norm, FragPos, viewDir);
23
24     FragColor = vec4(result, 1.0);
25 }

```

Implementing attenuation in the fragment shader is simple: compute an attenuation factor using the given formula and multiply it with the ambient, diffuse, and specular components. To apply the formula, we need the distance between the fragment and the light source, which can be obtained by subtracting their position vectors and calculating the length of the resulting vector using GLSL's built-in `length` function, and remember we need to recompute light direction:

```

1  vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
2  {
3      vec3 lightDir = normalize(light.position - fragPos);
4      // diffuse shading
5      float diff = max(dot(normal, lightDir), 0.0);
6      // specular shading
7      vec3 reflectDir = reflect(-lightDir, normal);
8      float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
9      // attenuation
10     float distance = length(light.position - fragPos);
11     float attenuation = 1.0 / (light.constant + light.linear * distance +

```

```

12         light.quadratic * (distance * distance));
13     // combine results
14     vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
15     vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
16     vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
17     ambient *= attenuation;
18     diffuse *= attenuation;
19     specular *= attenuation;
20     return (ambient + diffuse + specular);
21 }

```

Then we set the terms in our application: we want the light to cover a distance of 50 so we'll use the appropriate constant, linear and quadratic terms from the table:

```

1  cubeShader.setUniform("pointLight.position", glm::vec3(1.5f, 0.0f, 0.0f));
2  cubeShader.setUniform("pointLight.ambient", glm::vec3(0.05f, 0.05f, 0.05f));
3  cubeShader.setUniform("pointLight.diffuse", glm::vec3(0.8f, 0.8f, 0.8f));
4  cubeShader.setUniform("pointLight.specular", glm::vec3(1.0f, 1.0f, 1.0f));
5  cubeShader.setUniform("pointLight.constant", 1.0f);
6  cubeShader.setUniform("pointLight.linear", 0.09f);
7  cubeShader.setUniform("pointLight.quadratic", 0.032f);
8
9  // also show the lighting source cube
10 lightingCubeShader.use();
11
12 model = glm::mat4(1.0f);
13 model = glm::translate(model, glm::vec3(1.5f, 0.0f, 0.0f));
14 model = glm::scale(model, glm::vec3(0.3f));
15
16 lightingCubeShader.setUniform("model", model);
17 lightingCubeShader.setUniform("view", view);
18 lightingCubeShader.setUniform("projection", projection);

```

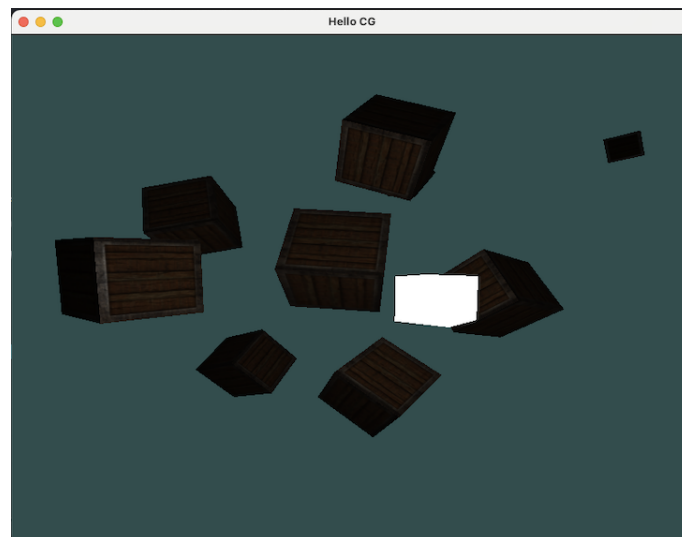


Figure 4: With attenuation

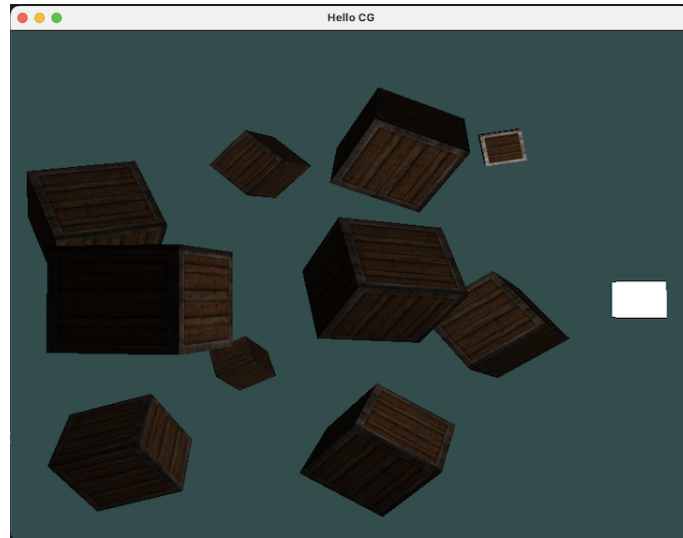
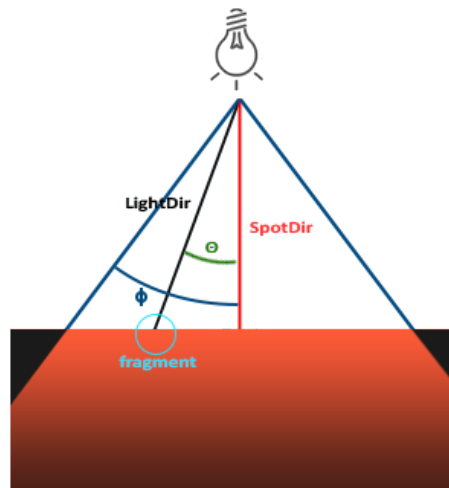


Figure 5: Without attenuation

6.5.3 Spot Light

A **spotlight** is a light source positioned in the scene that emits light only in a specific direction, rather than in all directions. This creates a cone-shaped region of illumination where only objects within the cone are lit, while everything outside remains dark. In OpenGL, a spotlight is defined by its world-space position, direction, and cutoff angle that determines the cone's radius. For each fragment, we check whether it lies within the spotlight's cone and, if so, illuminate it accordingly—similar to how a street lamp or flashlight works.



-
- **LightDir**: the vector pointing from the fragment to the light source.
- **SpotDir**: the direction in which the spotlight is aimed.
- **Cutoff angle** (ϕ): defines the boundary of the spotlight's cone; fragments outside this angle are not illuminated.
- **Angle** (θ): the angle between **LightDir** and **SpotDir**; a fragment is lit only if $\theta < \phi$.

To implement a spotlight, we compute the dot product between the **LightDir** and **SpotDir** vectors, which gives the cosine of the angle between them. We then compare this value with the cosine of the cutoff angle ϕ to determine whether a fragment lies inside the spotlight's cone. With this concept in mind, we can now create a spotlight in the form of a flashlight.

A **flashlight** is a type of spotlight positioned at the viewer's location and aimed forward from the player's perspective. It behaves like a regular spotlight, except its position and direction are continuously updated according to the player's movement and orientation. In the fragment shader, we need the spotlight's position vector (to compute the fragment-to-light direction), its direction vector, and the cutoff angle, all of which can be stored in the **Light** struct.

```

1 struct SpotLight {
2     vec3 position;
3     vec3 direction;
4     float cutOff;
5     float outerCutOff;
6
7     float constant;
8     float linear;
9     float quadratic;
10
11     vec3 ambient;
12     vec3 diffuse;
13     vec3 specular;
14 };
15 uniform SpotLight spotLight;
16 vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir);

```

Instead of passing the cutoff angle directly, we precompute its cosine and send that value to the fragment shader. This is because the dot product between **LightDir** and **SpotDir** in the shader yields a cosine value, not an angle, and comparing an angle with a cosine is invalid. Computing the inverse cosine in the shader to retrieve the angle would be costly, so we avoid it by comparing cosine values instead. Thus, by calculating $\cos(\phi)$ in advance, we can efficiently compare it with the cosine of the angle θ (obtained from the dot product) to determine whether the fragment lies inside or outside the spotlight.

```

1 vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
2 {
3     vec3 lightDir = normalize(light.position - fragPos);
4     // diffuse shading
5     float diff = max(dot(normal, lightDir), 0.0);
6     // specular shading
7     vec3 reflectDir = reflect(-lightDir, normal);
8     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
9     // attenuation
10    float distance = length(light.position - fragPos);
11    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
12    // spotlight intensity
13    float theta = dot(lightDir, normalize(-light.direction));
14    if(theta > light.cutOff){
15        vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
16        vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
17        vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
18        ambient *= attenuation;
19        diffuse *= attenuation;
20        specular *= attenuation;
21        return (ambient + diffuse + specular);
22    }
23    else // else, use ambient light so scene isn't completely dark outside the spotlight.
24    return light.ambient * vec3(texture(material.diffuse, TexCoords));
25 }

```

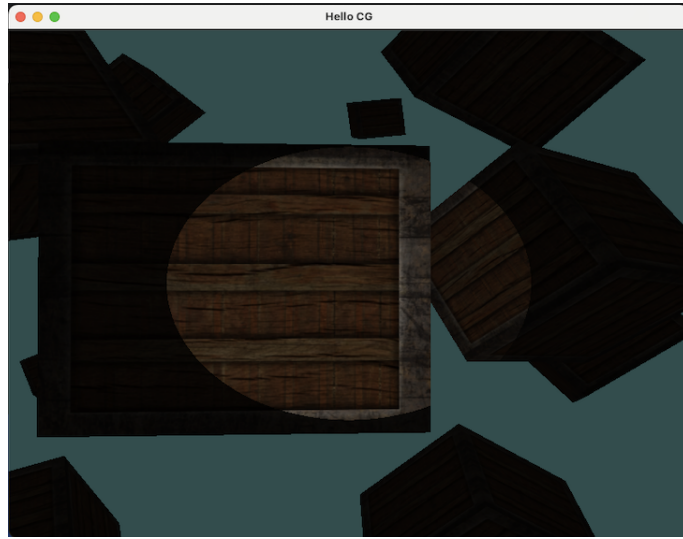
Finally we pass the appropriate values to the shader:

```

1 cubeShader.setUniform("spotLight.position", camera.Position);
2 cubeShader.setUniform("spotLight.direction", camera.Front);
3 cubeShader.setUniform("spotLight.ambient", glm::vec3(0.0f, 0.0f, 0.0f));
4 cubeShader.setUniform("spotLight.diffuse", glm::vec3(1.0f, 1.0f, 1.0f));
5 cubeShader.setUniform("spotLight.specular", glm::vec3(1.0f, 1.0f, 1.0f));
6 cubeShader.setUniform("spotLight.constant", 1.0f);
7 cubeShader.setUniform("spotLight.linear", 0.09f);
8 cubeShader.setUniform("spotLight.quadratic", 0.032f);
9 cubeShader.setUniform("spotLight.cutOff", glm::cos(glm::radians(12.5f)));

```

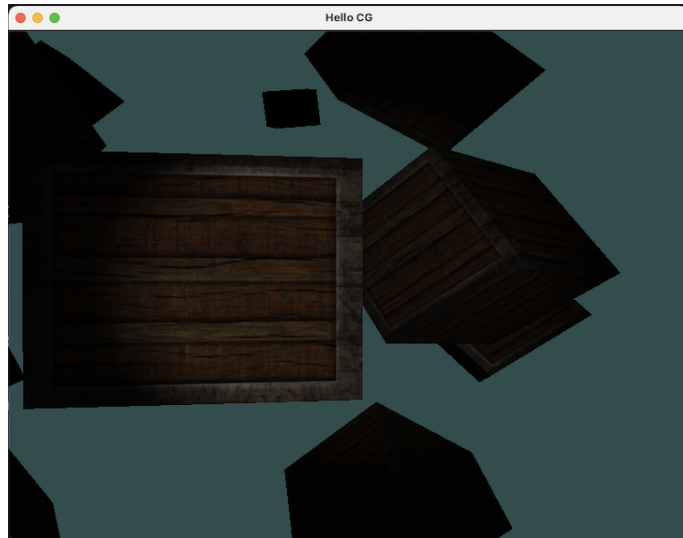
```
10 cubeShader.setUniform("spotLight.outerCutOff", glm::cos(glm::radians(15.0f)));
```



To create a smoothly edged spotlight, we simulate both an inner and an outer cone. The inner cone defines the fully lit region, while the outer cone provides a gradual fade from light to dark. We introduce another cosine value for the outer cone's angle and compute an intensity factor I between 0.0 and 1.0, where fragments inside the inner cone have $I = 1.0$ and those outside the outer cone have $I = 0.0$. Fragments between the two cones are partially lit, with the intensity given by the equation $I = \frac{\theta - \gamma}{\epsilon}$, where $\epsilon = \phi - \gamma$ represents the cosine difference between the inner angle ϕ and the outer angle γ .

We interpolate between the outer and inner cosine values based on the angle θ , producing an intensity value that varies smoothly across the spotlight's cone. This intensity is negative outside the cone, greater than 1.0 inside the inner cone, and transitions gradually near the edges. After clamping the result to the valid range, we can directly multiply the light components by this intensity in the fragment shader without using conditional statements.

```
1  vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
2  {
3      vec3 lightDir = normalize(light.position - fragPos);
4      // diffuse shading
5      float diff = max(dot(normal, lightDir), 0.0);
6      // specular shading
7      vec3 reflectDir = reflect(-lightDir, normal);
8      float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
9      // attenuation
10     float distance = length(light.position - fragPos);
11     float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
12     // spotlight intensity
13     float theta = dot(lightDir, normalize(-light.direction));
14     float epsilon = light.cutOff - light.outerCutOff;
15     float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
16     // combine results
17     vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
18     vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
19     vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
20
21     diffuse *= attenuation * intensity;
22     specular *= attenuation * intensity;
23     return (ambient + diffuse + specular);
24 }
```



To better organize our code, we put all lights and materials configurations into a single header file, which give us a more concise way to organize our main rendering loop:

```

1  //
2  // Created by dengq on 11/18/25.
3  //
4
5  #ifndef DEMO_LIGHTS_CONFIG_H
6  #define DEMO_LIGHTS_CONFIG_H
7  #include <string>
8  #include <vector>
9  #include <glm.hpp>
10
11 #include "shaderprogram.h"
12
13
14 namespace gfx {
15
16     struct DirLightConfig {
17         glm::vec3 direction { -0.2f, -1.0f, -0.3f };
18         glm::vec3 ambient   { 0.05f, 0.05f, 0.05f };
19         glm::vec3 diffuse   { 0.6f, 0.6f, 0.6f };
20         glm::vec3 specular  { 0.8f, 0.8f, 0.8f };
21     };
22
23     struct PointLightConfig {
24         glm::vec3 position { 1.5f, 0.0f, 0.0f };
25
26         float constant = 1.0f;
27         float linear   = 0.09f;
28         float quadratic = 0.032f;
29
30         glm::vec3 ambient { 0.05f, 0.05f, 0.05f };
31         glm::vec3 diffuse { 0.8f, 0.8f, 0.8f };
32         glm::vec3 specular { 1.0f, 1.0f, 1.0f };
33     };
34
35     struct SpotLightConfig {
36         glm::vec3 position { 0.0f, 0.0f, 0.0f };
37         glm::vec3 direction { 0.0f, 0.0f, -1.0f };
38
39         float cutOff      = glm::cos(glm::radians(12.5f));
40         float outerCutOff = glm::cos(glm::radians(17.5f));
41
42         float constant = 1.0f;
43         float linear   = 0.09f;
44         float quadratic = 0.032f;

```

