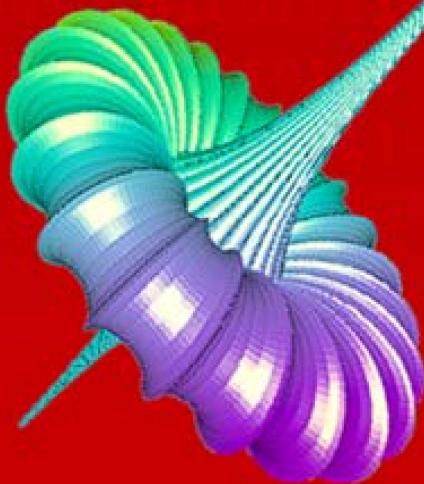




# Practical GPU Graphics with `wgpu` and Rust

Creating Advanced Graphics on Native  
Devices and the Web in Rust Using `wgpu` –  
the Next-Generation Graphics API



**Jack Xu, PhD**

Buy eBook at <https://drxudotnet.com>

# Practical GPU Graphics with wgpu and Rust

YouTube Channel: Practical Programming with Dr. Xu



# Practical GPU Graphics with wgpu and Rust

Creating Advanced GPU Graphics on Native  
Devices and the Web in Rust Using wgpu – the  
Next-Generation Graphics API

*Jack Xu, PhD*



UniCAD Publishing

Practical GPU Graphics with wgpu and Rust  
Copyright © 2021 by Jack Xu, PhD  
Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1UC

Editor: Anna Xu

All rights reserved. No part of the contents of this book and corresponding example source code may be reproduced or transmitted in any form or by any means without the written permission of the author.

The author has made every effort in the preparation of this book to ensure the accuracy of the information; however, this book is sold without warranty, either express or implied. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained in the book.

Published by UniCAD Publishing.  
New York, USA  
ISBN: 979-8404949377

Publisher's Cataloging-in-Publication Data

Xu, Jack  
Practical GPU Graphics with wgpu and Rust: Creating Advanced GPU Graphics on Native Devices and the Web in Rust Using wgpu – the Next-Generation Graphics API / Jack Xu  
– 1st ed.  
p.cm.  
ISBN 979-8404949377

1. wgpu. 2. Rust. 3. WebGPU. 4. Shader. 5. GPU Acceleration. 6. Graphics on Web. 7. SPIR-V. 8. WebGL. 9. WebAssembly. 10. WASM. 11. Graphics Rendering. 12. GPU Computing.  
I. Title. II. Title. III Title: Practical GPU Graphics with wgpu and Rust

# Contents

<b>Contents .....</b>	<b>v</b>
<b>Introduction .....</b>	<b>1</b>
Overview .....	1
What this Book Includes.....	3
Is this Book for You?.....	3
What Do You Need to Use this Book?.....	4
How this Book Is Organized .....	4
Using Code Examples .....	6
Customer Support.....	6
<b>1 Setting Up Development Tools.....</b>	<b>7</b>
1.1 Hardware Requirements.....	7
1.2 Install C++ Build Tools.....	10
1.3 Install Rust .....	10
1.4 Install Visual Studio Code .....	11
1.5 Run a Rust Application .....	11
1.6 Configuration .....	12
1.7 Creating a Window .....	15
1.8 Rust Basics .....	15
1.8.1 Data Types .....	17
1.8.2 Functions.....	17

