A Beginner's Guide to Vulkan

Dylan Perks
Tyler Crandall
Vivian Jones

Table of Contents

Example

This chapter is specifically made for demonstration on how to add various elements to the book and this chapter SHOULD NOT be included in the final copy of the book.

Make sure to add figures for any images, because a figure table will be generated at the end of the book for easy look up.

```csharp
using System;

namespace Example
{
    public static class Program
    {
        // Entry Point
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Figure 1: Example C# Snippet

Also if for any reason you wish to make placement of figure between 2 paragraphs, there are two ways to accomplish it depending on your intent:

Enclose your figure inside
begin{frame}{} and
end{frame} to ensure it is strictly placed between 2 paragraphs.

Another way is assuming that you want figure to be placed on the side of a paragraph is to use
begin{wrapfigure}{r}{0.5
textwidth} and
end{wrapfigure} instead of figure begin/end.

```glsl
#version 450

#extension GL_ARB_separate_shader_objects : enable
#extension GL_ARB_shading_language_420pack : enable

layout (location = 0) in vec2 vsin_position;
layout (location = 1) in vec2 vsin_texCoord;
layout (location = 2) in vec4 vsin_color;

layout (binding = 0) uniform Projection
{
    mat4 projection;
};

layout (location = 0) out vec4 vsout_color;
layout (location = 1) out vec2 vsout_texCoord;

layout (constant_id = 0) const bool IsClipSpaceYInverted = true;
layout (constant_id = 1) const bool UseLegacyColorSpaceHandling =
    false;

out gl_PerVertex
{
    vec4 gl_Position;
};

vec3 SrgbToLinear(vec3 srgb)
{
    return srgb * (srgb * (srgb * 0.305306011 + 0.682171111) +
        0.012522878);
}