```

45
46     glm::vec3 ambient { 0.0f, 0.0f, 0.0f };
47     glm::vec3 diffuse { 1.0f, 1.0f, 1.0f };
48     glm::vec3 specular { 1.0f, 1.0f, 1.0f };
49 };
50
51 struct SceneConfig {
52     std::vector<DirLightConfig>   dirLights;
53     std::vector<PointLightConfig> pointLights;
54     std::vector<SpotLightConfig>  spotLights;
55 };
56
57 // -----
58 // Default scene (1 dir, 1 point, 1 spot { but you can change sizes)
59 // -----
60 inline SceneConfig makeDefaultSceneConfig()
61 {
62     SceneConfig cfg;
63
64     cfg.dirLights.push_back(DirLightConfig{});
65     cfg.pointLights.push_back(PointLightConfig{});
66     cfg.spotLights.push_back(SpotLightConfig{});
67
68     return cfg;
69 }
70
71 // -----
72 // Lights: array-style helpers (variable count via vector::size())
73 // GLSL side: dirLights[i], pointLights[i], spotLights[i]
74 // -----
75
76 inline void applyDirLights(ShaderProgram& shader,
77                             const std::vector<DirLightConfig>& L,
78                             const std::string& arrayName = "dirLights",
79                             const std::string& countName = "numDirLights")
80 {
81     shader.use();
82     int n = static_cast<int>(L.size());
83     shader.setUniform(countName, n);      // can be 0
84     for (int i = 0; i < n; ++i) {
85         std::string base = arrayName + "[" + std::to_string(i) + "]";
86         shader.setUniform(base + ".direction", L[i].direction);
87         shader.setUniform(base + ".ambient",   L[i].ambient);
88         shader.setUniform(base + ".diffuse",   L[i].diffuse);
89         shader.setUniform(base + ".specular",  L[i].specular);
90     }
91 }
92
93 inline void applyPointLights(ShaderProgram& shader,
94                              const std::vector<PointLightConfig>& L,
95                              const std::string& arrayName = "pointLights",
96                              const std::string& countName = "numPointLights")
97 {
98     shader.use();
99     int n = static_cast<int>(L.size());
100    shader.setUniform(countName, n);      // can be 0
101    for (int i = 0; i < n; ++i) {
102        std::string base = arrayName + "[" + std::to_string(i) + "]";
103        shader.setUniform(base + ".position", L[i].position);
104        shader.setUniform(base + ".constant", L[i].constant);
105        shader.setUniform(base + ".linear",   L[i].linear);
106        shader.setUniform(base + ".quadratic", L[i].quadratic);
107        shader.setUniform(base + ".ambient",  L[i].ambient);
108        shader.setUniform(base + ".diffuse",  L[i].diffuse);
109        shader.setUniform(base + ".specular", L[i].specular);
110    }

```

```

111     }
112
113     inline void applySpotLights(ShaderProgram& shader,
114                                const std::vector<SpotLightConfig>& L,
115                                const std::string& arrayName = "spotLights",
116                                const std::string& countName = "numSpotLights")
117     {
118         shader.use();
119         int n = static_cast<int>(L.size());
120         shader.setUniform(countName, n);           // can be 0
121         for (int i = 0; i < n; ++i) {
122             std::string base = arrayName + "[" + std::to_string(i) + "]";
123             shader.setUniform(base + ".position",   L[i].position);
124             shader.setUniform(base + ".direction",  L[i].direction);
125             shader.setUniform(base + ".cutOff",      L[i].cutOff);
126             shader.setUniform(base + ".outerCutOff", L[i].outerCutOff);
127             shader.setUniform(base + ".constant",   L[i].constant);
128             shader.setUniform(base + ".linear",     L[i].linear);
129             shader.setUniform(base + ".quadratic",  L[i].quadratic);
130             shader.setUniform(base + ".ambient",    L[i].ambient);
131             shader.setUniform(base + ".diffuse",     L[i].diffuse);
132             shader.setUniform(base + ".specular",    L[i].specular);
133         }
134     }
135
136 } // namespace gfx
137
138 #endif //DEMO_LIGHTS_CONFIG_H
139

```

We need to do a few more modifications in our fragment shader program to allow us create multiple lighting objects:

```

1  #version 410 core
2
3  out vec4 FragColor;
4
5  in vec3 Normal;
6  in vec3 FragPos;
7  in vec2 TexCoords;
8
9  // -----
10 // Structs (same layout as in your C++ config)
11 // -----
12 struct Material {
13     sampler2D diffuse;
14     sampler2D specular;
15     float shininess;
16     float alpha;
17 };
18
19 uniform Material material;
20
21 struct DirLight {
22     vec3 direction;
23
24     vec3 ambient;
25     vec3 diffuse;
26     vec3 specular;
27 };
28
29 struct PointLight {
30     vec3 position;
31

```

```

32     float constant;
33     float linear;
34     float quadratic;
35
36     vec3 ambient;
37     vec3 diffuse;
38     vec3 specular;
39 };
40
41 struct SpotLight {
42     vec3 position;
43     vec3 direction;
44     float cutOff;
45     float outerCutOff;
46
47     float constant;
48     float linear;
49     float quadratic;
50
51     vec3 ambient;
52     vec3 diffuse;
53     vec3 specular;
54 };
55
56 // -----
57 // Light arrays + counters (match C++: applyDirLights / applyPointLights / applySpotLights)
58 // -----
59 #define MAX_DIR_LIGHTS 4
60 #define MAX_POINT_LIGHTS 8
61 #define MAX_SPOT_LIGHTS 4
62
63 uniform int numDirLights;
64 uniform int numPointLights;
65 uniform int numSpotLights;
66
67 uniform DirLight  dirLights[MAX_DIR_LIGHTS];
68 uniform PointLight pointLights[MAX_POINT_LIGHTS];
69 uniform SpotLight spotLights[MAX_SPOT_LIGHTS];
70
71 uniform vec3 viewPos;
72
73 // -----
74 // Function declarations
75 // -----
76 vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
77 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
78 vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
79
80 // (optional depth helper { unused here})
81 float near = 0.1;
82 float far  = 100.0;
83 float LinearizeDepth(float depth)
84 {
85     float z = depth * 2.0 - 1.0; // back to NDC
86     return (2.0 * near * far) / (far + near - z * (far - near));
87 }
88
89 // -----
90 // main: accumulate all lights, then apply alpha from Material
91 // -----
92 void main()
93 {
94     vec3 norm    = normalize(Normal);
95     vec3 viewDir = normalize(viewPos - FragPos);
96
97     vec3 result = vec3(0.0);

```

```

98
99 // Directional lights
100 for (int i = 0; i < numDirLights; ++i) {
101     result += CalcDirLight(dirLights[i], norm, viewDir);
102 }
103
104 // Point lights
105 for (int i = 0; i < numPointLights; ++i) {
106     result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
107 }
108
109 // Spot lights
110 for (int i = 0; i < numSpotLights; ++i) {
111     result += CalcSpotLight(spotLights[i], norm, FragPos, viewDir);
112 }
113
114 // Final color: RGB from lighting, A from material.alpha
115 FragColor = vec4(result, material.alpha);
116 }
117
118 // -----
119 // Light calculations (unchanged except they use material + TexCoords)
120 // -----
121 vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
122 {
123     vec3 lightDir = normalize(-light.direction);
124     float diff = max(dot(normal, lightDir), 0.0);
125
126     vec3 reflectDir = reflect(-lightDir, normal);
127     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
128
129     vec3 texDiffuse = vec3(texture(material.diffuse, TexCoords));
130     vec3 texSpec     = vec3(texture(material.specular, TexCoords));
131
132     vec3 ambient  = light.ambient * texDiffuse;
133     vec3 diffuse  = light.diffuse * diff * texDiffuse;
134     vec3 specular = light.specular * spec * texSpec;
135
136     return ambient + diffuse + specular;
137 }
138
139 vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
140 {
141     vec3 lightDir = normalize(light.position - fragPos);
142     float diff = max(dot(normal, lightDir), 0.0);
143
144     vec3 reflectDir = reflect(-lightDir, normal);
145     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
146
147     float distance    = length(light.position - fragPos);
148     float attenuation = 1.0 /
149         (light.constant +
150          light.linear * distance +
151          light.quadratic * (distance * distance));
152
153     vec3 texDiffuse = vec3(texture(material.diffuse, TexCoords));
154     vec3 texSpec     = vec3(texture(material.specular, TexCoords));
155
156     vec3 ambient  = light.ambient * texDiffuse;
157     vec3 diffuse  = light.diffuse * diff * texDiffuse;
158     vec3 specular = light.specular * spec * texSpec;
159
160     ambient *= attenuation;
161     diffuse *= attenuation;
162     specular *= attenuation;
163

```

```

164     return ambient + diffuse + specular;
165 }
166
167 vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
168 {
169     vec3 lightDir = normalize(light.position - fragPos);
170     float diff = max(dot(normal, lightDir), 0.0);
171
172     vec3 reflectDir = reflect(-lightDir, normal);
173     float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
174
175     float distance = length(light.position - fragPos);
176     float attenuation = 1.0 /
177         (light.constant +
178         light.linear * distance +
179         light.quadratic * (distance * distance));
180
181     float theta = dot(lightDir, normalize(-light.direction));
182     float epsilon = light.cutOff - light.outerCutOff;
183     float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
184
185     vec3 texDiffuse = vec3(texture(material.diffuse, TexCoords));
186     vec3 texSpec = vec3(texture(material.specular, TexCoords));
187
188     vec3 ambient = light.ambient * texDiffuse;
189     vec3 diffuse = light.diffuse * diff * texDiffuse;
190     vec3 specular = light.specular * spec * texSpec;
191
192     ambient *= attenuation * intensity;
193     diffuse *= attenuation * intensity;
194     specular *= attenuation * intensity;
195
196     return ambient + diffuse + specular;
197 }

```

In our main function, for example, we only want to use one point light:

```

1     std::string vertPath = std::string(SHADER_DIR) + "cube_vertex.vert";
2     std::string fragPath = std::string(SHADER_DIR) + "container_fragment.frag";
3     ShaderProgram containerShaderProgram(vertPath, fragPath);
4     GLuint cube_diffuse = containerShaderProgram.bindTexture2D("material.diffuse",
5         std::string(ASSETS_DIR) + "container2.png", 0, false);
6     GLuint cube_specular = containerShaderProgram.bindTexture2D("material.specular",
7         std::string(ASSETS_DIR) + "container2_specular.png", 1, false);
8     containerShaderProgram.setUniform("material.shininess", 32.0f);
9     containerShaderProgram.setUniform("material.alpha", 1.0f);
10
11     ... ..
12
13     gfx::SceneConfig cfg = gfx::makeDefaultSceneConfig();
14
15     cfg.dirLights.clear();
16     cfg.pointLights.resize(1);
17     cfg.spotLights.clear();
18     gfx::applyDirLights(containerShaderProgram, cfg.dirLights);
19     gfx::applyPointLights(containerShaderProgram, cfg.pointLights);
20     gfx::applySpotLights(containerShaderProgram, cfg.spotLights);
21
22     while (!glfwWindowShouldClose(window)) {
23         float currentFrame = glfwGetTime();
24         deltaTime = currentFrame - lastFrame;
25         lastFrame = currentFrame;
26         camera.ProcessKeyboard(window, deltaTime);
27

```

```

28     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
29     glEnable(GL_DEPTH_TEST);
30     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
31
32     //objects
33     containerShaderProgram.use();
34     glm::mat4 view      = camera.GetViewMatrix();
35     glm::mat4 projection = camera.GetProjection((float)SCR_WIDTH / SCR_HEIGHT);
36
37     containerShaderProgram.setUniform("view",      view);
38     containerShaderProgram.setUniform("projection", projection);
39     containerShaderProgram.setUniform("viewPos",   camera.Position);
40
41     for (unsigned int i = 0; i < 10; ++i) {
42         glm::mat4 model = glm::mat4(1.0f);
43         model = glm::translate(model, cubePositions[i]);
44         float angle = 20.0f * i;
45         model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
46         containerShaderProgram.setUniform("model", model);
47         container.draw();
48     }
49
50     //just for point lights
51     lightShaderProgram.use();
52     lightShaderProgram.setUniform("view",      view);
53     lightShaderProgram.setUniform("projection", projection);
54
55     for (int i = 0; i < (int)cfg.pointLights.size(); ++i)
56     {
57         glm::vec3 pos  = cfg.pointLights[i].position;
58
59         // Build model matrix for the small light cube
60         glm::mat4 model = glm::mat4(1.0f);
61         model = glm::translate(model, pos);
62         model = glm::scale(model, glm::vec3(0.2f)); // small cube size
63
64         lightShaderProgram.setUniform("model", model);
65         container.draw();
66     }
67
68     glfwSwapBuffers(window);
69     glfwPollEvents();
70 }

```

7 Advanced OpenGL

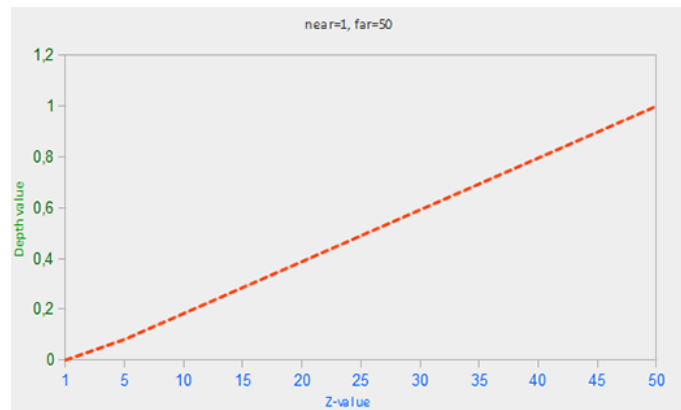
7.1 Depth Testing

In earlier chapters we used the depth buffer to ensure proper occlusion, preventing fragments that should be behind others from being drawn in front. The depth buffer is a per-fragment buffer, similar in size to the color buffer, that stores depth values—typically as 16-, 24-, or 32-bit floats. When depth testing is enabled via `glEnable(GL_DEPTH_TEST)`, OpenGL compares each fragment’s depth (`glFragCoord.z`) to the stored value. If the test passes, the fragment is rendered and the buffer is updated; otherwise, the fragment is discarded. Since leftover depth values cause incorrect results, the buffer must be cleared each frame using `glClear(GL_DEPTH_BUFFER_BIT)`. Modern GPUs often support early depth testing, allowing invisible fragments to be rejected before the fragment shader runs, improving performance as long as the shader does not manually write to fragment depth.

OpenGL also allows control over how depth values are compared through `glDepthFunc`, which sets the depth test function (e.g., `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, etc.). The default, `GL_LESS`, accepts fragments whose depth is smaller than the stored value. In some cases, applications may want to read from the depth buffer without writing new depth values, which can be achieved by disabling depth writes using `glDepthMask(GL_FALSE)` while keeping depth testing active. Together, these mechanisms give fine-grained control over visibility, occlusion, and rendering order in 3D scenes.

The depth buffer stores values in the range $[0, 1]$, but the actual view-space z -values of objects lie between the frustum’s *near* and *far* planes. To compare these values consistently, we must map view-space depths to the $[0, 1]$ interval. A straightforward idea is to apply a linear mapping,

$$F_{\text{depth}} = \frac{z - \text{near}}{\text{far} - \text{near}},$$



which assigns a depth value of 0 to objects on the near plane and 1 to objects on the far plane. Although simple, this linear mapping is not used in real OpenGL pipelines because perspective projection does not preserve linearity in the z -direction.

Before projection: z is in view space. After applying the model and view matrices, a vertex is expressed in view-space coordinates

$$(x_{\text{view}}, y_{\text{view}}, z_{\text{view}}).$$

Here, $z_{\text{view}} < 0$ indicates that the point lies in front of the camera, and the z -value is measured in camera units (meters, centimeters, etc.). In this space the depth is linear, but this linear depth is not suitable for direct use in the depth buffer.

Projection matrix: creation of clip space. The projection matrix transforms the view-space coordinates into clip-space coordinates:

$$(x_{\text{clip}}, y_{\text{clip}}, z_{\text{clip}}, w_{\text{clip}}).$$

For a perspective projection, the matrix is constructed so that

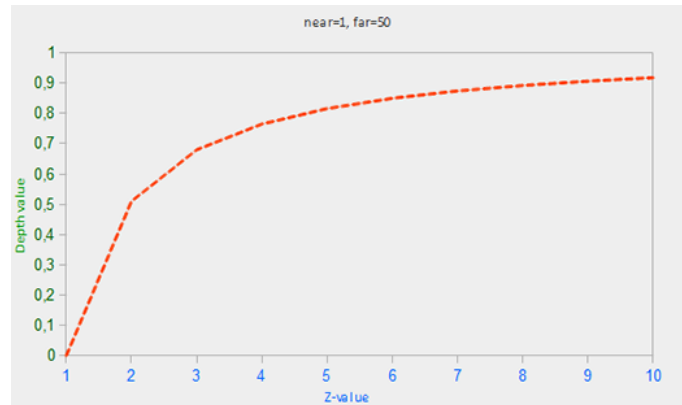
$$z_{\text{clip}} = A \cdot z_{\text{view}} + B, \quad w_{\text{clip}} = -z_{\text{view}},$$

where A and B are constants determined by the near and far plane settings. After this transformation, the GPU performs the perspective divide:

$$z_{\text{ndc}} = \frac{z_{\text{clip}}}{w_{\text{clip}}},$$

which introduces the well-known $1/z$ dependence. This results in a non-linear distribution of depth values, providing significantly higher precision for objects close to the camera. The common form of this non-linear mapping, incorporating both near and far planes, is

$$F_{\text{depth}} = \frac{\frac{1}{z} - \frac{1}{\text{near}}}{\frac{1}{\text{far}} - \frac{1}{\text{near}}}.$$



While the exact derivation may be unintuitive, the key insight is that depth buffer values are **not linear** with respect to distance after projection. A depth value of 0.5 does *not* mean the fragment lies halfway between the near and far planes; in fact, it is still relatively close to the near plane due to the steep curvature of the $1/z$ function. This non-linear transformation is built directly into the projection matrix, so each vertex automatically receives the correct depth value during the view-to-clip and clip-to-screen transformations. The practical impact of this non-linearity becomes clear when visualizing the depth buffer, where near objects dominate the precision budget.

We know that the `z`-component of the built-in `gl_FragCoord` vector contains the depth value of the current fragment. If we simply output this value as a grayscale color, we can visualize the depth buffer directly:

```
1 void main()
2 {
3     FragColor = vec4(vec3(gl_FragCoord.z), 1.0);
4 }
```

However, when running this code, the entire scene often appears almost completely white. This occurs because most depth values are extremely close to 1.0, giving the impression that all fragments lie at the far plane. Why is this the case?

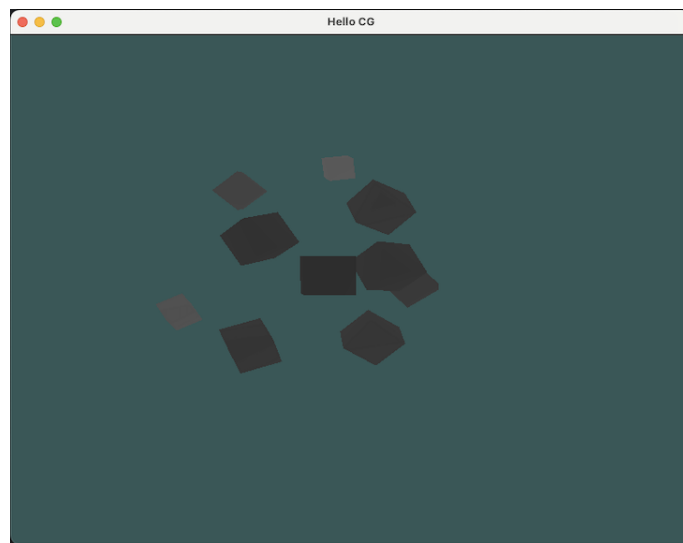
As discussed earlier, depth values stored in screen space are *non-linear*: the depth buffer allocates very high precision to fragments near the camera and dramatically lower precision to fragments farther away. Because the depth increases rapidly with distance, nearly all visible fragments end up with values near 1.0. **Only when the camera moves very close to an object do the depth values noticeably decrease, resulting in darker shades.** This demonstrates the strong non-linearity of the depth distribution, where small movements near the camera cause significant changes in the stored depth, while large movements far away have very little effect.



To properly visualize depth in a perceptually meaningful way, we can convert the non-linear depth values back into their linear counterparts. This requires reversing the steps of the projection transform for depth alone. First, we remap the depth from the $[0, 1]$ range back to normalized device coordinates $[-1, 1]$. Then, by inverting the non-linear perspective equation (as derived from projection matrix), we recover a depth value that varies linearly with distance from the camera.

```
1  float near = 0.1;
2  float far  = 100.0;
3
4  float LinearizeDepth(float depth)
5  {
6      float z = depth * 2.0 - 1.0; // back to NDC
7      return (2.0 * near * far) / (far + near - z * (far - near));
8  }
9
10 void main()
11 {
12     float depth = LinearizeDepth(gl_FragCoord.z) / far; // divide by far for demonstration
13     FragColor = vec4(vec3(depth), 1.0);
14 }
```

Running the application now yields depth values that change linearly with distance. Moving through the scene demonstrates this linear progression clearly. However, most of the resulting colors appear dark because the linear depth spans from a near plane at 0.1 to a far plane at 100, placing the viewer very close to the near plane relative to the full depth range. Consequently, most depth values fall near the lower end of the $[0, 1]$ interval, producing predominantly darker shades.



7.1.1 z-fighting

Z-fighting is a common visual artifact that occurs when two surfaces lie so close together that the depth buffer lacks sufficient precision to determine which one is in front. As a result, the rendered order of the surfaces fluctuates from frame to frame, producing a characteristic flickering or “fighting” pattern. This typically happens when two triangles or planes are nearly coplanar, causing their depth values to be indistinguishably similar.

Z-fighting becomes more noticeable at larger distances from the camera, where the depth buffer has lower precision due to its non-linear distribution. While it cannot be completely eliminated, several practical techniques can greatly reduce its occurrence. The most effective approach is to avoid placing objects so close that their geometry nearly overlaps; even a slight positional offset (such as moving the containers a small amount upward in the y -direction) is often sufficient. Another strategy is to place the near clipping plane as far from the camera as reasonably possible, which increases depth precision across the viewing frustum. Care must be taken, however, not to clip objects that should appear close to the viewer. Finally, using a depth buffer with higher precision—for example, upgrading from a 24-bit to a 32-bit depth format—provides more accurate depth comparisons at the cost of some performance.

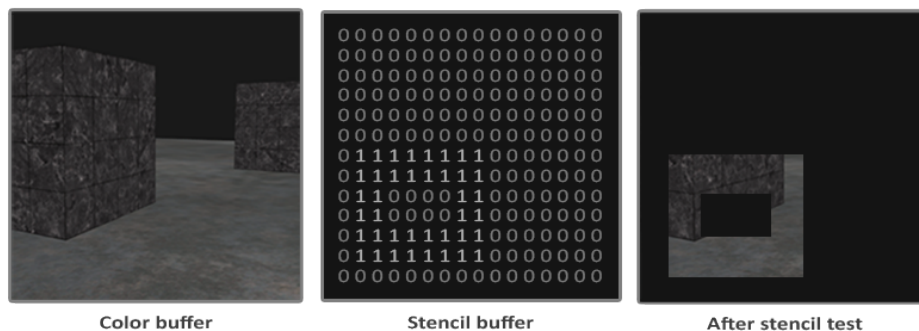
These techniques are the most common and straightforward ways to mitigate Z-fighting. Although more advanced methods exist, they tend to be more complex and still cannot guarantee complete removal of the artifact. In practice,

a combination of positional offsets, careful near-plane placement, and higher depth precision is usually sufficient to minimize Z-fighting in most scenes.

7.2 Stencil Testing

Once a fragment exits the fragment shader, it first undergoes the *stencil test*, which—similar to the depth test—can conditionally discard the fragment. Only fragments that pass the stencil test proceed to the depth test, where OpenGL may discard additional fragments. The stencil test operates on a dedicated *stencil buffer*, an auxiliary buffer that can be modified during rendering to implement masking, outlining, and many other effects.

A stencil buffer typically stores 8 bits per pixel, giving 256 possible stencil values per pixel. We may write arbitrary integer values into this buffer and later configure OpenGL to accept or reject fragments based on these stored values. GLFW automatically creates a stencil buffer as part of the default framebuffer, but other windowing libraries may require explicit configuration. For example, imagine clearing the stencil buffer to zeros and then drawing a hollow rectangle that writes ones into the stencil buffer. Subsequent rendering can be restricted so that fragments are drawn only where the stencil value equals one.



Stencil operations allow us to update the stencil buffer whenever a fragment is rendered. By writing stencil values during one rendering pass and reading them back in the same or subsequent passes, we can control visibility relationships between geometric objects. A typical workflow is as follows:

1. Enable stencil writes.
2. Render one or more objects, thereby updating the stencil buffer.
3. Disable stencil writes.
4. Render additional geometry, this time discarding fragments based on the previously written stencil values.

In this way, the stencil buffer acts as a per-pixel mask that determines where subsequent fragments are allowed to draw.

Stencil testing is enabled by calling:

```
1 glEnable(GL_STENCIL_TEST);
```

Once enabled, each frame must also clear the stencil buffer, just like the color and depth buffers:

```
1 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

OpenGL also provides control over whether stencil values may be written. The function `glStencilMask` defines a bitmask that is ANDed with any stencil value written to the buffer. A mask of `0xFF` allows all bits to be written, while a mask of `0x00` effectively disables stencil writes, analogous to calling `glDepthMask(GL_FALSE)` for the depth buffer:

```
1 glStencilMask(0xFF); // all bits may be written
2 glStencilMask(0x00); // stencil writes disabled
```

Although most applications use only `0x00` or `0xFF`, custom masks are available for more advanced control.

7.2.1 Stencil Functions

Stencil testing provides fine-grained control over whether a fragment should be drawn, similar in spirit to depth testing but operating on an 8-bit stencil buffer. OpenGL exposes two key functions for configuring stencil behavior: `glStencilFunc` and `glStencilOp`. Together, these determine (1) when a fragment passes the stencil test and (2) how the stencil buffer is updated as a result.

The stencil test is configured via

```
glStencilFunc(func, ref, mask),
```

which takes three parameters:

- **func**: the comparison function that determines whether the fragment passes. It compares the masked stencil value to the masked reference value. Possible options include `GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`.
- **ref**: the reference integer used in the comparison.
- **mask**: a bitmask applied to both the stencil buffer value and the reference value before comparison (default: all 1s).

For example:

```
glStencilFunc(GL_EQUAL, 1, 0xFF)
```

allows a fragment to pass only if the stencil value at that pixel is equal to 1; otherwise it is discarded. Note that `glStencilFunc` controls only whether the fragment passes the stencil test—it does not specify how the stencil buffer itself should be modified.

Updates to the stencil buffer are controlled through

```
glStencilOp(sfail, dpfail, dppass),
```

which specifies what action to take in three situations:

- **sfail**: the stencil test fails.
- **dpfail**: the stencil test passes, but the depth test fails.
- **dppass**: both stencil and depth tests pass.

Each of these parameters may take one of the following actions:

<code>GL_KEEP</code>	keep the current stencil value,
<code>GL_ZERO</code>	set the stencil value to 0,
<code>GL_REPLACE</code>	replace with the reference value from <code>glStencilFunc</code> ,
<code>GL_INCR</code>	increment the value (clamped at maximum),
<code>GL_INCR_WRAP</code>	increment with wrap-around,
<code>GL_DECR</code>	decrement the value (clamped at minimum),
<code>GL_DECR_WRAP</code>	decrement with wrap-around,
<code>GL_INVERT</code>	bitwise invert the value.

By default, the stencil operations are

```
(GL_KEEP, GL_KEEP, GL_KEEP),
```

meaning the stencil buffer is never modified regardless of test outcomes. To write new values into the stencil buffer, at least one of the operations must be changed.

7.2.2 Application: Object Outlining

Stencil testing can be difficult to fully grasp at first, so it is helpful to examine a practical technique that relies entirely on the stencil buffer: *object outlining*. The goal of object outlining is to render a visible colored border around one or more objects. This is commonly used in applications such as strategy games, where selected units must be highlighted. The outline effect is created by rendering each object twice while using the stencil buffer to mask out the original pixels.

The outlining procedure works as follows. First, stencil writing is enabled and the stencil operation is configured with `GL_ALWAYS`, so that drawing an object writes the value 1 to the stencil buffer wherever its fragments are rendered. After drawing the original objects, stencil writes are disabled and depth testing is turned off. Each object is then rendered again, but using a model matrix that applies a slight uniform scale and a fragment shader that outputs a single outline color. During this pass, the stencil function is set to accept only fragments whose stencil value is *not* equal to 1, ensuring that the scaled versions draw only outside the regions occupied by the original geometry. Finally, depth testing and the default stencil settings are restored.

This process marks all pixels covered by the original objects with a stencil value of 1. When the scaled versions are drawn, any fragments that lie within the original objects' silhouette fail the stencil test and are discarded, leaving only a thin border visible around the geometry. In this way, the stencil buffer allows us to mask out the interior of the scaled objects and produce a clean outline effect.

```

1  gfx::SceneConfig cfg = gfx::makeDefaultSceneConfig();
2
3  cfg.dirLights.clear();
4  cfg.pointLights.clear();
5  cfg.spotLights.clear();
6
7  gfx::applyDirLights(cubeShader,  cfg.dirLights);
8  gfx::applyPointLights(cubeShader, cfg.pointLights);
9  gfx::applySpotLights(cubeShader,  cfg.spotLights);
10
11 while (!glfwWindowShouldClose(window)) {
12     float currentFrame = glfwGetTime();
13     deltaTime = currentFrame - lastFrame;
14     lastFrame = currentFrame;
15     camera.ProcessKeyboard(window, deltaTime);
16
17     glEnable(GL_DEPTH_TEST);
18     glEnable(GL_STENCIL_TEST);
19     //glStencilFunc(GL_NOTEQUAL, 1, 0xFF);
20     glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
21     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
22     //stencil buffer init to all 0's
23     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
24
25
26     cubeShader.use();
27
28     glm::mat4 view      = camera.GetViewMatrix();
29     glm::mat4 projection = camera.GetProjection((float)SCR_WIDTH / SCR_HEIGHT);
30
31     cubeShader.setUniform("view",      view);
32     cubeShader.setUniform("projection", projection);
33     cubeShader.setUniform("viewPos",   camera.Position);
34
35
36     glStencilMask(0xFF);           // allow writing to stencil
37     glStencilFunc(GL_ALWAYS, 1, 0xFF); // every fragment writes 1
38     for (unsigned int i = 0; i < 10; i++){
39         glm::mat4 model = glm::mat4(1.0f);
40         model = glm::translate(model, cubePositions[i]);
41         float angle = 20.0f * i;
42         model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
43         cubeShader.setUniform("model", model);
44         cube.draw();
45     }
46
47     glStencilFunc(GL_NOTEQUAL, 1, 0xFF); // only draw where stencil != 1
48     glStencilMask(0x00);                 // don't write to stencil
49     glDisable(GL_DEPTH_TEST);           // draw outline on top
50
51     for (unsigned int i = 0; i < 10; i++){
52         lightingCubeShader.use();
53         float scale = 1.1f;
54         glm::mat4 model = glm::mat4(1.0f);
55         model = glm::translate(model, cubePositions[i]);
56         float angle = 20.0f * i;
57         model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
58         model = glm::scale(model, glm::vec3(scale, scale, scale));
59         lightingCubeShader.setUniform("model", model);
60         lightingCubeShader.setUniform("view", view);
61         lightingCubeShader.setUniform("projection", projection);

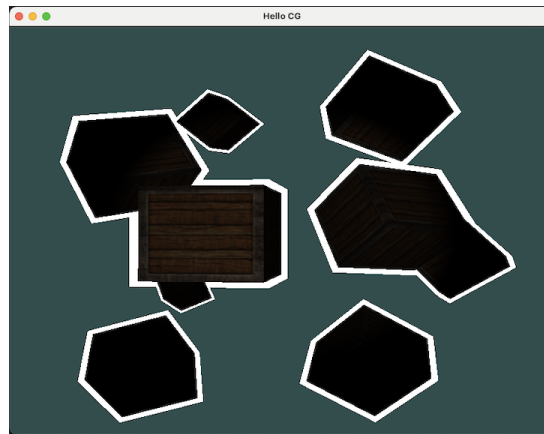
```

```

62     cube.draw();
63 }
64 // ----- Reset state -----
65 glStencilMask(0xFF);
66 glStencilFunc(GL_ALWAYS, 0, 0xFF);
67 glEnable(GL_DEPTH_TEST);
68
69 glfwSwapBuffers(window);
70 glfwPollEvents();
71 }

```

The reason we disable depth test is scaling operation always pushes the surface farther away from the camera where you care about the silhouette. If we do not disable depth testing, then none of these fragments will be able to past depth test.