1.8.3	Control Flows.....	18
1.8.4	Arrays, Vectors, and Slices .....	21
1.8.5	Structs .....	22
1.8.6	Enums .....	23
1.8.7	Generics .....	24
<b>2</b>	<b>wgpu Basics .....</b>	<b>25</b>
2.1	First wgpu Example .....	25
2.1.1	WGSL Shaders.....	25
2.1.2	Common Rust Code.....	26
2.1.3	Rust Main Function.....	28
2.2	wgpu API .....	30
2.2.1	wgpu Backend.....	30
2.2.2	Surface .....	31
2.2.3	Load Shaders.....	32
2.2.4	Rendering Pipeline .....	32
2.2.5	Rendering Output.....	34
2.3	Shader Program.....	35
2.3.1	Why Use WGSL Shaders? .....	36
2.3.2	Writing Shader Code.....	36
2.4	Triangle with Different Vertex Colors.....	38
2.4.1	Shader Code .....	39
2.4.2	Rust Code .....	39
<b>3</b>	<b>wgpu Primitives .....</b>	<b>41</b>
3.1	Creating Points .....	42
3.1.1	Rust Code.....	42
3.1.2	Shader Code .....	43
3.1.3	Run Application .....	44
3.2	Creating Lines .....	45
3.3	Creating Triangles .....	47
3.3.1	Rust Code.....	47
3.3.2	Shader Code .....	48
3.3.3	Run Application .....	49
<b>4</b>	<b>GPU Buffers.....</b>	<b>51</b>
4.1	GPU Buffer .....	51
4.2	Creating a Colored Triangle.....	52
4.2.1	Rust Code.....	53

4.2.2	Shader Code .....	60
4.2.3	Run Application .....	60
4.3	Creating a Colored Square .....	61
4.3.1	Rust Code.....	62
4.3.2	Run Application .....	62
4.4	Creating a Square with an Index Buffer.....	63
4.4.1	Rust Code.....	63
4.4.2	Run Application .....	65
<b>5</b>	<b>3D Transformations .....</b>	<b>67</b>
5.1	Basics of 3D Matrices and Transformations .....	67
5.1.1	Introducing <i>cgmath</i> .....	67
5.1.2	3D Vector and Matrix Operations.....	68
5.1.3	Scaling.....	69
5.1.4	Translation .....	71
5.1.5	Rotation.....	72
5.1.6	Combining Transformations .....	74
5.2	Projections and Viewing .....	75
5.2.1	Transforming Coordinates .....	75
5.2.2	Viewing Transform .....	76
5.2.3	Perspective Projection.....	79
5.2.4	Orthographic Projection.....	83
5.3	Transformations in <i>wgpu</i> .....	86
<b>6</b>	<b>3D Shapes and Camera.....</b>	<b>87</b>
6.1	Uniform Buffers and Bind Groups.....	87
6.2	Creating a 3D Line .....	89
6.2.1	Common Code .....	89
6.2.2	Rust Code.....	92
6.2.3	Shader Program.....	98
6.2.4	Run Application .....	98
6.3	Creating a Cube with Distinct Face Colors.....	100
6.3.1	Create Vertex Data.....	100
6.3.2	Rust Code.....	102
6.3.3	Shader Program.....	109
6.3.4	Run Application .....	110
6.4	Creating a Cube with Distinct Vertex Colors .....	111
6.4.1	Create Vertex Data.....	111
6.4.2	Rust Code.....	111

6.4.3	Run Application .....	118
6.5	Rotating Objects .....	119
6.5.1	Rust Code .....	119
6.5.2	Run Application .....	120
6.6	Camera Controls.....	121
6.6.1	Camera Code.....	121
6.6.2	Rust Code .....	123
6.6.3	Run Application .....	130
<b>7</b>	<b>3D Wireframe Shapes .....</b>	<b>131</b>
7.1	Common Code .....	131
7.2	Cube Wireframe .....	136
7.2.1	Rust File .....	136
7.2.2	Run Application .....	137
7.3	Sphere Wireframe .....	138
7.3.1	Spherical Coordinate System .....	138
7.3.2	Rust Code .....	140
7.3.3	Run Application .....	141
7.4	Cylinder Wireframe .....	141
7.4.1	Cylindrical Coordinate System .....	142
7.4.2	Rust Code .....	143
7.4.3	Run Application .....	145
7.5	Cone Wireframe .....	146
7.5.1	Rust Code .....	147
7.5.2	Run Application .....	148
7.6	Torus Wireframe .....	149
7.6.1	Rust Code .....	150
7.6.2	Run Application .....	151
<b>8</b>	<b>Lighting in WGPU .....</b>	<b>153</b>
8.1	Light Components .....	153
8.2	Normal Vectors .....	154
8.2.1	Surface Normal of a Cube .....	155
8.2.2	Surface Normal of a Sphere .....	155
8.2.3	Surface Normal of a Cylinder .....	155
8.2.4	Surface Normal of a Polyhedral Surface .....	155
8.3	Lighting Calculation.....	156
8.3.1	Diffuse Light .....	156
8.3.2	Specular Light .....	157