void main()
{
    gl_Position = projection * vec4(vsin_position, 0, 1);
    vsout_color = vsin_color;
    if (!UseLegacyColorSpaceHandling)
    {
        vsout_color.rgb = SrgbToLinear(vsin_color.rgb);
    }
    vsout_texCoord = vsin_texCoord;
    if (IsClipSpaceYInverted)
    {
        gl_Position.y = -gl_Position.y;
    }
}
```

Figure 2: Example GLSL Snippet

References

Also for references.bib, look into here for more information on how to add references for book, article, and other information to be used in this book.

Comment Bubbles

The quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog.

The quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog. the quick brown fox jumps right over the lazy dog.

Chapter 1

Introduction

Special thanks to Manuchi on Pixabay for the image used on the front cover! Used under the Pixabay License.

Dylan Perks, Tyler Crandall, and Vivian Jones wrote this book under the Ultz umbrella; with the generous help of an assortment of passionate community contributors.

This book will use C++, but it will be written as loosely as possible to allow readers looking to use Vulkan in other languages to rewrite the snippets without significant difficulty. We have excluded typical C++ boilerplate such as headers, which you must provide yourself. The full C++ source is available in our GitHub repository. We recommend having an intermediate understanding of how C++ works before continuing.

If you're reading this in one of the finished copies we distribute, thank you so much for supporting us! If you're just reading this on GitHub, then please consider supporting the authors of this book by buying an official copy. We only charge a minor price - cheaper than most other Vulkan books available, in fact.

- GitHub: https://github.com/Ultz/VulkanBook
- Amazon:

1 Overview

Over the past few decades, the computing industry has seen lots of graphics technology rise, thrive, and decline to make way for better graphics technology. The weird & wonderful world of graphic development is always changing, and it's hard to keep up. The best example of this has been the recent debut of Vulkan.

OpenGL has been the leading graphics API for nearly 30 years now, however along the way there seems to be a common consensus among developers: OpenGL doesn't give them enough control. Now, that's not to say that OpenGL hasn't seen its fair share of changes to accommodate developers, because it has changed a lot since its original inception in 1992. In 2008, OpenGL 3 released, bringing with it a whole new API for making high performance and effective applications using hardware graphics and for a while this was enough.

Then in 2013, everything changed when AMD released their Mantle API to the world, popularizing a whole other level of control. Mantle popularised the verbose style of APIs we're used to see pop up these days, such as Microsoft's Direct3D 12, Sony's Gnm, Nintendo/NVIDIA's NVN, Apple's Metal, and Khronos' Vulkan, to name a few.

Vulkan is the spiritual successor to Mantle, which struggled to gain traction, being primarily only available on AMD hardware. Vulkan is an open standard and is open for implementation on any platform by any hardware manufacturer and, while some companies prefer to stick to their own exclusive APIs, Vulkan is pretty universally supported.

This book aims to teach you about Vulkan and the fundamental concepts behind it. We wrote this book because there wasn't a good way to grasp Vulkan without being well-versed in one of the higher-level APIs such as OpenGL, which isn't very productive if you have to learn another API just to learn Vulkan, and the only other alternative was paying a hefty premium for the few books out there that are geared towards beginners.

We hope that anyone can pick up this book and delve into the world of Vulkan graphics development from little prior knowledge of graphics. Having struggled with grasping Vulkan (even with a fair amount of prior experience with OpenGL between us), we hope that we can save other aspiring Vulkan developers from going down the rocky path we went down in learning Vulkan with only a few good learning resources.

2 Rasterization

With Vulkan, everything is in 3D space. However, you may notice that your screen is a 2D panel. An enormous part of Vulkan's job is transforming the 3D coordinates you give it into a 2D frame. There are many ways to do this, but the most common method (and the method that Vulkan primarily uses) is called rasterization. This is where Vulkan takes your 3D coordinates, assembles shapes (usually triangles) according to your vertex data, places them onto a 2D grid of pixels (i.e. your screen), colouring the pixels according to your configuration. .

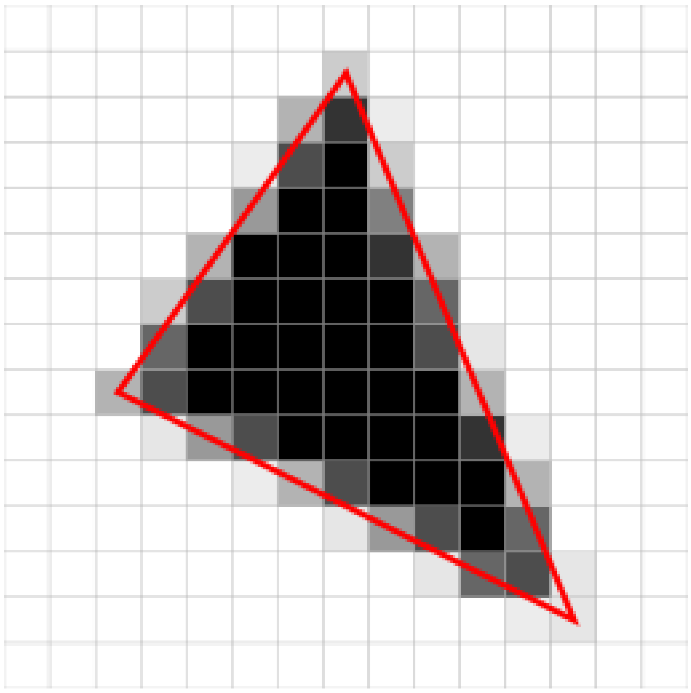


Figure 1.1: A picture showing how the edges of a triangle are rasterized onto a grid of pixels

The reason this is done on a dedicated graphics card instead of on your CPU in most cases is because instead of having a smaller number of cores with a high clock count, GPUs have a larger number of cores with a lower clock count, making it easier to do lots of small operations in parallel. This is useful for rasterization, which is an algorithm which is highly optimized for running in parallel and, as a result, is capable of rasterizing thousands of frames per second on higher-end graphics cards.

3 About Vulkan

Vulkan is an Application Programming Interface (API) specification developed and maintained by the Khronos Group, a consortium of lots of stakeholders in graphics development such as NVIDIA, AMD, Intel, and Arm; as well as other big names in the industry such as Epic Games and Valve, just to name a few. There are over 100 member organisations in the Khronos Group, and they collectively contribute to build open standards and specifications such as Vulkan.

This is important to note, as Vulkan by itself is not an API - it's the specification and the driver that together provide the standardized interface to the GPU. The Vulkan specification only specifies the expected behaviour for operations you request and the input that you provide to it. This means it's up to the driver vendor to choose how to implement the specified behaviour, which can lead to weird quirks between different implementations of Vulkan. You probably won't come across any of those quirks within this book, as we've done our best not to encourage usage of any unspecified behaviour ("unspecified behaviour" being behaviour of a function or operation isn't defined in the specification).

Usually, Vulkan is implemented by the GPU hardware manufacturers themselves (apart from a few exceptions like Google's swiftshader, which runs on CPUs), so each graphics card you buy only supports specific versions of Vulkan - the ones that the graphics card's Vulkan drivers are specifically built for. Therefore, it's recommended to update your graphics drivers if you are experiencing bugs or undefined behaviour.

Most operating systems allow GPU vendors to provide drivers for Vulkan, however there are some cases (such as on macOS) where the GPU manufacturers can't provide Vulkan drivers. With macOS, it's because Apple have their own Metal API, which is built specifically for Apple iPhones and Apple Macs. Luckily, a group called The Brenwill Workshop have developed the open-source (thanks to Valve) Vulkan emulator called MoltenVK which is used to implement Vulkan using Apple's Metal API as a translation layer.

The Khronos Group publicly hosts all specification documents for all Vulkan versions, which you can find on their GitHub repository linked at the bottom of this page. If you're interested in grabbing a copy of the Vulkan specification for reading alongside this book, the version we're using is Vulkan 1.2 which will teach you more about the specifics of Vulkan.

- Vulkan's GitHub: https://github.com/KhronosGroup/Vulkan-Docs

4 Vulkan's Extension Mechanism

One thing Khronos APIs are brilliant at doing is allowing Vulkan implementations to extend the normal functionality of Vulkan and add new techniques & features to the Vulkan API independently of the rest of the API and other implementations. For example, when NVIDIA's RTX platform launched it was the first platform to feature real-time ray-tracing en masse alongside typical graphics workloads like Vulkan. Because NVIDIA was the first to allow raytracing with Vulkan, they had to create a specially made extension called VK_NV_ray_tracing to add this missing functionality.

If lots of Vulkan drivers support an extension (as anyone can implement any Vulkan extension provided it conforms with the original author's intentions), the vendors involved will often work together to make a multi-vendor extension, which are prefixed with EXT (for generic multi-vendor) or KHR (for extensions by the Khronos Group). If a KHR extension sees widespread adoption, Khronos will often promote an extension into the core specification as part of a new Vulkan version, allowing it to be used anywhere where that specific version of Vulkan is supported. An example of this would be VK_KHR_get_physical_device_properties2 being promoted and included as part of Vulkan 1.1.

If the hardware and graphics driver supports the extra features defined in extensions, then the developers can query and use those extensions for their projects. All graphics card only supports a specific set of extensions, and they must be queried and enabled explicitly before accessing them. You can write it this way as an example:

```
if (VK_KHR_extension_name)
{
    // use the new cool features!
}
else
{
    // do things the old way...
}
```

Obviously it's not that simple to check if it supports an extension, but keep that example in mind. We will discuss further approach and practices that you can use to query and enable various features that your graphic card driver may provide. The conclusion to be made here is that the Vulkan standard is flexible in a way that drivers can extend Vulkan with custom functionality through "extensions", which you can then query and enable to use that functionality.

Chapter 2

Mathematics Principles

For almost any non-trivial graphical program, geometry will need to be moved across the screen. The simplest way to do that would be to update the geometry's vertices to the new position, but constantly sending new data to the graphics card is very slow, and not feasible for a large-scale program.

In addition, Normalized Device Coordinates are too abstract for most graphical work. For example, drawing an image at the proper size is nearly impossible when all you have are NDC. Wouldn't it be easier to use a different coordinates system?

The answer to both problems comes in the form of linear algebra. This chapter will be a *brief* introduction to the concept, and how it relates to graphics programming. If you find the topic interesting and want to know more, it is heavily recommended that you continue studying outside of this book. Several potentially-useful things (such as quaternions) are omitted here for brevity.

With that said, let's begin!

1 OpenGL Mathematics Library (GLM)

This tutorial is provided in C++, a language that does not include built-in linear algebra. Instead, we will use an external library: OpenGL Mathematics Library, or GLM for short. Despite the name, it will work for Vulkan just fine, as the mathematics required are the same. Download from here:

https://glm.g-truc.net/0.9.8/index.html

At the time of writing, version 0.9.9.8 is the current stable release. If you encounter problems, be certain that you use the same version as we do.

GLM is a header-only library, so no compiling or linking are necessary. Just download and copy the header folder into your includes folder.

To ensure that it works, try running the following code: .

```cpp
#include <cstdio>
#include "glm/glm.hpp"

int main() {
 glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f);
 glm::vec4 vec2(0.0f, 1.0f, 0.0f, 1.0f);

 vec += vec2;

 std::printf("{%f, %f, %f, %f}", vec.x, vec.y, vec.z, vec.w);
}
```

Figure 2.1: GLM test snippet

If GLM has been installed correctly, this code should print "1.0, 1.0, 0.0, 2.0".

2 Introduction to Vectors

In mathematics, a vector is a list of elements. A single element on its own is known as a scalar. A vector is basically a one-dimensional array of elements.

With GLM, you can create a vector as follows: .