7.3 Blending

To support partial transparency, OpenGL offers *blending*, which is enabled with:

```

1 glEnable(GL_BLEND);
2 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

```

Once blending is active, OpenGL must also be told how to blend newly generated fragment colors with the color already stored in the color buffer.

Blending in OpenGL is governed by the equation

$$\bar{C}_{\text{result}} = \bar{C}_{\text{source}} \cdot F_{\text{source}} + \bar{C}_{\text{destination}} \cdot F_{\text{destination}},$$

where:

- $\bar{C}_{\text{source}}$ is the fragment color output by the fragment shader,
- $\bar{C}_{\text{destination}}$ is the color currently stored in the color buffer,
- F_{source} and $F_{\text{destination}}$ are scaling factors that determine each color's contribution.

After a fragment shader completes and all associated per-fragment tests succeed, this blend equation is applied to produce the final stored color.

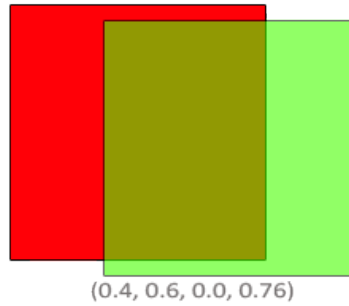
A Simple Example Consider two overlapping quads: an opaque red quad already stored in the color buffer (the destination color) and a semi-transparent green quad drawn on top (the source color). Suppose the green quad has an alpha value of 0.6. We want its contribution to the final color to be 0.6, while the red quad contributes the remaining 0.4 of the final color. This corresponds to choosing

$$F_{\text{source}} = \alpha = 0.6, \quad F_{\text{destination}} = 1 - \alpha = 0.4.$$

The blend equation then becomes

$$\bar{C}_{\text{result}} = \begin{pmatrix} 0.0 \\ 1.0 \\ 0.0 \\ 0.6 \end{pmatrix} (0.6) + \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} (0.4),$$

resulting in a final color that is 60% green and 40% red. This blended color replaces the destination color in the color buffer.



Specifying the Blend Factors OpenGL provides the function

`glBlendFunc(GLenum sfactor, GLenum dfactor)`

to set the source and destination factors; the values are selected from a predefined list, including:

<code>GL_ZERO</code>	<code>factor = 0,</code>
<code>GL_ONE</code>	<code>factor = 1,</code>
<code>GL_SRC_COLOR</code>	<code>factor = $\bar{C}_{\text{source}}$,</code>
<code>GL_ONE_MINUS_SRC_COLOR</code>	<code>factor = $1 - \bar{C}_{\text{source}}$,</code>
<code>GL_DST_COLOR</code>	<code>factor = $\bar{C}_{\text{destination}}$,</code>
<code>GL_ONE_MINUS_DST_COLOR</code>	<code>factor = $1 - \bar{C}_{\text{destination}}$,</code>
<code>GL_SRC_ALPHA</code>	<code>factor = α_{source},</code>
<code>GL_ONE_MINUS_SRC_ALPHA</code>	<code>factor = $1 - \alpha_{\text{source}}$,</code>
<code>GL_DST_ALPHA</code>	<code>factor = $\alpha_{\text{destination}}$,</code>
<code>GL_ONE_MINUS_DST_ALPHA</code>	<code>factor = $1 - \alpha_{\text{destination}}$,</code>
<code>GL_CONSTANT_COLOR</code>	<code>factor = $\bar{C}_{\text{constant}}$,</code>
<code>GL_ONE_MINUS_CONSTANT_COLOR</code>	<code>factor = $1 - \bar{C}_{\text{constant}}$,</code>
<code>GL_CONSTANT_ALPHA</code>	<code>factor = α_{constant},</code>
<code>GL_ONE_MINUS_CONSTANT_ALPHA</code>	<code>factor = $1 - \alpha_{\text{constant}}$.</code>

Using the correct blend factors for the alpha-blended example leads to:

`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`

which implements the traditional and most commonly used form of typical alpha blending.

Now, let's modify our code, first we need to introduce a new attribute into our Material struct:

```
1 struct Material {
2     sampler2D diffuse;
3     sampler2D specular;
4     float shininess;
5
6     // NEW: alpha attribute inside Material
7     float alpha;
8 };
```

Now, we directly render cubes without lighting:

```
1 void main()
2 {
3     FragColor = vec4(vec3(texture(material.diffuse, TexCoords)), material.alpha);
4 }
```

Before we go to modify the main rendering loop, there is one more thing we need to consider:

Why Order Matters for Transparent Objects When using standard alpha blending in OpenGL,

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA),
```

the resulting color stored in the color buffer is computed as

$$\mathbf{C}_{\text{out}} = \alpha \mathbf{C}_{\text{src}} + (1 - \alpha) \mathbf{C}_{\text{dst}},$$

where $\mathbf{C}_{\text{src}}$ is the color of the fragment currently being drawn and $\mathbf{C}_{\text{dst}}$ is the color already present in the framebuffer. This blend equation is *not commutative*: blending object A over B does not produce the same result as blending B over A .

Because of this, overlapping transparent objects must be drawn in a specific order. If the nearest transparent object is drawn first, it blends only with the background; drawing a farther transparent object afterward causes it to blend incorrectly with a color that does not yet contain its own contribution. **To obtain physically correct transparency, transparent objects must therefore be rendered from *farthest to nearest* relative to the camera.**

A correct rendering pipeline involving transparent objects typically follows this sequence:

1. **Draw all opaque objects first.** Depth testing is enabled, blending has no visible effect on opaque surfaces, and no sorting is necessary.
2. **Sort all transparent objects by their distance to the camera.** Transparent objects must be ordered from farthest to nearest.
3. **Draw transparent objects in sorted order.** Depth testing remains enabled, blending is active, and back-to-front rendering ensures correct accumulation of transparency.

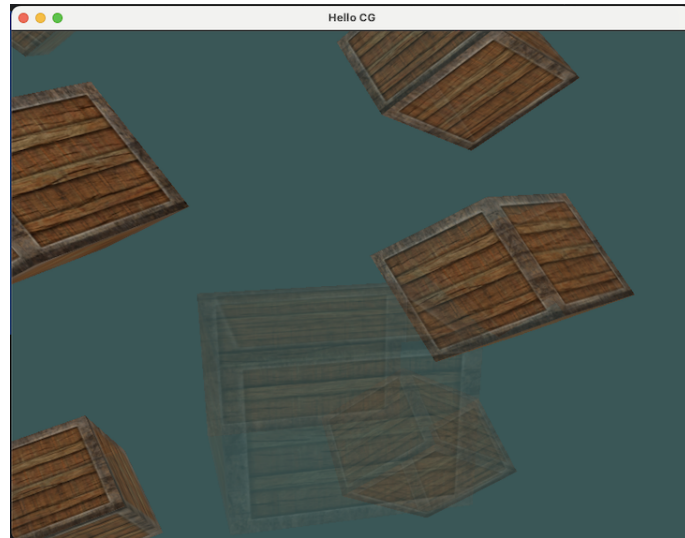
Following this back-to-front ordering guarantees that each transparent surface blends correctly with all surfaces behind it. In the main rendering loop:

```
1  gfx::SceneConfig cfg = gfx::makeDefaultSceneConfig();
2
3  cfg.dirLights.clear();
4  cfg.pointLights.clear();
5  cfg.spotLights.clear();
6  gfx::applyDirLights(cubeShader,  cfg.dirLights);
7  gfx::applyPointLights(cubeShader, cfg.pointLights);
8  gfx::applySpotLights(cubeShader,  cfg.spotLights);
9
10 std::vector<float> cubeAlpha(10);
11 std::random_device rd;
12 std::mt19937 gen(rd());
13 std::uniform_real_distribution<float> dist(0.3f, 1.0f);
14 for (float &v : cubeAlpha) {v = dist(gen);}
15
16 while (!glfwWindowShouldClose(window)) {
17     float currentFrame = glfwGetTime();
18     deltaTime = currentFrame - lastFrame;
19     lastFrame = currentFrame;
20     camera.ProcessKeyboard(window, deltaTime);
21
22     glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
23     glEnable(GL_DEPTH_TEST);
24     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
25     glEnable(GL_BLEND);
26     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
27
28
29     cubeShader.use();
30     glm::mat4 view      = camera.GetViewMatrix();
31     glm::mat4 projection = camera.GetProjection((float)SCR_WIDTH / SCR_HEIGHT);
32
33     cubeShader.setUniform("view",      view);
34     cubeShader.setUniform("projection", projection);
35     cubeShader.setUniform("viewPos",   camera.Position);
36
37
38     // -----
```

```

39 // STEP 1: Draw all opaque cubes (alpha >= 0.95 considered opaque)
40 // -----
41
42 for (int i = 0; i < 10; i++) {
43     if (cubeAlpha[i] >= 0.95f) { // treat near-1.0 alpha as opaque
44         cubeShader.setUniform("material.alpha", cubeAlpha[i]);
45
46         glm::mat4 model = glm::mat4(1.0f);
47         model = glm::translate(model, cubePositions[i]);
48         model = glm::rotate(model, glm::radians(20.0f * i),
49                             glm::vec3(1.0f, 0.3f, 0.5f));
50         cubeShader.setUniform("model", model);
51
52         cube.draw();
53     }
54 }
55
56 // -----
57 // STEP 2: Collect transparent cubes and sort them by distance
58 // -----
59
60 std::vector<std::pair<float, int>> transparentList;
61 transparentList.reserve(10);
62
63 for (int i = 0; i < 10; i++) {
64     if (cubeAlpha[i] < 0.95f) { // treat as transparent
65         float dist = glm::length(camera.Position - cubePositions[i]);
66         transparentList.emplace_back(dist, i);
67     }
68 }
69
70 std::sort(transparentList.begin(), transparentList.end(),
71           [](const auto& a, const auto& b) {
72               return a.first < b.first; // ascending
73           });
74
75 // -----
76 // STEP 3: Draw transparent cubes from far to near
77 // -----
78
79 for (auto it = transparentList.rbegin(); it != transparentList.rend(); ++it) {
80     int idx = it->second;
81
82     cubeShader.setUniform("material.alpha", cubeAlpha[idx]);
83
84     glm::mat4 model = glm::mat4(1.0f);
85     model = glm::translate(model, cubePositions[idx]);
86     model = glm::rotate(model, glm::radians(20.0f * idx),
87                         glm::vec3(1.0f, 0.3f, 0.5f));
88     cubeShader.setUniform("model", model);
89
90     cube.draw();
91 }
92
93 glfwSwapBuffers(window);
94 glfwPollEvents();
95 }

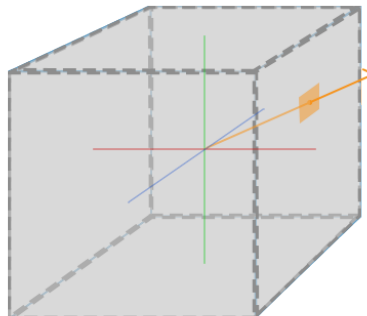
```



We do not combine the lighting here, it is out of the scope and we need to apply a much more complex lighting model.

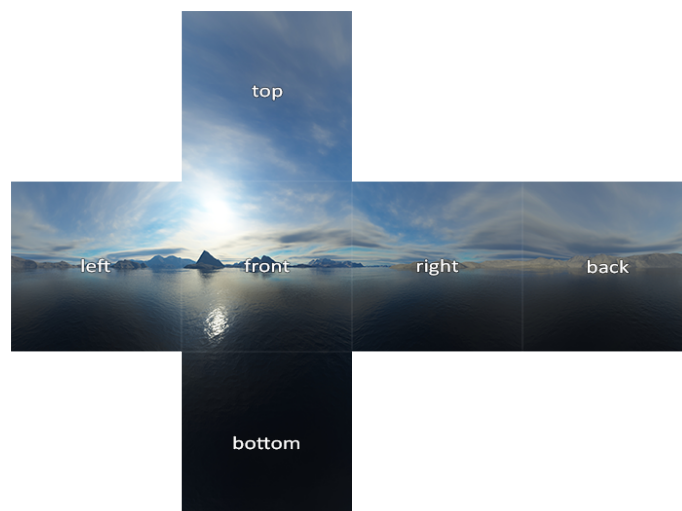
7.4 Cubemaps

A cubemap is a special texture composed of six 2D images forming the sides of a cube, allowing OpenGL to sample it using a *direction vector* rather than traditional 2D coordinates. Because the magnitude of this vector is irrelevant, OpenGL simply determines which face the direction intersects and returns the appropriate texel. When a cube is centered at the origin, its (interpolated) vertex positions naturally act as texture coordinates, enabling each direction vector to select the correct face of the cubemap and making all cube vertices directly usable for sampling.



7.4.1 Skybox

A skybox is a (large) cube that encompasses the entire scene and contains 6 images of a surrounding environment, giving the player the illusion that the environment he's in is actually much larger than it actually is.



We first generate a texture object and bind it to the target `GL_TEXTURE_CUBE_MAP`. In our engine, this process is wrapped inside a helper function of the `ShaderProgram` class, which also binds the sampler uniform and texture unit:

```

1  GLuint ShaderProgram::bindCubemap(const std::string& samplerName,
2                                  const std::vector<std::string>& faces,
3                                  GLint textureUnit,
4                                  bool flipVertical,
5                                  GLint wrap,
6                                  GLint minFilter,
7                                  GLint magFilter) const
8  {
9      if (samplerName.empty()) {
10         std::cerr << "bindCubemap: empty samplerName\n";
11         return 0;
12     }
13     if (faces.size() != 6) {
14         std::cerr << "bindCubemap: expected 6 faces, got "
15                     << faces.size() << "\n";
16         return 0;
17     }
18
19     use();
20     stbi_set_flip_vertically_on_load(flipVertical ? 1 : 0);
21
22     GLuint texID = 0;
23     glGenTextures(1, &texID);
24
25     glActiveTexture(GL_TEXTURE0 + textureUnit);
26     glBindTexture(GL_TEXTURE_CUBE_MAP, texID);

```

Because a cubemap contains six images—one for each face—we must upload each face individually. Each image is associated with a specific cubemap target (`GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, etc.). The helper function loops over a vector of file paths, loads each image, determines the correct internal format, and assigns it to the appropriate face:

```

1      int w = 0, h = 0, ch = 0;
2      for (size_t i = 0; i < faces.size(); ++i) {
3          unsigned char* data = stbi_load(faces[i].c_str(),
4                                          &w, &h, &ch, 0);
5
6          if (!data) {
7              std::cerr << "Failed to load cubemap face: "
8                          << faces[i] << "\n";
9              continue;
10         }
11
12         GLenum srcFormat = GL_RGB;
13         GLint internal = GL_RGB;
14         if (ch == 4) { srcFormat = GL_RGBA; internal = GL_RGBA; }
15         else if (ch == 3) { srcFormat = GL_RGB; internal = GL_RGB; }
16         else if (ch == 1) { srcFormat = GL_RED; internal = GL_RED; }
17
18         glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X
19                     + static_cast<GLenum>(i),
20                     0, internal, w, h, 0,
21                     srcFormat, GL_UNSIGNED_BYTE, data);
22
23         stbi_image_free(data);
24     }

```

OpenGL defines six special cubemap targets, corresponding to cube face orientations:

- `GL_TEXTURE_CUBE_MAP_POSITIVE_X`: Right
- `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`: Left

- GL_TEXTURE_CUBE_MAP_POSITIVE_Y: Top
- GL_TEXTURE_CUBE_MAP_NEGATIVE_Y: Bottom
- GL_TEXTURE_CUBE_MAP_POSITIVE_Z: Back
- GL_TEXTURE_CUBE_MAP_NEGATIVE_Z: Front

These enum values are linearly increasing, so looping from POSITIVE_X through all six faces is straightforward, as shown above.

Because a cubemap is still a texture, we must also specify its wrapping and filtering methods. The helper configures the minification and magnification filters and sets the wrapping mode for all three axes:

```
1  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, minFilter);
2  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, magFilter);
3  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, wrap);
4  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, wrap);
5  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, wrap);
```

The GL_TEXTURE_WRAP_R parameter controls the cubemap's third texture coordinate (its "R" or depth axis). We typically use GL_CLAMP_TO_EDGE to avoid sampling artifacts along cube edges, where texture coordinates may lie between adjacent faces.

Finally, the cubemap sampler must be assigned to the correct texture unit:

```
1  GLint loc = glGetUniformLocation(ID, samplerName.c_str());
2  if (loc >= 0) {
3      glUniform1i(loc, textureUnit);
4  } else {
5      std::cerr << "Warning: samplerCube uniform not found: "
6                << samplerName << "\n";
7  }
8
9  return texID;
10 }
```

A complete bindCubemap function is below:

```
1  GLuint ShaderProgram::bindCubemap(const std::string& samplerName,
2                                   const std::vector<std::string>& faces,
3                                   GLint textureUnit,
4                                   bool flipVertical,
5                                   GLint wrap,
6                                   GLint minFilter,
7                                   GLint magFilter) const
8  {
9      if (samplerName.empty()) {
10         std::cerr << "bindCubemap: empty samplerName\n";
11         return 0;
12     }
13     if (faces.size() != 6) {
14         std::cerr << "bindCubemap: expected 6 faces, got " << faces.size() << "\n";
15         return 0;
16     }
17
18     // Ensure we set uniforms on the correct program
19     use();
20
21     // Usually we do NOT flip cubemap faces, but let the caller decide
22     stbi_set_flip_vertically_on_load(flipVertical ? 1 : 0);
23
24     GLuint texID = 0;
25     glGenTextures(1, &texID);
26 }
```

```

27     glActiveTexture(GL_TEXTURE0 + textureUnit);
28     glBindTexture(GL_TEXTURE_CUBE_MAP, texID);
29
30     int w = 0, h = 0, ch = 0;
31     for (size_t i = 0; i < faces.size(); ++i) {
32         unsigned char* data = stbi_load(faces[i].c_str(), &w, &h, &ch, 0);
33         if (!data) {
34             std::cerr << "Failed to load cubemap face: " << faces[i] << "\n";
35             // You might want to delete texture here, but we'll just continue
36             continue;
37         }
38
39         GLenum srcFormat = GL_RGB;
40         GLint internal = GL_RGB;
41         if (ch == 4) { srcFormat = GL_RGBA; internal = GL_RGBA; }
42         else if (ch == 3) { srcFormat = GL_RGB; internal = GL_RGB; }
43         else if (ch == 1) { srcFormat = GL_RED; internal = GL_RED; }
44
45         glTexImage2D(
46             GL_TEXTURE_CUBE_MAP_POSITIVE_X + static_cast<GLenum>(i),
47             0,
48             internal,
49             w, h,
50             0,
51             srcFormat,
52             GL_UNSIGNED_BYTE,
53             data
54         );
55
56         stbi_image_free(data);
57     }
58
59     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, minFilter);
60     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, magFilter);
61     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, wrap);
62     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, wrap);
63     glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, wrap);
64
65     // Bind sampler uniform to this texture unit
66     GLint loc = glGetUniformLocation(ID, samplerName.c_str());
67     if (loc >= 0) {
68         glUniform1i(loc, textureUnit);
69     } else {
70         std::cerr << "Warning: samplerCube uniform not found: " << samplerName << "\n";
71     }
72
73     return texID;
74 }

```

Remember to declare this in ShaderProgram Class header file:

```

1  GLuint bindCubemap(const std::string& samplerName,
2                      const std::vector<std::string>& faces,
3                      GLint textureUnit,
4                      bool flipVertical = false,
5                      GLint wrap = GL_CLAMP_TO_EDGE,
6                      GLint minFilter = GL_LINEAR,
7                      GLint magFilter = GL_LINEAR) const;

```

Now put all your skybox images into the folder skybox under the assets directory. Before the main rendering loop, we need to first load the skybox texture:

```

1  std::vector<std::string> faces
2      {
3          std::string(ASSETS_DIR) + "skybox/right.jpg",
4          std::string(ASSETS_DIR) + "skybox/left.jpg",
5          std::string(ASSETS_DIR) + "skybox/top.jpg",
6          std::string(ASSETS_DIR) + "skybox/bottom.jpg",
7          std::string(ASSETS_DIR) + "skybox/front.jpg",
8          std::string(ASSETS_DIR) + "skybox/back.jpg"};
9
10 GLuint cubemapTexture = skyboxShader.bindCubemap("skybox", faces, /*textureUnit=*/0);

```

We also need to define shaders to render the skybox. A skybox cubemap can be sampled directly from the cube's local vertex positions. If the cube is centered at the origin, each vertex position is also a direction vector from the center. A cubemap lookup requires exactly such a direction, so the cube's position vectors can be passed directly to the fragment shader as texture coordinates—no explicit UVs are needed.

To render the skybox, we use a minimal vertex shader that forwards the cube's local positions as cubemap directions:

```

1  #version 410 core
2  layout (location = 0) in vec3 aPos;
3
4  out vec3 TexCoords;
5  uniform mat4 projection;
6  uniform mat4 view;
7
8  void main(){
9      TexCoords = aPos;
10     vec4 pos = projection * view * vec4(aPos, 1.0);
11 }

```

The fragment shader simply samples the cubemap using this direction vector:

```

1  #version 410 core
2
3  out vec4 FragColor;
4  in vec3 TexCoords;
5  uniform samplerCube skybox;
6
7  void main(){
8      FragColor = texture(skybox, TexCoords);
9  }

```

Then we need to properly create a ShaderProgram object for our skybox cube:

```

1  std::string skybox_vertPath = std::string(SHADER_DIR) + "skybox_vertex.vert";
2  std::string skybox_fragPath = std::string(SHADER_DIR) + "skybox_fragment.frag";
3  ShaderProgram skyboxShader(skybox_vertPath, skybox_fragPath);
4  Mesh skybox(std::string(ASSETS_DIR) + "skybox.obj", skyboxShader.getID());

```

Rendering the skybox is straightforward: bind the cubemap texture, disable depth writing so the skybox always appears behind the scene, draw the cube, then restore depth writes. However, using the full view matrix causes the skybox to move when the camera moves. To fix this, we remove the translation from the view matrix so only rotation affects the skybox:

```

1  glDepthFunc(GL_LEQUAL); // allow skybox to draw where depth == 1.0
2  skyboxShader.use();
3
4  // remove translation from the view matrix so the skybox stays centered
5  glm::mat4 viewNoTrans = glm::mat4(glm::mat3(view));

```

```

6 skyboxShader.setUniform("view", viewNoTrans);
7 skyboxShader.setUniform("projection", projection);
8 // reuse the same cube mesh as skybox geometry
9 skybox.draw();
10
11 glDepthFunc(GL_LESS); // restore default

```

This keeps the skybox centered around the viewer, preserving the illusion of an infinitely distant environment.

Rendering the skybox first is simple but inefficient: the skybox's fragment shader runs for every pixel, even those later overwritten by scene geometry. A better approach is to render the skybox last, after the depth buffer is fully populated, **so only visible skybox fragments run the shader**. To prevent the skybox cube from passing depth tests incorrectly, we force its depth to always be the farthest possible value. Since depth is computed as $\text{pos.z}/\text{pos.w}$, we can set:

```

1 void main()
2 {
3     TexCoords = aPos;
4     vec4 pos = projection * view * vec4(aPos, 1.0);
5     gl_Position = pos.xyww; // ensures z = w → depth = 1.0
6 }

```

Finally, because the skybox writes depth values of 1.0, we use:

```

1 glDepthFunc(GL_LEQUAL);

```

This allows the skybox to render only where no other geometry is present, while still drawing behind all scene objects. The result is a lightweight, infinite-looking background that greatly enhances scale and realism with minimal shader complexity.

7.4.2 Environment Mapping: Reflection

With a complete cubemap of the surrounding environment, we can use this texture not only for skyboxes but also to create reflective or refractive materials. Such techniques fall under *environment mapping*, where the two most common approaches are reflection and refraction. Reflection simulates surfaces that mirror their surroundings based on the viewer's angle. A reflection vector is computed from the view direction and surface normal (using GLSL's **reflect** function), and this vector is then used to sample the cubemap. The sampled color represents the reflected environment.

Now we want to make the metal boarder of the wooden box even shinier, reflect the skybox color. Since our renderer already includes a skybox, adding reflection is straightforward. Below is the adapted version of our own fragment shader, modified to blend standard Phong lighting with environment-based reflection and use the material's specular map as a reflection mask:

```

1 // Fragment shader (reflection + Phong + specular mask)
2 #version 410 core
3 // ... previous data structure
4
5 uniform samplerCube skybox;
6
7 // ... light structs and declarations omitted for brevity ...
8
9 void main()
10 {
11
12     // ... previous lighting calculation
13     // 2) Environment reflection from cubemap
14     // viewDir points from fragment → camera, so we use reflect(-viewDir, N).
15     vec3 R = reflect(-viewDir, norm);
16     vec3 envColor = texture(skybox, R).rgb;
17

```

```

18 // 3) Use specular map as reflection mask (bright = more reflective)
19 float mask = texture(material.specular, TexCoords).r;
20
21 // 4) Blend Phong result and environment using the mask
22 // - center wood (dark spec map) : pure Phong
23 // - border (bright spec map) : more env reflection
24 vec3 finalColor = mix(result, envColor, mask);
25
26 FragColor = vec4(finalColor, material.alpha);
27 }

```

Before rendering any reflective object (such as your wood box with a reflective border), bind the cubemap texture. In our scene_config.h file:

```

1 struct MaterialConfig {
2     std::string diffuseTexPath;
3     std::string specularTexPath;
4     std::vector<std::string> skyboxTexPath;
5     float shininess = 32.0f;
6     float alpha = 1.0f;
7
8     int diffuseUnit = 0;
9     int specularUnit = 1;
10    int skyboxUnit = 2;
11    bool srgbDiffuse = false;
12    bool srgbSpecular = false;
13 };
14
15 inline SceneConfig makeDefaultSceneConfig()
16 {
17     SceneConfig cfg;
18
19     cfg.material.diffuseTexPath = std::string(ASSETS_DIR) + "container2.png";
20     cfg.material.specularTexPath = std::string(ASSETS_DIR) + "container2_specular.png";
21
22     cfg.material.skyboxTexPath = std::vector<std::string>{
23         std::string(ASSETS_DIR) + "skybox/right.jpg",
24         std::string(ASSETS_DIR) + "skybox/left.jpg",
25         std::string(ASSETS_DIR) + "skybox/top.jpg",
26         std::string(ASSETS_DIR) + "skybox/bottom.jpg",
27         std::string(ASSETS_DIR) + "skybox/front.jpg",
28         std::string(ASSETS_DIR) + "skybox/back.jpg"};
29
30     cfg.dirLights.push_back(DirLightConfig{});
31     cfg.pointLights.push_back(PointLightConfig{});
32     cfg.spotLights.push_back(SpotLightConfig{});
33
34     return cfg;
35 }

```

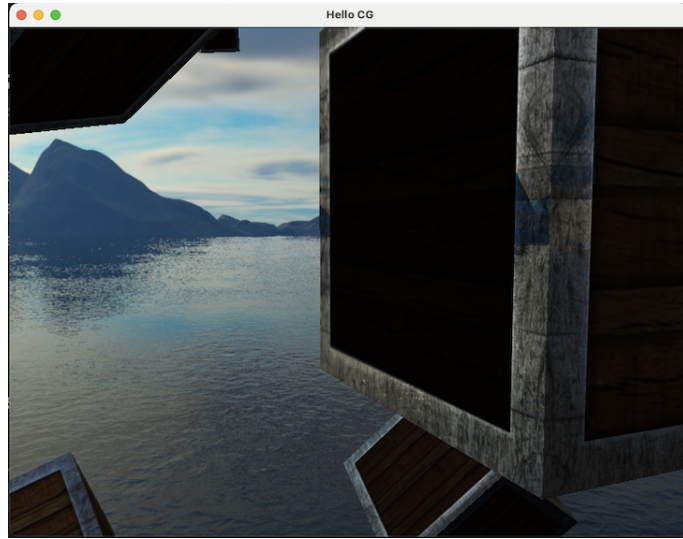
before the main rendering loop, let enable 1 directional light, and setup all material properties:

```

1 gfx::SceneConfig cfg = gfx::makeDefaultSceneConfig();
2
3 cfg.dirLights.resize(1);
4 cfg.pointLights.clear();
5 cfg.spotLights.clear();
6
7 cubeShader.use();
8 gfx::applyMaterial(cubeShader, cfg.material);
9 gfx::applyDirLights(cubeShader, cfg.dirLights);
10 gfx::applyPointLights(cubeShader, cfg.pointLights);

```

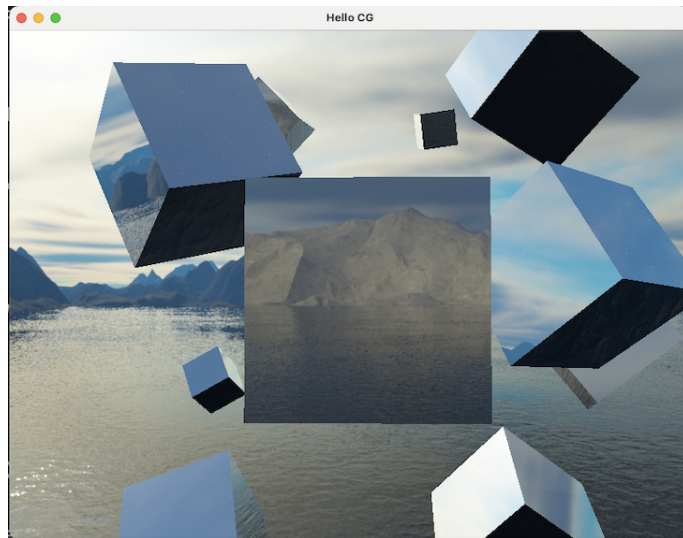
```
gfx::applySpotLights(cubeShader,  cfg.spotLights);
```



If you change your fragment shader's final output to output only skebox environment reflection color:

```
1 FragColor = vec4(envColor, material.alpha);
```

You will get:



7.4.3 Environment Mapping: Refraction

With a complete cubemap representing the surrounding environment, we can extend its use beyond skyboxes to create refractive materials. Refraction describes the bending of light as it passes from one medium to another, governed physically by Snell's law. In the context of environment mapping, a view vector is refracted through the surface normal, producing a new direction that samples the cubemap. Because each medium has its own refractive index—air (1.00), water (1.33), glass (1.52), diamond (2.42)—the distortion can be adjusted by using the ratio of indices. For an object made of glass in air, the refraction ratio is approximately

$$\eta = \frac{1.00}{1.52} \approx 0.658.$$

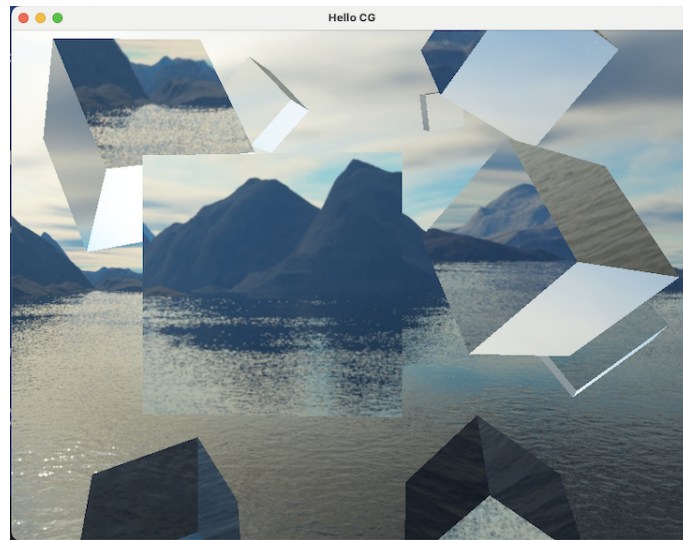
Since our renderer already provides world-space normals, world-space fragment positions, camera position, and a cubemap, implementing glass reduces to modifying the fragment shader so that each box uses cubemap refraction instead of Phong shading.