8.4	Lighting in Shaders .....	158
8.4.1	Transform Normals .....	158
8.4.2	Shader with Lighting.....	159
8.5	Common Code .....	161
8.6	Cube with Lighting .....	170
8.6.1	Rust Code.....	170
8.6.2	Run Application .....	171
8.7	Sphere with Lighting.....	171
8.7.1	Vertex and Normal Data .....	172
8.7.2	Rust Code.....	173
8.7.3	Run Application .....	174
8.8	Cylinder with Lighting.....	174
8.8.1	Vertex and Normal Data .....	175
8.8.2	Rust Code.....	177
8.8.3	Run Application .....	177
8.9	Cone with Lighting .....	178
8.9.1	Vertex and Normal Data .....	179
8.9.2	Rust Code.....	180
8.9.3	Run Application .....	181
8.10	Torus with Lighting .....	181
8.10.1	Vertex and Normal Data .....	182
8.10.2	Rust Code.....	183
8.10.3	Run Application .....	183
<b>9</b>	<b>Colormaps and 3D Surfaces.....</b>	<b>185</b>
9.1	Color Models.....	185
9.2	Colormaps .....	186
9.2.1	Colormap Data .....	186
9.2.2	Color Interpolation .....	187
9.3	Shaders with Lighting and Vertex Color .....	188
9.4	Simple 3D Surfaces.....	190
9.4.1	Position, Normal, and Color Data .....	191
9.4.2	Common Code .....	193
9.4.3	Sinc Surface .....	201
9.4.4	Peaks Surface .....	203
9.5	Parametric 3D Surfaces.....	205
9.5.1	Vertex and Normal Data .....	205
9.5.2	Klein Bottle .....	206
9.5.3	Wellenkugel Surface .....	209

9.5.4	Seashell Surface .....	210
9.5.5	Sievert-Enneper Surface .....	212
9.5.6	Breather Surface.....	213
<b>10</b>	<b>Textures.....</b>	<b>217</b>
10.1	Texture Coordinates .....	217
10.2	Texture Mapping in WGPU .....	218
10.3	Shaders with Texture.....	222
10.4	Common Code .....	223
10.5	Simple 3D Shapes .....	232
10.5.1	Cube with Texture.....	232
10.5.2	Sphere with Texture .....	235
10.5.3	Cylinder with Texture .....	237
10.6	Simple 3D Surfaces.....	241
10.6.1	Sinc Surface with Texture.....	243
10.6.2	Peaks Surface with Texture .....	245
10.7	Parametric 3D Surfaces.....	246
10.7.1	Klein Bottle with Texture .....	248
10.7.2	Wellenkugel Surface with Texture .....	250
10.8	Multiple Textures .....	252
10.8.1	Texture Coordinates.....	252
10.8.2	Rust Code.....	254
10.8.3	Run Application .....	254
<b>11</b>	<b>3D Surface Charts.....</b>	<b>257</b>
11.1	Wireframe as Texture.....	258
11.1.1	Square Textures .....	258
11.1.2	Texture Coordinates.....	259
11.2	Shaders for 3D Charts .....	259
11.3	Common Code .....	261
11.4	Simple 3D Surface Charts .....	269
11.4.1	Sinc Surface Chart .....	271
11.4.2	Peaks Surface Chart .....	274
11.5	Parametric 3D Surface Charts .....	275
11.5.1	Sphere Surface Chart .....	277
11.5.2	Torus Surface Chart .....	278
11.5.3	Klein Bottle Surface Chart.....	280
11.5.4	Wellenkugel Surface Chart .....	282