```cpp
vec4 vec(0.0f, 0.0f, 0.0f, 0.0f);
```

Figure 2.2: GLM test snippet

Vectors can be of any size, but in graphics programming, only vectors of sizes 2 to 4 are commonly used, so those are the only ones included in GLM.

Mathematic notation for vectors is as follows:

$$\vec{v} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

In order, a vector's elements can be referred to as: X, Y, Z, and W. X, Y, and Z are positions in 3D space. W is different; it is known as the homogeneous vertex coordinate. It isn't important for now, but will be discussed later.

So, what can you do with a vector? Let's start with basic vector arithmetic.

2.1 Scalar operations

Scalar operations refer to any operation that uses both a scalar (a single element) and a vector (multiple elements). Simply put, the scalar is applied to each element of the vector separately. For example:

$$\begin{pmatrix}2\\1\\0\end{pmatrix} + 3 = \begin{pmatrix}2+3\\1+3\\0+3\end{pmatrix} = \begin{pmatrix}5\\4\\3\end{pmatrix}$$

It works exactly the same for subtraction, multiplication, and division.

In GLM: .

```cpp
#include "glm/glm"

int main() {
    glm::vec4 vec(1.0f, 0.0f, 2.0f);
    glm::vec4 vec2 = vec + 3;
}
```

Figure 2.3: Scalar operations

2.2 Vector operations

Vector operations are any operation that uses two vectors. The operation is applied to each element of the vector (X with X, Y with Y, etc.).

$$\begin{pmatrix}2\\1\\0\end{pmatrix} + \begin{pmatrix}9\\8\\7\end{pmatrix} = \begin{pmatrix}2+9\\1+8\\0+7\end{pmatrix} = \begin{pmatrix}11\\9\\7\end{pmatrix}$$

Vector addition and subtraction work the same way, but multiplication (and division, which is a form of multiplication) are different; see 3.3: Vector Products. Vector operations are only allowed when both vectors are of the same size. For example, you can't add a vector with a size of 2 to a vector with a size of 4.

In GLM: .

```cpp
#include "glm/glm.hpp"

int main() {
    glm::vec4 vec1(1.0f, 1.0f, 1.0f);
    glm::vec4 vec2(2.5f, 0.4f, 1.5f);

    glm::vec4 vec3 = vec1 + vec2;
}
```

Figure 2.4: Vector operations

2.3 Length of Vectors

The length of a vector is something very important to know for several different reasons. The length of a vector is notated as follows:

$$||\vec{v}||$$

Luckily, it's not hard to figure out. You can think of the X and Y coordinates as being the height and width of a triangle, like so: .

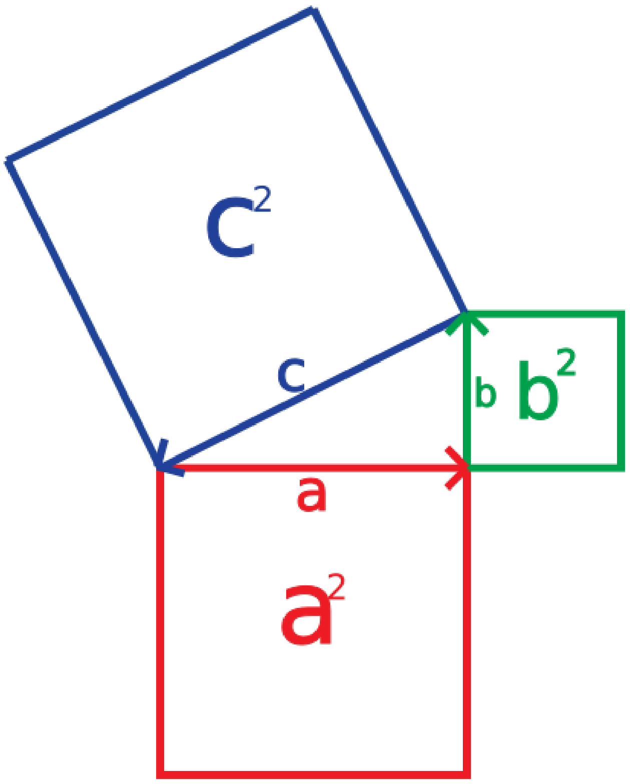


Figure 2.5: Pythagorean Theorem

Using this example, the length would be the hypotenuse of the triangle. To find this, you can use what's known as the Pythagorean Theorem. This is a very simple operation, just do the following:

1. Square all of the elements.
2. Add all the elements together.
3. Square root the result.

On paper, it looks like this:

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

In GLM: .