```

1  #version 410 core
2
3  void main()
4  {
5      vec3 norm    = normalize(Normal);
6      vec3 viewDir = normalize(viewPos - FragPos);
7
8      float eta = 1.00 / 1.52;
9      vec3 R = refract(-viewDir, normalize(Normal), eta);
10     vec3 envColor = texture(skybox, R).rgb;
11
12     FragColor = vec4(envColor, material.alpha);
13 }

```

With these modifications, each box behaves as a glass object: surface normals distort incoming rays according to the refractive index, and the cubemap provides the refracted environment color. Adjusting the refractive index changes the distortion strength, enabling a range of materials from water to diamond.



7.5 Shadows

7.5.1 Preparation

Because we will use a single shader program to render two different models with different textures—especially the plane needed to visualize the shadow—we need a more flexible method for managing texture binding during rendering. Fortunately, each texture has a unique OpenGL ID, which allows us to bind or unbind textures easily and efficiently.

Now, slightly change our texture binding code:

```

1  GLuint cube_diffuse = cubeShader.bindTexture2D("material.diffuse", std::string(ASSETS_DIR) +
2                                             "container2.png", 0, false);
3  GLuint cube_specular = cubeShader.bindTexture2D("material.specular", std::string(ASSETS_DIR) +
4                                             "container2_specular.png", 1, false);
5  GLuint plane_diffuse = cubeShader.bindTexture2D("material.diffuse", std::string(ASSETS_DIR) +
6                                             "wood.png", 0, false);
7  GLuint plane_specular = cubeShader.bindTexture2D("material.specular", std::string(ASSETS_DIR) +
8                                             "wood_specular.png", 1, false);
9  GLuint skyboxID = cubeShader.bindCubemap("skybox", faces, 2);

```

Inside the main rendering loop:

```

1  glActiveTexture(GL_TEXTURE0);
2  glBindTexture(GL_TEXTURE_2D, cube_diffuse);

```

```

3  glActiveTexture(GL_TEXTURE1);
4  glBindTexture(GL_TEXTURE_2D, cube_specular);
5
6  for (unsigned int i = 0; i < 10; i++){
7      glm::mat4 model = glm::mat4(1.0f);
8      model = glm::translate(model, cubePositions[i]);
9      float angle = 20.0f * i;
10     model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
11     cubeShader.setUniform("model", model);
12     cube.draw();
13 }
14
15 glActiveTexture(GL_TEXTURE0);
16 glBindTexture(GL_TEXTURE_2D, plane_diffuse);
17 glActiveTexture(GL_TEXTURE1);
18 glBindTexture(GL_TEXTURE_2D, plane_specular);
19 cubeShader.setUniform("model", glm::mat4(1.0f));
20 plane.draw();

```

7.5.2 Intro to Directional Light Shadow Mapping

Shadow mapping is a widely used real-time shadowing technique, it leverages a depth comparison between the camera's view and the light's view to determine whether a fragment is shadowed. Directional lights (e.g., sunlight) require the use of an orthographic projection rather than perspective projection, because parallel rays have no perspective foreshortening.

Required additions:

- A depth-only framebuffer for the shadow map.
- A first rendering pass from the light's point of view.
- A light-space matrix for transforming world coordinates.
- Shadow lookup and PCF (percentage closer filtering) in the main fragment shader.

To determine whether a surface point is shadowed, we must transform it into the light's clip space. Given a world-space point P_w , the transformation is:

$$P_{\text{light}} = M_{\text{lightProj}} M_{\text{lightView}} P_w,$$

where:

- $M_{\text{lightView}} = \text{lookAt}(L, T, \hat{y})$ defines the light's coordinate frame.
- $M_{\text{lightProj}}$ is an *orthographic* projection:

$$M_{\text{lightProj}} = \text{ortho}(x_{\min}, x_{\max}, y_{\min}, y_{\max}, z_{\min}, z_{\max}).$$

After the homogeneous divide and normalization:

$$P_{\text{ndc}} = \frac{P_{\text{light}}}{w}, \quad P_{\text{uv}} = \frac{P_{\text{ndc}} + 1}{2},$$

the z component is compared with the stored depth value in the shadow map.

A fragment is shadowed if:

$$z_{\text{fragment}} > z_{\text{shadowMap}} + \text{bias}.$$

A bias term is essential to prevent *shadow acne* caused by floating-point rounding. Directional shadow mapping consists of two render passes, the first pass is rendering depth from light's view:

1. Bind a framebuffer object (FBO) with a depth texture.
2. Render the scene using a simplified shader that outputs only depth.
3. Store depths in a 2D texture (the shadow map).

First, let's write a function purely initializes a depth-only framebuffer used to store the shadow map in Pass 1.

```
1 // -----
2 // createShadowMap
3 //
4 // This function initializes a *depth-only framebuffer* used in Pass 1 of
5 // directional shadow mapping. It does NOT render anything; its sole purpose
6 // is to allocate and configure:
7 //
8 // (1) A framebuffer object (FBO)
9 // (2) A depth texture that will store the shadow map
10 //
11 // During the shadow pass, the scene is rendered from the light's point of view,
12 // and gl_FragCoord.z is written into this depth texture.
13 //
14 // The main rendering pass then samples this depth map to determine whether each
15 // fragment is occluded by comparing its light-space depth with the stored value.
16 // -----
17 void createShadowMap(GLuint &depthFBO, GLuint &depthMap,
18                     int SHADOW_WIDTH, int SHADOW_HEIGHT)
19 {
20     // -----
21     // (1) Generate a framebuffer object that will hold ONLY a depth attachment.
22     // No color attachments will be used.
23     // -----
24     glGenFramebuffers(1, &depthFBO);
25
26     // -----
27     // (2) Create a 2D texture to store the shadow map.
28     // This texture stores ONLY depth values (GL_DEPTH_COMPONENT).
29     // -----
30     glGenTextures(1, &depthMap);
31     glBindTexture(GL_TEXTURE_2D, depthMap);
32
33     // -----
34     // Allocate the texture memory:
35     // - Internal format: GL_DEPTH_COMPONENT
36     // - Resolution : SHADOW_WIDTH x SHADOW_HEIGHT
37     // - Type : GL_FLOAT (high precision depth)
38     //
39     // Passing nullptr means \allocate, but don't upload data."
40     // -----
41     glTexImage2D(
42         GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
43         SHADOW_WIDTH, SHADOW_HEIGHT,
44         0, GL_DEPTH_COMPONENT, GL_FLOAT, nullptr
45     );
46
47     // -----
48     // (3) Texture filtering.
49     //
50     // Depth comparisons become unstable if linear filtering is used, so
51     // NEAREST is strongly recommended for both magnification and minification.
52     // -----
53     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
54     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
55
56     // -----
57     // (4) Texture coordinate handling.
58     //
59     // A directional light uses an orthographic projection. When transformed
60     // light-space UVs fall outside the [0, 1] range, we clamp to a border color.
61     //
62     // Using GL_CLAMP_TO_BORDER prevents sampling undefined garbage outside
63     // the texture. The border color determines shadow behavior at the edges.
64     // -----
```

```

65     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
66     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
67
68     // -----
69     // (5) Border color.
70     //
71     // A border depth of 1.0 represents "no occlusion" (maximum depth).
72     // If the shadow lookup goes outside the map, the fragment is treated as lit.
73     // -----
74     float borderColor[] = {1.0f, 1.0f, 1.0f, 1.0f};
75     glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
76
77     // -----
78     // (6) Bind the FBO and attach the depth texture to GL_DEPTH_ATTACHMENT.
79     //
80     // This means: "When we render into this FBO, write depth values into
81     // the depthMap texture."
82     // -----
83     glBindFramebuffer(GL_FRAMEBUFFER, depthFBO);
84     glFramebufferTexture2D(
85         GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
86         GL_TEXTURE_2D, depthMap,
87         0
88     );
89
90     // -----
91     // (7) No color buffer is used for shadow mapping.
92     //
93     // Set both draw/read buffers to NONE to:
94     // - Avoid unnecessary color writes
95     // - Prevent OpenGL warnings or undefined behavior
96     // -----
97     glDrawBuffer(GL_NONE);
98     glReadBuffer(GL_NONE);
99
100    // -----
101    // (8) Unbind the FBO to avoid accidental use elsewhere.
102    // -----
103    glBindFramebuffer(GL_FRAMEBUFFER, 0);
104 }

```

To use this function, we first declare several global constants, put it on top of the main file:

```

1  void createShadowMap(GLuint &depthFBO, GLuint &depthMap, int SHADOW_WIDTH, int SHADOW_HEIGHT);
2
3  GLuint depthFBO;
4  GLuint depthMap;
5  const unsigned int SHADOW_WIDTH = 2048, SHADOW_HEIGHT = 2048;

```

Then, right after GLAD initialization, call the function to init global shadow map:

```

1  createShadowMap(depthFBO, depthMap, SHADOW_WIDTH, SHADOW_HEIGHT);

```

Next, we need shader programs to produce the depth data. Unlike the main shading pass, this depth pass does not compute lighting, materials, or textures. Its sole purpose is to output depth values. For this reason, the depth shaders are intentionally minimal:

```

1  #version 410 core
2  layout (location = 0) in vec3 aPos;
3

```

```

4 uniform mat4 lightSpaceMatrix;
5 uniform mat4 model;
6
7 void main(){
8     gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
9 }

```

```

1 #version 410 core
2 void main(){
3     // depth only; no color output
4 }

```

Before we enter the main rendering loop, we need to compute the directional light's space (MVP):

```

1 // -----
2 // Compute Light-Space Matrix for Directional Shadow Mapping
3 //
4 // This block constructs the transformation matrix that maps world-space
5 // coordinates into the coordinate system of the directional light.
6 // -----
7
8 // -----
9 // (1) Extract the directional light direction.
10 //
11 // The scene configuration stores the light direction in cfg.dirLights[0].
12 // In standard lighting conventions, a directional light \points FROM the
13 // sky TOWARD the scene." For shadow mapping, we want the direction FROM
14 // the light TOWARD the scene, so we take the negation:
15 //
16 //     lightDir = -direction
17 //
18 // This produces a vector that indicates where the directional light is
19 // looking.
20 // -----
21 glm::vec3 lightDir = -cfg.dirLights[0].direction;
22
23 // -----
24 // (2) Construct the orthographic projection matrix.
25 //
26 // Directional lights represent illumination from infinitely far away (like
27 // sunlight). They do not exhibit perspective foreshortening. Therefore,
28 // an ORTHOGRAPHIC projection is required.
29 //
30 // The bounds (-20, 20) define the size of the region of the world that is
31 // projected onto the shadow map. Too small + shadows clipped.
32 // Too large + reduced shadow precision.
33 //
34 // The near/far planes (1.0 to 50.0) define the depth range in light space.
35 // -----
36 glm::mat4 lightProjection = glm::ortho(
37     -20.0f, 20.0f, // left, right
38     -20.0f, 20.0f, // bottom, top
39     1.0f, 50.0f // near, far
40 );
41
42 // -----
43 // (3) Construct the light's view matrix using lookAt.
44 //
45 // The light's \eye position" is chosen by stepping 10 units *backwards*
46 // along the light's direction. This gives the light a vantage point
47 // similar to a camera:
48 //

```

```

49 //     eye = lightDir * 10
50 //
51 // The light looks at the origin (0,0,0) in this example, but this can be
52 // any point or even the center of your scene's bounding box.
53 //
54 // The up vector is (0,1,0), defining the vertical orientation of the light.
55 // -----
56 glm::mat4 lightView = glm::lookAt(
57     lightDir * 10.0f,      // eye position from which the light "views" the scene
58     glm::vec3(0.0f),      // target of the light
59     glm::vec3(0,1,0)      // up direction
60 );
61
62 // -----
63 // (4) Combine projection and view into the light-space matrix.
64 //
65 // Order matters:
66 //
67 //     lightSpaceMatrix = Projection * View
68 //
69 // Any world-space point P_world is transformed into light space by:
70 //
71 //     P_light = lightSpaceMatrix * P_world
72 //
73 // The depth pass uses this matrix in the vertex shader to map geometry
74 // into the shadow map. The main pass uses it in the fragment shader to
75 // compare fragment depth with stored shadow map depth.
76 // -----
77 glm::mat4 lightSpaceMatrix = lightProjection * lightView;

```

In the main rendering loop, we first render the pure depth data in the view of the directional light:

```

1 // -----
2 // PASS 1: RENDER SCENE FROM THE LIGHT'S POINT OF VIEW
3 //
4 // This pass draws the scene into the depth-only framebuffer (depthFBO).
5 // Instead of colors, only depth values are written into the shadow map
6 // texture we allocated earlier. The goal is to capture how far the closest
7 // surfaces are from the light.
8 //
9 // The resulting depth texture will later be sampled in PASS 2 to determine
10 // whether each fragment is shadowed.
11 // -----
12
13 // -----
14 // (1) Activate the depth-only shader program.
15 //
16 // This shader writes ONLY gl_FragCoord.z into the depth texture. It does not
17 // compute lighting, materials, or textures. The vertex shader transforms each
18 // vertex using:
19 //
20 //     gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
21 //
22 // making the geometry appear from the directional light's POV.
23 // -----
24 depthShader.use();
25 depthShader.setUniform("lightSpaceMatrix", lightSpaceMatrix);
26
27 // -----
28 // (2) Set the viewport to match the shadow map resolution.
29 //
30 // Shadow maps are frequently much higher than the window resolution (e.g.
31 // 2048x2048). Setting the viewport ensures that each texel of the depth
32 // texture corresponds to one rasterized fragment.

```

```

33 // -----
34 glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
35
36 // -----
37 // (3) Bind the depth-only framebuffer so depth output writes into depthMap.
38 //
39 // All rendering commands now target the off-screen FBO. Any depth values
40 // produced will populate the depth texture attached to depthFBO.
41 // -----
42 glBindFramebuffer(GL_FRAMEBUFFER, depthFBO);
43
44 // -----
45 // (4) Clear only the depth buffer.
46 //
47 // Since the FBO contains no color attachments, the depth buffer must be
48 // cleared before rendering. Otherwise, stale depth values from prior frames
49 // would remain in the shadow map.
50 // -----
51 glClear(GL_DEPTH_BUFFER_BIT);
52
53 // -----
54 // (5) Begin drawing the scene geometry from the light's viewpoint.
55 //
56 // First draw the plane. The model matrix is the identity for the ground.
57 // -----
58 depthShader.setUniform("model", glm::mat4(1.0f));
59 plane.draw();
60
61 // -----
62 // (6) Draw each cube with its own model transformation.
63 //
64 // Although the depth shader ignores textures and lighting, it MUST receive
65 // the correct model matrix, because the depth test depends on accurate
66 // world-space positioning.
67 //
68 // The transformed position:
69 //
70 //     gl_Position = lightSpaceMatrix * model * vec4(aPos, 1.0);
71 //
72 // determines the corresponding depth in the shadow map.
73 // -----
74 for (int i = 0; i < 10; i++) {
75     glm::mat4 model = glm::mat4(1.0f);
76     model = glm::translate(model, cubePositions[i]);
77     model = glm::rotate(model,
78                         glm::radians(20.0f * i),
79                         glm::vec3(1, 0.3, 0.5));
80
81     depthShader.setUniform("model", model);
82     cube.draw();
83 }
84
85 // -----
86 // (7) Restore default framebuffer.
87 //
88 // At this point, the shadow map (depthMap) is fully populated with depth
89 // values measuring distances from the light to the nearest surfaces.
90 // -----
91 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

Pass 2 is the main rendering pass. Here the scene is rendered from the camera's perspective using the standard lighting pipeline. The shadow map created in Pass 1 is sampled for each fragment to determine whether it lies in direct light or shadow. If the fragment is occluded in light space, its diffuse and specular contributions are reduced, producing visually correct shadows consistent with the directional light.