<b>12</b>	<b>Creating Multiple Objects.....</b>	<b>285</b>
12.1	Creating Two Cubes.....	285
12.1.1	Rust Code.....	285
12.1.2	Run Application .....	292
12.2	Creating Multiple Cubes with Instancing .....	292
12.2.1	Rust Code.....	293
12.2.2	Shader Code .....	300
12.2.3	Run Application .....	300
12.3	Creating Different Objects .....	301
12.3.1	Rust Code.....	301
12.3.2	Run Application .....	303
12.4	Creating Objects Using Multiple Pipelines.....	304
12.4.1	Rust Code.....	304
12.4.2	Run Application .....	315
12.5	Creating 3D Charts with Multiple Pipelines .....	315
12.5.1	Create Wireframe Data .....	316
12.5.2	Modify Shader Program.....	317
12.5.3	Rust Code.....	318
12.5.4	Run Application .....	327
12.6	Charts with Coordinate Axes .....	329
12.6.1	Shaders for Coordinate Axes .....	329
12.6.2	Rust Code.....	330
12.6.3	Run Application .....	333
12.7	Creating 3D Charts with Multiple Render Passes.....	334
12.7.1	Rust Code.....	334
12.7.2	Run Application .....	337
<b>13</b>	<b>Compute Shaders and Particles.....</b>	<b>339</b>
13.1	Compute Shader .....	339
13.1.1	Compute Space and Workgroups.....	340
13.1.2	Write and Read Buffer .....	341
13.2	2D Rotation in GPU .....	343
13.2.1	Rust Code.....	343
13.2.2	Shader Code .....	346
13.2.3	Run Application .....	347
13.3	Compute Boids.....	347
13.3.1	Rust Code.....	347
13.3.2	Shader Code .....	354
13.3.3	Run Application .....	356

13.4 Particles under Gravity.....	358
13.4.1 Rust Code.....	358
13.4.2 Shader Code .....	366
13.4.3 Run Application .....	368
13.5 Particle Collision.....	370
13.5.1 Rust Code.....	370
13.5.2 Shader Code .....	378
13.5.3 Run Application .....	380
<b>14 Visualizing Complex Functions .....</b>	<b>383</b>
14.1 Complex Functions in Shader .....	383
14.1.1 Math Operations.....	384
14.1.2 Commonly Used Functions.....	384
14.2 Color Functions .....	386
14.2.1 Color Conversion in Shader .....	387
14.2.2 Colormaps in Shader.....	388
14.3 Domain Coloring for Complex Functions.....	392
14.3.1 Rust Code.....	392
14.3.2 Shader Code .....	397
14.3.3 Complex function with id = 0 .....	399
14.3.4 Complex Function with id = 1 .....	400
14.3.5 Complex Function with id = 2 .....	401
14.3.6 Complex Function with id = 3 .....	402
14.3.7 Complex Function with id = 4 .....	402
14.3.8 Complex Function with id = 5 .....	403
14.3.9 Complex Function with id = 6 .....	404
14.3.10 Complex Function with id = 7 .....	404
14.3.11 Complex Function with id = 8 .....	405
14.3.12 Complex Function with id = 9 .....	406
14.3.13 Complex Function with id = 10 .....	406
14.4 Domain Coloring for Iterated Functions .....	407
14.4.1 Shader Code .....	407
14.4.2 Iterated Function with id = 0.....	410
14.4.3 Iterated Function with id = 1 .....	410
14.4.4 Iterated Function with id = 2.....	411
14.4.5 Iterated Function with id = 3.....	412
14.4.6 Iterated Function with id = 4.....	412
14.4.7 Iterated Function with id = 5.....	413
14.4.8 Iterated Function with id = 6.....	414

14.4.9 Iterated Function with id = 7 .....	414
14.4.10 Iterated Function with id = 8 .....	415
14.4.11 Iterated Function with id = 9 .....	416
14.4.12 Iterated Function with id = 10 .....	416
<b>14.5 Fractal: Mandelbrot Set.....</b>	<b>417</b>
14.5.1 Mandelbrot Set Formula .....	417
14.5.2 Rust Code.....	418
14.5.3 Shader Code .....	422
14.5.4 Run Application .....	424
<b>14.6 Fractal: Julia Set.....</b>	<b>424</b>
14.6.1 Rust Code.....	424
14.6.2 Shader Code .....	429
14.6.3 Run Application .....	431
<b>14.7 3D Fractals .....</b>	<b>432</b>
14.7.1 Common Code .....	432
14.7.2 Mandelbulb .....	437
14.7.3 Mandelbrot in 3D Space .....	441
14.7.4 Mandelbox Sweeper.....	445
<b>Index .....</b>	<b>451</b>