```cpp
#include "glm/glm.hpp"

int main() {
    glm::vec4 vec(2.0f, 1.0f, 3.0f);

    float length = glm::length(vec);
}
```

Figure 2.6: Vector length

2.4 Unit vectors

A unit vector is a vector whose length is 1. Every vector has a non-zero unit vector, which can be found by dividing a vector by its length:

$$\frac{\vec{v}}{||\vec{v}||}$$

The unit vector is simple to find, but it is used in many different formulas. One major one will be discussed in the next section.

In GLM: .

```cpp
#include "glm/glm.hpp"

int main() {
    glm::vec4 vec(5.0, 2.6, 1.3);
    glm::vec4 vec2 = glm::normalize(vec);
}
```

Figure 2.7: Normalizing a vector

3 Vector products

Vector addition and subtraction are both very simple, but what of multiplication? It's not quite as easy. For a number of mathematical reasons that are beyond the scope of this book, multiplying component-wise isn't correct.

Instead, there are two different types of vector multiplication: *dot products*, and *cross products.*

3.1 Dot product

The dot product (defined with a center circle, ·) is another simple operation: multiply the matching elements (X with X, Y with Y...), and then add all the products together. On paper, it looks like this:

$$\begin{pmatrix}1\\2\\3\end{pmatrix}\cdot\begin{pmatrix}4\\5\\6\end{pmatrix} = (1 \ * \ 4) \ + \ (2 \ * \ 5) \ + \ (3 \ * \ 6) \ = 32$$

On its own, this answer might seem random, but there's an important property of the result:

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

Simply put, the result will always be the length of vector V, multiplied by the length of vector K, multiplied by the cosine of the angle between them.

This isn't especially useful on its own. However, if you use unit vectors, the results are much more useful. Since both lengths are equal to 1, you're left with just the cosine of the angle between them. If you put the result into an arccos function, you get the angle of the vectors between them.

```cpp
#include <cmath>
#include "glm/glm.hpp"

int main() {
    glm::vec4 vec1(1.0f, 2.5f, 1.7f);
    glm::vec4 vec2(2.0f, 0.6f, 9.0f);

    vec1 = glm::normalize(vec1);
    vec2 = glm::normalize(vec2);

    double dotproduct = glm::dot(vec1, vec2);

    double angle = std::acos(dotproduct);
}
```

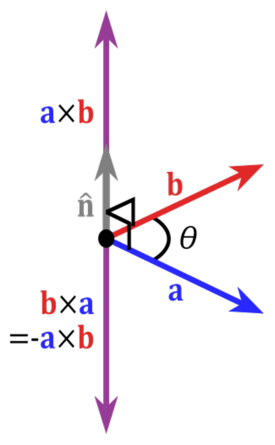
Figure 2.8: Dot product

In GLM, you can get the dot product and angle as follows: .

3.2 Cross product

The cross product (defined with the multiplication sign, $\times$) isn't quite as commonly-used as the dot product, but it's still important to know. Unlike other vector operations, where any vector sizes are permitted so long as they are the same, cross products are only defined for vectors with three elements.

The cross product (defined with the multiplication sign, $\times$) isn't quite as commonly-used as the dot product, but it's still important to know. Unlike other vector operations, where any vector sizes are permitted so long as they are the same, cross products are only defined for vectors with three elements. .



Original image taken from Wikipedia, and licensed under the public domain.

Figure 2.9: Cross Product.

The cross product formula is as follows:

$$\begin{pmatrix} A_x \\ A_y \\ A_z \end{pmatrix} \times \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} = \begin{pmatrix} A_y \cdot B_z - A_z \cdot B_y \\ A_z \cdot B_x - A_x \cdot B_z \\ A_x \cdot B_y - A_y \cdot B_x \end{pmatrix} \tag{2.1}$$

The cross-product formula is the first operation discussed that is not order-independent; BŒA is not the same as AŒB.

If you don't understand the cross product formula, don't worry, GLM provides a function for it: .