Let's first modify our cube vertex shader program, we only need to add two lines:

```
1
2 // -----
3 // The light-space transformation matrix.
4 //
5 // This matrix converts world-space positions into the coordinate system of
6 // the directional light. It is the same matrix used during the depth pass
7 // to render the shadow map:
8 //
9 //     lightSpaceMatrix = lightProjection * lightView
10 //
11 // Using the same matrix in the main pass ensures both passes agree on how
12 // world-space points map to shadow-map (u,v,z) coordinates.
13 // -----
14 uniform mat4 lightSpaceMatrix;
15
16 // -----
17 // Compute the fragment's position in light space.
18 //
19 // FragPos is already computed as:
20 //
21 //     FragPos = vec3(model * vec4(aPos, 1.0));
22 //
23 // which is the vertex position in world coordinates. To determine whether
24 // this fragment is shadowed, the fragment shader must compare its light-space
25 // depth with the shadow map.
26 //
27 // We therefore transform the world-space position into the light's clip space:
28 //
29 //     FragPosLightSpace = lightSpaceMatrix * vec4(FragPos, 1.0);
30 //
31 // The resulting homogeneous coordinate will later be perspective-divided
32 // and remapped into [0,1] to index the shadow map.
33 // -----
34 void main(){
35     ...
36     FragPosLightSpace = lightSpaceMatrix * vec4(FragPos, 1.0);
37 }
```

We need to compute shader in the cube fragment shader program, we need to introduce several variables first, and define a function that is used to compute shadows:

```
1 // -----
2 // Light-space position passed from the vertex shader.
3 //
4 // This is the position of the current fragment transformed into the light's
5 // clip space:
6 //
7 //     FragPosLightSpace = lightSpaceMatrix * vec4(FragPos, 1.0);
8 //
9 // It is used to determine the fragment's (u, v, depth) inside the shadow map.
10 // -----
11 in vec4 FragPosLightSpace;
12
13 // -----
14 // The shadow map generated in Pass 1.
15 //
16 // This texture stores the depth of the closest surfaces from the light's
17 // perspective. In Pass 2 we compare each fragment's computed light-space
18 // depth against the shadow map to determine whether it is shadowed.
19 // -----
20 uniform sampler2D shadowMap;
21
```

```

22 // -----
23 // Declaration of the shadow calculation function.
24 // Computes a shadow factor in [0,1], where:
25 //     0 = fully lit
26 //     1 = fully in shadow
27 // -----
28 float ShadowCalculation(vec4 fragPosLS);

```

In the shadow computation function, we need to compare each fragment's depth with the global map to compute a shadow factor:

```

1 float ShadowCalculation(vec4 fragPosLS)
2 {
3     // -----
4     // (1) Perform perspective divide.
5     //
6     // The value passed in from the vertex shader is in homogeneous clip space.
7     // To convert it to normalized device coordinates (NDC):
8     //
9     //     proj = fragPosLS.xyz / fragPosLS.w
10    //
11    // NDC lies in [-1, 1] along each axis.
12    // -----
13    vec3 proj = fragPosLS.xyz / fragPosLS.w;
14
15    // -----
16    // (2) Transform from NDC [-1,1] to shadow map texture coordinates [0,1].
17    // -----
18    proj = proj * 0.5 + 0.5;
19
20    // -----
21    // (3) If the projected depth is outside the light's far plane,
22    //     treat the fragment as fully lit (no shadow).
23    //
24    // Values > 1.0 lie beyond the shadow map range.
25    // -----
26    if (proj.z > 1.0)
27        return 0.0;
28
29    // -----
30    // (4) Compute depth bias.
31    //
32    // Bias prevents "shadow acne", an artifact caused when numerical precision
33    // makes fragments incorrectly self-shadow.
34    //
35    // A slope-scaled bias is applied:
36    //
37    //     bias = k * (1 - dot(n, l))
38    //
39    // where:
40    // - n is the fragment normal
41    // - l is the light direction
42    //
43    // Flat surfaces get small bias; steep angles get larger bias.
44    // -----
45    float bias = max(
46        0.0007 * (1.0 - dot(normalize(Normal),
47                           normalize(-dirLights[0].direction))),
48        0.0003
49    );
50
51    // -----
52    // (5) Percentage Closer Filtering (PCF).
53    //

```

```

54 // Instead of a single depth comparison, PCF samples a 3x3 kernel of
55 // shadow map texels centered at proj.xy. This produces soft shadow edges.
56 //
57 // The logic:
58 //
59 //     if (fragmentDepth - bias > sampledDepth)
60 //         the sample is in shadow
61 //
62 // The final shadow factor is averaged over the kernel.
63 // -----
64 float shadow = 0.0;
65 vec2 texel = 1.0 / textureSize(shadowMap, 0);
66
67 for (int x = -1; x <= 1; x++)
68 for (int y = -1; y <= 1; y++)
69 {
70     float pcf = texture(shadowMap,
71                         proj.xy + vec2(x,y) * texel).r;
72
73     shadow += (proj.z - bias > pcf) ? 1.0 : 0.0;
74 }
75
76 // Normalize the result to [0, 1].
77 shadow /= 9.0;
78
79 return shadow;
80 }

```

Once we get the shadow factor, we can integrate it in the directional light computation function:

```

1 // -----
2 // Original directional light computation:
3 //
4 //     return ambient + diffuse + specular;
5 //
6 // To integrate shadows, we compute a shadow factor using the fragment's
7 // light-space position. The factor lies in [0,1].
8 //
9 //     shadow = 0 + fully lit
10 //     shadow = 1 + fully shadowed
11 //
12 // Ambient lighting is usually left unshadowed for artistic and visibility
13 // reasons. Only diffuse and specular terms are attenuated.
14 // -----
15
16 float shadow = ShadowCalculation(FragPosLightSpace);
17
18 // Apply shadow to the diffuse and specular contributions.
19 // Ambient term remains unchanged.
20 return ambient + (1.0 - shadow) * (diffuse + specular);

```

In the main rendering loop, we then just render the objects as usual, the only tricky thing is we need to restore our viewport:

```

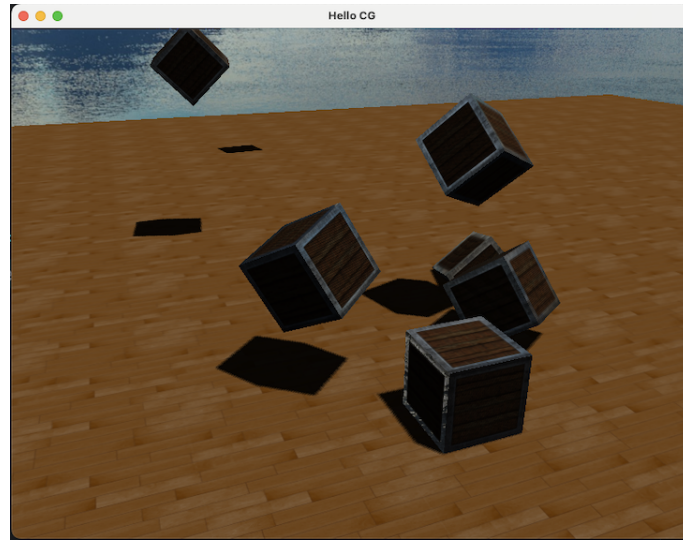
1 // -----
2 // PASS 2: MAIN RENDERING FROM THE CAMERA'S POINT OF VIEW
3 //
4 // At this stage, the shadow map generated in Pass 1 is complete.
5 // Now we render the scene normally from the camera, but for each fragment
6 // we also query the shadow map to decide whether it is lit or shadowed.
7 // -----
8

```

```

9 // -----
10 // (1) Restore the viewport to the window resolution.
11 //
12 // The depth pass used the shadow-map resolution (SHADOW_WIDTH x SHADOW_HEIGHT).
13 // Now we switch back to rendering into the default framebuffer (the window),
14 // so the viewport must match SCR_WIDTH and SCR_HEIGHT.
15 // -----
16 glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
17
18 // -----
19 // (2) Clear the default framebuffer.
20 //
21 // We clear both the color buffer (to the current clear color) and the depth
22 // buffer (so previous frame's depth does not interfere with the new frame).
23 // -----
24 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
25
26 cubeShader.use();
27 cubeShader.setUniform("lightSpaceMatrix", lightSpaceMatrix);
28 glActiveTexture(GL_TEXTURE3);
29 glBindTexture(GL_TEXTURE_2D, depthMap);
30 cubeShader.setUniform("shadowMap", 3);
31
32 glm::mat4 view = camera.GetViewMatrix();
33 glm::mat4 projection = camera.GetProjection((float)SCR_WIDTH / SCR_HEIGHT);
34
35 cubeShader.setUniform("view", view);
36 cubeShader.setUniform("projection", projection);
37 cubeShader.setUniform("viewPos", camera.Position);
38
39 glActiveTexture(GL_TEXTURE0);
40 glBindTexture(GL_TEXTURE_2D, cube_diffuse);
41
42 glActiveTexture(GL_TEXTURE1);
43 glBindTexture(GL_TEXTURE_2D, cube_specular);
44
45 for (unsigned int i = 0; i < 10; i++) {
46     glm::mat4 model = glm::mat4(1.0f);
47     model = glm::translate(model, cubePositions[i]);
48     float angle = 20.0f * i;
49     model = glm::rotate(model,
50                         glm::radians(angle),
51                         glm::vec3(1.0f, 0.3f, 0.5f));
52
53     cubeShader.setUniform("model", model);
54     cube.draw();
55 }
56 glActiveTexture(GL_TEXTURE0);
57 glBindTexture(GL_TEXTURE_2D, plane_diffuse);
58
59 glActiveTexture(GL_TEXTURE1);
60 glBindTexture(GL_TEXTURE_2D, plane_specular);
61
62 cubeShader.setUniform("model", glm::mat4(1.0f));
63 plane.draw();

```



7.5.3 Intro to Point Light Shadow Mapping

Omnidirectional shadow mapping works much like directional shadow mapping: we render a depth map from the light's point of view, then compare each fragment's depth with the stored value to determine whether it lies in shadow. The key difference is that a point light emits light in all directions, so a single 2D depth map is insufficient. Instead, we render the scene six times—one for each face of a cubemap—to capture depth information in every direction around the light. This depth cubemap is then sampled in the fragment shader using a direction vector from the light to the fragment to retrieve the closest depth value. The overall idea is similar to standard shadow mapping, but the cubemap-based depth pass makes omnidirectional shadows slightly more complex.

The function `createPointShadowMap` initializes the framebuffer and cubemap texture required for point-light shadow mapping. Unlike directional shadows, which store depth in a 2D texture, point lights must capture depth in all six directions, forming a depth cubemap. This function allocates a dedicated framebuffer object (FBO), initializes a 6-face cubemap where each face stores depth values, configures appropriate texture parameters to avoid seams or interpolation artifacts, and prepares the FBO to be used in the six-pass depth rendering stage. Importantly, the cubemap faces are not attached here; instead, each face will be attached during each of the six rendering passes corresponding to the six cube directions.

```

1 // -----
2 // Create a depth cubemap for point-light shadow mapping.
3 // Unlike directional light shadows (which use a 2D depth map), point-light
4 // shadows require sampling depth in all directions. This is implemented using
5 // a cubemap where each of the 6 faces stores depth information for one of
6 // the light's view directions.
7 // -----
8 void createPointShadowMap(GLuint &depthFBO, GLuint &depthCubemap,
9                           int SHADOW_SIZE)
10 {
11     // -----
12     // 1. Create the framebuffer that will hold the depth cubemap.
13     // -----
14     glGenFramebuffers(1, &depthFBO);
15
16     // -----
17     // 2. Create the cubemap texture. Each of the six faces needs a
18     //    depth-image allocation. No color buffer is used.
19     // -----
20     glGenTextures(1, &depthCubemap);
21     glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
22
23     for (int i = 0; i < 6; i++)
24     {
25         // Allocate an empty depth texture for each cubemap face.
26         // GL_DEPTH_COMPONENT as internal format stores depth only.
27         glTexImage2D(
28             GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, // face index
29             0, // mipmap level

```

```

30         GL_DEPTH_COMPONENT,           // internal depth format
31         SHADOW_SIZE, SHADOW_SIZE,     // resolution per face
32         0,                             // border (must be 0)
33         GL_DEPTH_COMPONENT,           // input format
34         GL_FLOAT,                     // data type
35         nullptr                        // no data yet (empty)
36     );
37 }
38
39 // -----
40 // 3. Texture parameters: no filtering and clamp-to-edge
41 // (required to avoid seams and interpolation artifacts).
42 // -----
43 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
44 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
45
46 // Clamp all three cube axes to avoid sampling beyond face boundaries.
47 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
48 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
49 glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
50
51 // -----
52 // 4. Bind the framebuffer. We intentionally do NOT attach a
53 // cubemap face here, because point-light shadow mapping
54 // requires 6 separate rendering passes. Each pass manually
55 // attaches the corresponding cube face using:
56 //
57 //     glFramebufferTexture2D(
58 //         GL_FRAMEBUFFER,
59 //         GL_DEPTH_ATTACHMENT,
60 //         GL_TEXTURE_CUBE_MAP_POSITIVE_X + face,
61 //         depthCubemap,
62 //         0
63 //     );
64 //
65 // This ensures each face receives the correct depth image.
66 // -----
67 glBindFramebuffer(GL_FRAMEBUFFER, depthFBO);
68
69 // No color buffer is required for depth-only shadow mapping.
70 glDrawBuffer(GL_NONE);
71 glReadBuffer(GL_NONE);
72
73 // Unbind the framebuffer so it can be used during the render loop.
74 glBindFramebuffer(GL_FRAMEBUFFER, 0);
75 }