# Introduction

## Overview

Welcome to *Practical GPU Graphics with wgpu and Rust*. The *wgpu* API is based on the WebGPU standard and is a Rust implementation of the WebGPU API specifications. WebGPU is the next-generation graphics API for the web, which is being developed by the W3C GPU for the Web Community Group with engineers from Apple, Google, Microsoft, Mozilla, and others. It is a future web standard for graphics and compute, aiming to provide modern 3D graphics and computation capabilities with GPU acceleration on the web.

The *wgpu* API is a cross-platform, safe, pure-Rust graphics API. Even though it is based on the WebGPU standard, it can run not only on the web via WebAssembly, but also natively on Vulkan, Metal, DirectX12, DirectX11, and OpenGL ES. This book will provide all the tools you need to help you create advanced 3D graphics and GPU computing in Rust on native devices using this new graphics API. I hope that this book will be useful for web developers, graphics creators, computer graphics programmers, game developers, and students of all skill levels who are interested in graphics development on the web and on devices with native modern graphics APIs.

Unlike WebGL which is based on OpenGL, WebGPU and *wgpu* are not direct ports of any existing native APIs. They are based on concepts in the Vulkan, Metal, and Direct3D12 APIs and are intended to provide high performance on these modern native graphics APIs across mobile and desktop platforms.

In order to understand *wgpu* technology, we need to review a brief history of native graphics technologies. The first to come was OpenGL, originally developed in the early 1990s. It is a low-level high performance graphics technology, which WebGL is based on. Since its inception, many graphics applications based on OpenGL have been developed. Modern GPUs actually work very differently from how the original OpenGL did – but many of the core concepts of OpenGL remain the same.

As GPUs became more complex and powerful, the graphics driver ended up having to do a lot of extremely complex work. This made graphics drivers notoriously buggy, and in many cases slower, too, as they had to do all the work on the fly. To improve OpenGL's performance, Khronos, the group behind OpenGL, proposed a new, completely redesigned modern graphics API called Vulkan, which was released in 2016. Vulkan is even more low-level, faster, and simpler, and is a much better match for modern hardware.

However, using Vulkan also meant that applications had to completely rewrite all their graphics code in order to support it. This kind of tectonic shift in technology takes years to play out, and as a result, there is still a lot of OpenGL out there.

## 2 | Practical GPU Graphics with `wgpu` and Rust

While Vulkan was designed to be a standard API able to work on all systems, as has long been the case with standards, Apple also came up with Metal for iOS and MacOS, while Microsoft came up with DirectX12 for Windows and Xbox. Both are more or less the same idea as Vulkan: new, lower-level APIs that throw out all the historical baggage and start with a clean slate design that much more closely matches how modern GPU hardware works.

With the graphics community moving on to this new generation of APIs, the question then became what to do with the web. WebGL is essentially OpenGL with many of the same pitfalls, while high-performance web game engines still stand to greatly benefit from the new generation of graphics APIs.

Unfortunately, unlike OpenGL, Vulkan has run into trouble achieving true cross-platform reach thanks to Apple. MacOS and iOS only support Metal and have no official support for Vulkan, although there are third-party libraries for it. Furthermore, Vulkan itself is still not very suitable for the web – it is just too low-level, even dealing with minutiae like GPU memory allocators so that AAA game engines can extract the maximum conceivable performance. This is overkill for web platforms, plus security is a much more significant concern in browsers.

So the solution was an all-new API design, high-level enough to be usable and secure in a browser, and able to be implemented on top of any one of Vulkan, Metal and DirectX12. This is WebGPU, which looks like the only truly cross-platform, modern, and low-level graphics API for web applications.

Based on the WebGPU standard, the `wgpu` API is a native WebGPU implementation in Rust. It can run natively on cross-platform devices with any modern graphics API such as Vulkan, Metal, or DirectX12. It can also run on the web by converting `wgpu` apps into WebAssembly packages.