```cpp
#include "glm/glm.hpp"

int main() {
    glm::vec4 vecX(3, -3, 1);
    glm::vec4 vecY(4, 9, 2);

    glm::vec4 vecZ = glm::cross(vecX, vecY);
}
```

Figure 2.10: Cross product example

4 Intro to Matrices

In mathematics, a matrix (plural: matrices) is a grid of elements. You can think of a matrix as being like a 2D array. For example:

$$\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$$

Just like vectors, matrices can be of any size, but only sizes from 1 to 4 are used. Matrices are named based on their dimensions; a matrix that is 3 elements wide and 2 elements tall would be a 3x2 matrix. In computing, when the height and width are the same, it is common to abbreviate it to just the one number; a 4x4 matrix is just called a mat4 in GLM.

You can create a matrix in GLM as follows: .

Matrices are very important for graphics programming; matrices are how you can move around your polygons without having to replace their vertices. Scaling, translation, and rotation are all accomplished using matrices.

```cpp
#include "glm/glm.hpp"

int main() {
    glm::mat4 matrix(1.0f);
}
```

Figure 2.11: Creating a matrix

4.1 Row Major vs Column Major

Before you can learn anything about matrices, you need to understand matrix ordering.

There is some debate as to how matrices should be laid out in memory. Matrices are made up of rows (horizontal) and columns (vertical). There are two different ways a matrix can be laid out: row-major, and column-major.

In row-major, the index progresses from left to right, dropping down to the next row after reaching the end. In column-major matrices, the index progresses from top to bottom, moving right to the next column after reaching the end. To demonstrate: .

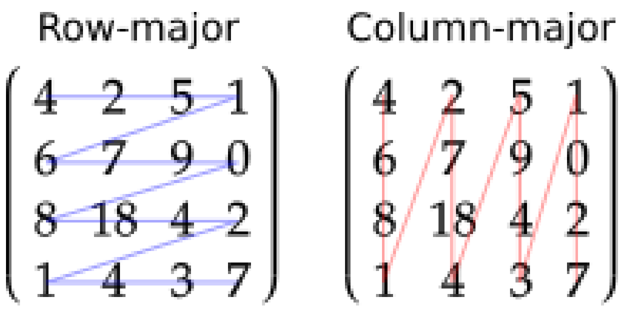


Figure 2.12: Row-major vs column-major matrix ordering

In this example, if you looked at the fifth element, you'd see 6 if you were using row-major, or 2 if you're using column major.

GLM uses row-major for all operations, so this book will as well. *If you run into unexpected behavior while using matrices, it could be because you're using column-major instead.*

Luckily, converting between row-major and column-major (and vice-versa) is very simple. The process is known as *transposing*. If you get incorrect results from a matrix operation, transposing should be your first debugging step. You can transpose a matrix in GLM as follows: .

```cpp
#include "glm/glm.hpp"

int main() {
    glm::mat4 matrix(1.0f);

    matrix = glm::transpose(matrix);
}
```

Figure 2.13: Creating a matrix

4.2 Matrix addition and subtraction

Like vectors, matrix addition and subtraction are both done component-wise:

$$\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + \begin{pmatrix} 8 & 7 \\ 4 & 9 \end{pmatrix} = \begin{pmatrix} 2+8 & 1+8 \\ 0+4 & 3+9 \end{pmatrix} = \begin{pmatrix} 10 & 9 \\ 4 & 12 \end{pmatrix}$$

Scalar operations also still work the same way:

$$\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} + 7 = \begin{pmatrix} 2+7 & 1+7 \\ 0+7 & 3+7 \end{pmatrix} = \begin{pmatrix} 9 & 8 \\ 7 & 10 \end{pmatrix}$$

4.3 Matrix multiplication

Matrices can be multiplied by a scalar the same way vectors can:

$$\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix} \times 3 = \begin{pmatrix} 2\times 3 & 1\times 3 \\ 0\times 3 & 3\times 3 \end{pmatrix} = \begin{pmatrix} 6 & 3 \\ 0 & 9 \end{pmatrix}$$

Multiplication with a vector or another matrix is where matrices become more difficult, but also more useful. Like with vectors, component-wise matrix multiplication isn't possible. Unlike vectors, the dot product and cross product are not defined for matrices.

Chapter 3

Graphics Principles

Chapter 4

Initial Setup

Chapter 5

Shaders

Chapter 6

Pipelines

Chapter 7

Render Passes

Chapter 8

Drawing

Chapter 9

Project Guidance

Chapter 10

Textures

Chapter 11

Advanced Textures

Chapter 12

Mapping

Chapter 13

Instancing

Chapter 14

Tessellation