```

Point-light shadow mapping requires rendering the scene from six different viewpoints—one for each face of the cubemap. The function `buildPointLightMatrices` computes these six view-projection matrices that correspond to the positive/negative X, Y, and Z directions of the cubemap. Each matrix is built by combining a 90° perspective projection with a `lookAt` transformation that orients the camera as if located at the point light’s position. Carefully chosen up-vectors eliminate seams and maintain consistent orientation across the cubemap faces. These matrices are later used in the point-light depth pass to render depth into each cubemap face.

```

1 // -----
2 // Build the six view-projection matrices used for rendering a depth cubemap
3 // for point-light shadow mapping.
4 //
5 // Each cubemap face has a different camera orientation (+X, +Y, +Z), but
6 // shares the same 90° perspective projection. The resulting matrices allow us
7 // to render the scene from the light’s point of view in all directions.
8 // -----
9 std::vector<glm::mat4> buildPointLightMatrices(glm::vec3 lightPos,
10                                              float nearPlane,

```

```

11         float farPlane)
12     {
13         // -----
14         // 1. Perspective projection for each cubemap face:
15         //    - 90° FOV so each face covers exactly one cube direction.
16         //    - Aspect ratio = 1.0 because each face is square.
17         // -----
18         glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f),
19                                                 1.0f,
20                                                 nearPlane,
21                                                 farPlane);
22
23         std::vector<glm::mat4> matrices;
24         matrices.reserve(6); // Prevent reallocations
25
26         // -----
27         // 2. Six view matrices | one for each cubemap direction.
28         //
29         //    The key part is choosing correct:
30         //    - look direction
31         //    - up-vector
32         //
33         //    The up-vector MUST be consistent with OpenGL's cubemap conventions.
34         //    Choosing the wrong up-vector will cause seams or rotated faces.
35         // -----
36
37         // +X direction
38         matrices.push_back(
39             shadowProj *
40             glm::lookAt(lightPos,
41                         lightPos + glm::vec3(1, 0, 0), // look toward +X
42                         glm::vec3(0, -1, 0))           // up-vector
43         );
44
45         // -X direction
46         matrices.push_back(
47             shadowProj *
48             glm::lookAt(lightPos,
49                         lightPos + glm::vec3(-1, 0, 0), // look toward -X
50                         glm::vec3(0, -1, 0))           // same up-vector as +X
51         );
52
53         // +Y direction
54         matrices.push_back(
55             shadowProj *
56             glm::lookAt(lightPos,
57                         lightPos + glm::vec3(0, 1, 0), // look toward +Y
58                         glm::vec3(0, 0, 1))           // up-vector for +Y
59         );
60
61         // -Y direction
62         matrices.push_back(
63             shadowProj *
64             glm::lookAt(lightPos,
65                         lightPos + glm::vec3(0, -1, 0), // look toward -Y
66                         glm::vec3(0, 0, -1))           // inverted up-vector
67         );
68
69         // +Z direction
70         matrices.push_back(
71             shadowProj *
72             glm::lookAt(lightPos,
73                         lightPos + glm::vec3(0, 0, 1), // look toward +Z
74                         glm::vec3(0, -1, 0))           // same up-vector as +X
75         );
76

```

```

77     // -Z direction
78     matrices.push_back(
79         shadowProj *
80         glm::lookAt(lightPos,
81                     lightPos + glm::vec3(0, 0, -1), // look toward -Z
82                     glm::vec3(0, -1, 0))
83     );
84
85     // Return all six VP matrices
86     return matrices;
87 }

```

Set up some global variables and function declarations on top of the main file:

```

1  void createPointShadowMap(GLuint &depthFBO, GLuint &depthCubemap, int SHADOW_SIZE);
2  std::vector<glm::mat4> buildPointLightMatrices(glm::vec3 lightPos, float nearPlane, float farPlane);
3
4  // =====
5  // POINT LIGHT SHADOW MAPPING GLOBALS
6  // =====
7  const unsigned int POINT_SHADOW_SIZE = 1024; // cubemap resolution
8  GLuint pointDepthFBO = 0; // depth-only FBO
9  GLuint pointDepthCubemap = 0; // depth cubemap
10 float pointNear = 1.0f;
11 float pointFar = 25.0f;

```

Right after GLAD initialization, call the `createPointShadowMap` function to set up buffers:

```

1  createPointShadowMap(pointDepthFBO, pointDepthCubemap, POINT_SHADOW_SIZE);

```

Now the buffer objects are ready, we need to create shader programs to compute our cube depth map. The point-light depth vertex shader transforms each vertex into the coordinate system of a specific cubemap face so that its depth can be stored in the corresponding layer of the depth cubemap. Because point lights illuminate in all directions, the shader is executed six times—once per cubemap face—each time using a different `shadowMatrix` provided by the CPU. The shader outputs the world-space fragment position to the fragment shader, enabling accurate depth calculations and normalization for the cubemap's depth values. This shader contains only the minimal transformations needed for shadow map rendering, making it efficient and tightly focused on producing a depth-only pass.

```

1  #version 410 core
2
3  // -----
4  // INPUT VERTEX ATTRIBUTE
5  // -----
6  // aPos holds the vertex position in model space.
7  // Only positions are needed for shadow-map generation; no normals or UVs.
8  layout (location = 0) in vec3 aPos;
9
10
11 // -----
12 // UNIFORMS
13 // -----
14
15 // Converts vertices from model space + world space.
16 // Needed because shadow depth is measured in world units
17 // (distance from FragPos to the light position).
18 uniform mat4 model;
19
20 // The view-projection matrix for the CURRENT cubemap face.
21 // The CPU updates this uniform SIX TIMES, once for each face:
22 //

```

```

23 // - +X, -X, +Y, -Y, +Z, -Z
24 //
25 // This avoids the need for a geometry shader or layered rendering.
26 uniform mat4 shadowMatrix;
27
28
29 // -----
30 // OUTPUT TO FRAGMENT SHADER
31 // -----
32
33 // World-space fragment position used in point-depth.frag to compute:
34 //
35 //     dist = length(FragPos - lightPos)
36 //
37 // This world-space distance becomes the stored depth value.
38 out vec3 FragPos;
39
40
41 void main()
42 {
43     // -----
44     // Convert vertex from model space → world space.
45     // This ensures the fragment shader computes the correct world distance.
46     // -----
47     FragPos = vec3(model * vec4(aPos, 1.0));
48
49     // -----
50     // Project the world-space position into the view-projection space of
51     // the current cubemap face (shadowMatrix).
52     //
53     // The result determines where the vertex lands on that face of the
54     // depth cubemap, and the rasterizer computes depth accordingly.
55     // -----
56     gl_Position = shadowMatrix * vec4(FragPos, 1.0);
57 }

```

The point-light depth fragment shader computes the distance from each fragment to the point light and writes this value into the depth cubemap. This depth information will later be sampled during the lighting pass to determine whether a fragment is shadowed. Since the cubemap stores normalized depth values, the shader converts the world-space distance into a 0–1 range using the same farPlane value used during matrix construction. This shader outputs only depth and therefore requires no color output or additional data, keeping the point-light depth pass efficient and lightweight.

```

1 #version 410 core
2
3 // -----
4 // INPUT FROM VERTEX SHADER
5 // -----
6 // World-space position of this fragment. Provided by point_depth_vertex.vert.
7 // We need world coordinates because shadow maps store distances in world units.
8 in vec3 FragPos;
9
10
11 // -----
12 // UNIFORMS
13 // -----
14
15 // The world-space coordinates of the point light.
16 // Depth is measured FROM this position TO the fragment.
17 uniform vec3 lightPos;
18
19 // The far clipping distance of the light's projection.
20 // This must match the farPlane used in buildPointLightMatrices().
21 // It is used to normalize depth into [0, 1] for cubemap storage.

```

```

22 uniform float farPlane;
23
24
25 void main()
26 {
27     // -----
28     // 1. Compute the world-space distance from the light to this fragment.
29     //
30     // This distance is the "true" depth for point-light shadows.
31     // Later, in the main lighting shader, we will compare:
32     //
33     // dist(fragment, light) vs. closestDepthFromShadowMap
34     //
35     // to determine whether the fragment is in shadow.
36     // -----
37     float dist = length(FragPos - lightPos);
38
39     // -----
40     // 2. Normalize the depth to [0, 1].
41     //
42     // The depth cubemap stores depth as a floating-point value where:
43     // 0.0 → fragment is at the light position
44     // 1.0 → fragment is at exactly farPlane units away
45     //
46     // Writing normalized depth ensures consistent sampling and comparison.
47     // -----
48     gl_FragDepth = dist / farPlane;
49 }

```

Now, in the main function, let's create a ShaderProgram object:

```

1 std::string pointDepthVert = std::string(SHADER_DIR) + "point_depth_vertex.vert";
2 std::string pointDepthFrag = std::string(SHADER_DIR) + "point_depth_fragment.frag";
3 ShaderProgram pointDepthShader(pointDepthVert, pointDepthFrag);

```

In the main rendering loop, we need to perform the depth rendering for point-light shadow mapping by populating the depth cubemap associated with a point light. Because a point light illuminates in all directions, the scene must be rendered six times—once for each face of the cubemap—using six different view-projection matrices. In each iteration, the corresponding cubemap face is attached to the framebuffer's depth attachment, the depth buffer is cleared, and the scene (plane and cubes) is rendered from the light's perspective in that direction. The resulting depth values encode the distance from the light to the closest surface in every direction, forming the basis for omnidirectional shadows in the main lighting pass.

```

1 // =====
2 // PASS 1B | POINT LIGHT SHADOW DEPTH PASS (6 renders into cubemap)
3 // =====
4
5 // Configure the viewport to match the shadow cubemap resolution.
6 // Each cubemap face is rendered at POINT_SHADOW_SIZE * POINT_SHADOW_SIZE.
7 glViewport(0, 0, POINT_SHADOW_SIZE, POINT_SHADOW_SIZE);
8
9 // Bind the depth-only framebuffer used for point-light shadow mapping.
10 glBindFramebuffer(GL_FRAMEBUFFER, pointDepthFBO);
11
12 // Activate the point-light depth shader program.
13 pointDepthShader.use();
14
15 // The light's world-space position and the farPlane are required by the
16 // depth fragment shader to compute and normalize distances.
17 pointDepthShader.setUniform("lightPos", cfg.pointLights[0].position);
18 pointDepthShader.setUniform("farPlane", pointFar);
19

```

```

20 // -----
21 // Render the scene six times, once for each cubemap face.
22 //
23 // For each face:
24 // 1. Attach the corresponding cubemap face as the depth attachment.
25 // 2. Clear the depth buffer.
26 // 3. Upload the appropriate shadowMatrix (view-projection for this face).
27 // 4. Draw the plane and cubes using the current matrix.
28 // -----
29 for (int face = 0; face < 6; face++)
30 {
31     // Attach the current cube face to the framebuffer's depth attachment.
32     // This ensures the depth values for this pass are written into the
33     // correct face of the cubemap.
34     glFramebufferTexture2D(
35         GL_FRAMEBUFFER,
36         GL_DEPTH_ATTACHMENT,
37         GL_TEXTURE_CUBE_MAP_POSITIVE_X + face,
38         pointDepthCubemap,
39         0
40     );
41
42     // Clear the depth buffer for this face before drawing.
43     glClear(GL_DEPTH_BUFFER_BIT);
44
45     // Set the view-projection matrix for the current cubemap face.
46     // pointShadowMatrices[face] comes from buildPointLightMatrices().
47     pointDepthShader.setUniform("shadowMatrix", pointShadowMatrices[face]);
48
49     // -----
50     // Draw the plane into the depth cubemap
51     // -----
52     {
53         glm::mat4 model = glm::mat4(1.0f); // identity model transform
54         pointDepthShader.setUniform("model", model);
55         plane.draw();
56     }
57
58     // -----
59     // Draw each cube into the depth cubemap
60     // -----
61     for (int i = 0; i < 10; i++)
62     {
63         glm::mat4 model = glm::mat4(1.0f);
64         model = glm::translate(model, cubePositions[i]);
65         model = glm::rotate(
66             model,
67             glm::radians(20.0f * i),
68             glm::vec3(1.0f, 0.3f, 0.5f)
69         );
70
71         pointDepthShader.setUniform("model", model);
72         cube.draw();
73     }
74 }
75
76 // After finishing all six faces, unbind the framebuffer so that subsequent
77 // rendering goes back to the default framebuffer (or another FBO).
78 glBindFramebuffer(GL_FRAMEBUFFER, 0);

```

After rendering the real objects, we can also visualize the point light similar to what we do in the previous chapter:

```

1  lightingCubeShader.use();
2
3  glm::mat4 model = glm::mat4(1.0f);
4  model = glm::translate(model, cfg.pointLights[0].position);
5  model = glm::scale(model, glm::vec3(0.3f));
6
7  lightingCubeShader.setUniform("model", model);
8  lightingCubeShader.setUniform("view", view);
9  lightingCubeShader.setUniform("projection", projection);
10
11 lightingCube.draw();

```

Now we have everything we need, one last thing is to compute the shadow in the cube fragment shader, first, we declare some uniform variables and function:

```

1  uniform samplerCube pointShadowMap;
2  uniform float farPlane;
3
4  float PointShadowCalculation(vec3 fragPos, vec3 lightPos);

```

The function `PointShadowCalculation` determines whether a fragment is shadowed with respect to a point light by comparing its distance from the light against the depth stored in the point-light shadow cubemap. During the depth pass, each cubemap face stores the normalized distance from the light to the closest surface in the corresponding direction. In this function, the shader reconstructs the actual distance from the stored normalized value and compares it to the fragment's true distance. If the fragment lies farther than the depth recorded in the cubemap (taking into account a small bias to prevent acne), then it is considered occluded. The function returns a shadow factor in the range[0,1], where 1.0 means fully shadowed and 0.0 means fully lit.

```

1  float PointShadowCalculation(vec3 fragPos, vec3 lightPos)
2  {
3      // -----
4      // 1. Compute the direction and distance from the light to the fragment.
5      //
6      //     This direction vector is used both to:
7      //         (a) sample the cubemap
8      //         (b) compute the true world-space distance for comparison
9      //
10     //     IMPORTANT:
11     //         The direction MUST match the one used during the depth pass:
12     //             lookAt(lightPos, lightPos + direction)
13     // -----
14     vec3 lightToFrag = fragPos - lightPos;
15     float dist = length(lightToFrag);
16
17
18     // -----
19     // 2. Sample the cubemap to obtain the normalized stored depth.
20     //
21     //     texture(pointShadowMap, direction).r returns a value in [0,1],
22     //     representing:
23     //
24     //         storedDepth = trueDepth / farPlane
25     //
26     //     Multiplying by farPlane reconstructs the true world-space distance.
27     // -----
28     float closest = texture(pointShadowMap, lightToFrag).r * farPlane;
29
30
31     // -----
32     // 3. Apply a small depth bias to help prevent self-shadowing artifacts
33     //     (known as "shadow acne"). This shifts the fragment slightly toward

```

```

34 // the light in depth comparisons.
35 //
36 // A typical bias for point-light shadows ranges from 0.01 to 0.05.
37 // -----
38 float bias = 0.03;
39
40
41 // -----
42 // 4. Shadow test:
43 //
44 // If the fragment's distance from the light is greater than the
45 // depth stored in the cubemap (minus bias), the fragment is behind
46 // another object and therefore in shadow.
47 //
48 // Returns:
49 // 1.0 → fully shadowed
50 // 0.0 → not shadowed
51 // -----
52 return (dist - bias > closest) ? 1.0 : 0.0;
53 }

```

Now, modify the point light computation function to introduce shadow calculation:

```

1 float shadow = PointShadowCalculation(fragPos, light.position);
2
3 // Ambient stays unshadowed
4 return ambient + (1.0 - shadow) * (diffuse + specular);
5

```

Finally, let's create a point light object:

```

1 cfg.pointLights.resize(1);

```