Note that WebGPU and `wgpu` have not been finalized and are still in the early stages of development, so their API interfaces may change frequently before they are officially released. In addition, WebGPU and `wgpu` use a new shader language called WGSL (WebGPU Shading Language) instead of the traditional GLSL shader language used in OpenGL and WebGL applications.

*Practical GPU Graphics with wgpu and Rust* provides everything you need to create advanced 3D graphics objects in your `wgpu` applications using GPU acceleration. In this book, you will learn how to create a variety of 3D graphics and charts that range from simple 3D shapes such as cubes, spheres, cylinders, to complex 3D surface graphics such as 3D wireframes, 3D surface charts, and complex particle systems created using compute shaders. I will try my best to introduce readers to the `wgpu` API, the next-generation graphics API for native graphics devices, in a simple way – simple enough to be easily followed by programmers who have little experience in developing advanced graphics applications. You can learn from this book how to create a full range of 3D graphics applications using the `wgpu` API and WGSL shader program.

In fact, there are several bindings of the `wgpu-native` API in different programming languages, including C, Python, C# .NET, Java, and Julia, to name a few. Here, I use the Rust wrapper – `wgpu`, because of Rust's performance and safety features. Rust is a low-level, statically typed, system-programming language, which solves problems that C and C++ have been struggling with for a long time, such as errors and building concurrent programs. Compared to C and C++, Rust has three main benefits: better memory safety due to its compiler, easier concurrency due to a data ownership model that prevents data races, and zero-cost abstraction. According to a recent Stack-Overflow survey, Rust has been the most loved programming language for the last five years in a row (<https://insights.stackoverflow.com/survey/2020#technology>).

## What this Book Includes

This book and its sample code listings, which are available for download at my website at <https://drxudotnet.com>, provide you with

- A complete, in-depth instruction to practical 3D graphics programming in Rust with *wgpu*. After reading this book and running the example programs, you will be able to create various sophisticated 3D graphics and charts with GPU acceleration in your native applications.
- Over 50 ready-to-run example projects that allow you to explore the 3D graphics techniques described in this book. You can use these examples to get a better understanding of how 3D graphics and charts are created using the *wgpu* API and shader program. You can also modify the programs or add new features to them to form the basis of your own projects. Some of the example code listings provided with this book are already sophisticated chart and graphics projects, and can be directly used in your own real-world applications.
- Many functions and components in the sample code listings that you will find useful in your 3D graphics development. These functions and components include 3D transformation, projection, colormaps, lighting models created in fragment shaders, *wgpu* pipeline settings, WGSL shader code, and many other useful utility functions. You can extract these functions and components and plug them into your own applications.

## Is this Book for You?

You do not have to be an experienced Rust and graphics developer to use this book. I designed this book to be useful to people of all levels of programming experience. In fact, I believe that if you have some prior experience with programming languages such as Rust, C++, Java, R, Python, VBA, C#, or JavaScript, you will be able to sit down in front of your computer, start up Visual Studio Code, follow the examples provided in this book, and quickly become proficient with modern 3D graphics development using the *wgpu* API. For those of you who are already experienced Rust programmers or graphics/game developers, I believe this book has much to offer as well. A great deal of the information about *wgpu* programming in this book is not available in other tutorial and reference books. In addition, you can use most of the example programs in this book directly in your own real-world application development. This book will provide you with a level of detail, explanation, instruction, and sample program code that will enable you to do just about anything related to modern 3D graphics development for native devices and the web using the next-generation *wgpu* graphics API.

Throughout the book, I will emphasize the usefulness of *wgpu* graphics programming to real-world applications. If you follow the instructions presented in this book closely, you will easily be able to develop various graphics and chart applications with GPU acceleration from simple 3D shapes to 3D surfaces with powerful colormap, wireframe, and texture mapping. You can also use the compute and fragment shaders to create complicated particle systems, domain coloring for complex functions, and fractal images. At the same time, I will not spend too much time discussing program style and code optimization because there is a plethora of books out there already dealing with those topics. Most of the example programs you will find in this book omit error handlings, which makes the code easier to understand by focusing only on the key concepts and practical applications.

## What Do You Need to Use this Book?

You will need no special equipment to make the best use of this book and understand its algorithms. This book takes full advantage of open source frameworks and libraries. The sample programs accompanying this book can run on various operating systems, including Windows, Linux, iOS, and MacOS. This book uses Visual Studio Code (VS Code), Rust, and Cargo package manager for its development environment and tools. VS Code is a lightweight IDE and powerful source code editor that runs on various operating systems. It has support for Rust and WGSL with the help of relevant extensions.

Since the *wgpu* standard has not been finalized and is still in early development stage, its API may change frequently. This book uses the *wgpu* crate version 0.11 for implementing *wgpu* applications.

If you install other versions of the *wgpu* API, you may still be able to run most of the sample code with few modifications. Please remember, however, that this book is intended for that specific version of the *wgpu* API, on which all of the example programs were created and tested, so it is best to run the sample code in the same development environment and using the same version of the *wgpu* API.

In addition, your operating system needs to have a modern GPU as well as DirectX 12, Metal, or Vulkan API support on your graphics card.

## How this Book Is Organized

This book is organized into fourteen chapters, each of which covers a different topic about modern *wgpu* graphics programming. The following summaries of each chapter should give you an overview of the book's content:

### *Chapter 1, Setting Up Development Tools*

This chapter explains how to set up the packages and tools required for *wgpu* application development. VS Code, Rust, and Cargo package manager will be used as our development environment and tools. It also provides a brief introduction to Rust programming.

### *Chapter 2, WGPU Basics*

This chapter provides a brief overview on the current *wgpu* technology, and then uses a simple triangle example to explain key aspects of the *wgpu* API, including *wgpu* context, the rendering pipeline, the shader program, and rendering graphics on a window surface.

### *Chapter 3, WGPU Primitives*

This chapter demonstrates how to draw basic shapes in *wgpu*, including points, lines, and triangles. These basic shapes are referred to as primitives. There is no built-in support for curves or curved surfaces; they must be approximated by primitives. Currently, *wgpu* includes five primitives.

### *Chapter 4, GPU Buffers*

This chapter introduces GPU buffers that hold vertex data and color information, and explains how to use GPU buffers to create colorful triangles and squares with each vertex having a distinct color.

### *Chapter 5, 3D Transformations*

This chapter explains how to perform basic 3D transformations, including translation, scaling, and rotation. It also describes how to construct various matrix representations used in 3D graphics, including

the model matrix, viewing matrix, and projection matrix. These matrices will be used to display 3D graphics objects on a 2D screen.

### *Chapter 6, 3D Shapes and Camera*

This chapter shows how to use transformation, viewing, projection matrices, and the camera to create real-world 3D shapes – a 3D line and two cubes: one with distinct face colors and the other with distinct vertex colors. In doing so, you will learn two important concepts in *wgpu*: bind groups and uniform buffer objects.

### *Chapter 7, 3D Wireframe Shapes*

A wireframe model is a visual representation of 3D objects used in computer graphics. It is created by drawing just the outlines of the polygons that make up the object. This chapter explains how to create wireframe models in *wgpu* for various 3D shapes, including a cube, sphere, cylinder, cone, and torus. The key to creating 3D wireframe shapes is to specify the correct coordinates for their vertices.

### *Chapter 8, Lighting in WGpu*

This chapter demonstrates how to build a simple lighting model in *wgpu* and how to use it to simulate light sources and the way that the light they emit interacts with the objects in your scene. Here, I will discuss three light sources: ambient light, diffuse light, and specular light.

### *Chapter 9, Colormaps and 3D Surfaces*

This chapter explains how to use the color model and colormap to render the simple and parametric 3D surfaces by specifying various mathematical functions. Surfaces play an important role in various applications, including computer graphics, virtual reality, computer games, and 3D data visualization.

### *Chapter 10, Textures*

This chapter discusses 2D image textures that can be applied to a surface to make the color of the surface vary from point to point, somewhat like painting a copy of the image onto the surface. It shows how to map 2D textures onto various surfaces in *wgpu*.

### *Chapter 11, 3D Surface Charts*

Surface charts are plots of 3D data. Rather than displaying the individual data points, surface charts show a functional relationship between a dependent variable  $y$  and two independent variables  $x$  and  $z$ . This chapter explains how to create real-word 3D surface charts with colormaps, and how to add a wireframe to surface charts by mapping black square images onto the surface.

### *Chapter 12, Creating Multiple Objects*

This chapter explains several approaches used to create multiple objects in a scene. One approach is to use uniform transformations or instancing to render the same object multiple times. Another approach is to combine the vertex data of different objects together and render them as a single object. The third approach is to use different pipelines or different render passes to render different objects.

### *Chapter 13, Compute Shaders and Particles*

This chapter introduces the compute shader and describes how to use it in a simple 2D rotation example. It then applies the compute shader to particle systems – one system mimics the flocking behavior of birds, another simulates the effect of gravity on particles, and the third one models particle kinematics.

### *Chapter 14, Visualizing Complex Functions*

## 6 | Practical GPU Graphics with *wgpu* and Rust

This chapter illustrates how to generate domain coloring in *wgpu* for various complex functions. It also explains how to create fractal images for the Mandelbrot and Julia sets, as well as some 3D fractal shapes by writing the computation-intensive code directly in the fragment shaders.

## Using Code Examples

You may use the code in this book in your own applications and documentation. You do not need to contact the author for permission unless you are reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing the example code listings does require permission. Incorporating a significant amount of example code from this book into your applications and documentation also requires permission. Integrating the example code from this book into commercial products is not allowed without written permission from the author.

## Customer Support

I am always interested in hearing from readers, and would enjoy learning your thoughts on this book. You can send me comments by e-mail to [jxu@DrXuDotNet.com](mailto:jxu@DrXuDotNet.com). I also provide updates, bug fixes, and ongoing support via my website: <https://DrXuDotNet.com>.

You can also obtain the complete source code for all of the examples in this book from the abovementioned website.

# 1 Setting Up Development Tools

In this chapter, I will explain how to set up the packages and tools required for Rust-*wgpu* application development. As mentioned previously, the *wgpu* API is a native implementation of the WebGPU API in Rust, targeting both native graphics APIs and the web via WebAssembly. It is based on *gfx-hal* with the help of the *gpu-alloc* and *gpu-descriptor* crates. Its API depends on the upstream WebGPU specifications. The *wgpu* implementation consists of the following main parts:

- *wgpu*: Public Rust API for users.
- *wgpu-core*: Internal Rust API for WebGPU implementation to use.
- *wgpu-hal*: Internal unsafe GPU API abstraction layer.
- *wgpu-types*: Rust types shared between *wgpu-core* and *wgpu-rs*.
- *player*: standalone application for replaying API traces using the *winit* crate.

The *wgpu* API is an idiomatic Rust wrapper over *wgpu-core*. It is designed to be suitable for the general-purpose graphics and computation needs of the Rust community. I will now show you how to set up the development environment for *wgpu* applications.

## 1.1 Hardware Requirements

Currently, the *wgpu* API supports the following platforms:

- Windows: DirectX12, Vulkan or Angle.
- Linux & Android: Vulkan, GLES3, or Angle.
- MacOS & iOS: Metal.

Therefore, to run *wgpu* applications on your local machine, you have to install a modern graphics card with a modern graphics API. The Nvidia GeForce or AMD Radeon graphics cards with the Vulkan, DirectX12, or Metal API are good candidates for testing *wgpu* applications.

As an example, I am using a Windows 10 desktop PC with two graphics cards. Fig.1-1 shows the *Device Manager* of my PC, where two graphics cards are displayed under the *Display adapters*: one is an Intel HD Graphics 4600 and the other is an AMD Radeon R9 200 Series. The Intel graphics card, which is integrated with the motherboard, is the default graphics card. Since it has limited capability, *wgpu* applications usually cannot run on this default graphics card. For example, I tried to run a simple *wgpu* application that creates a 3D cube on my local machine. It failed to run, giving the error message shown in Fig.1-2.

## 8 | Practical GPU Graphics with wgpu and Rust

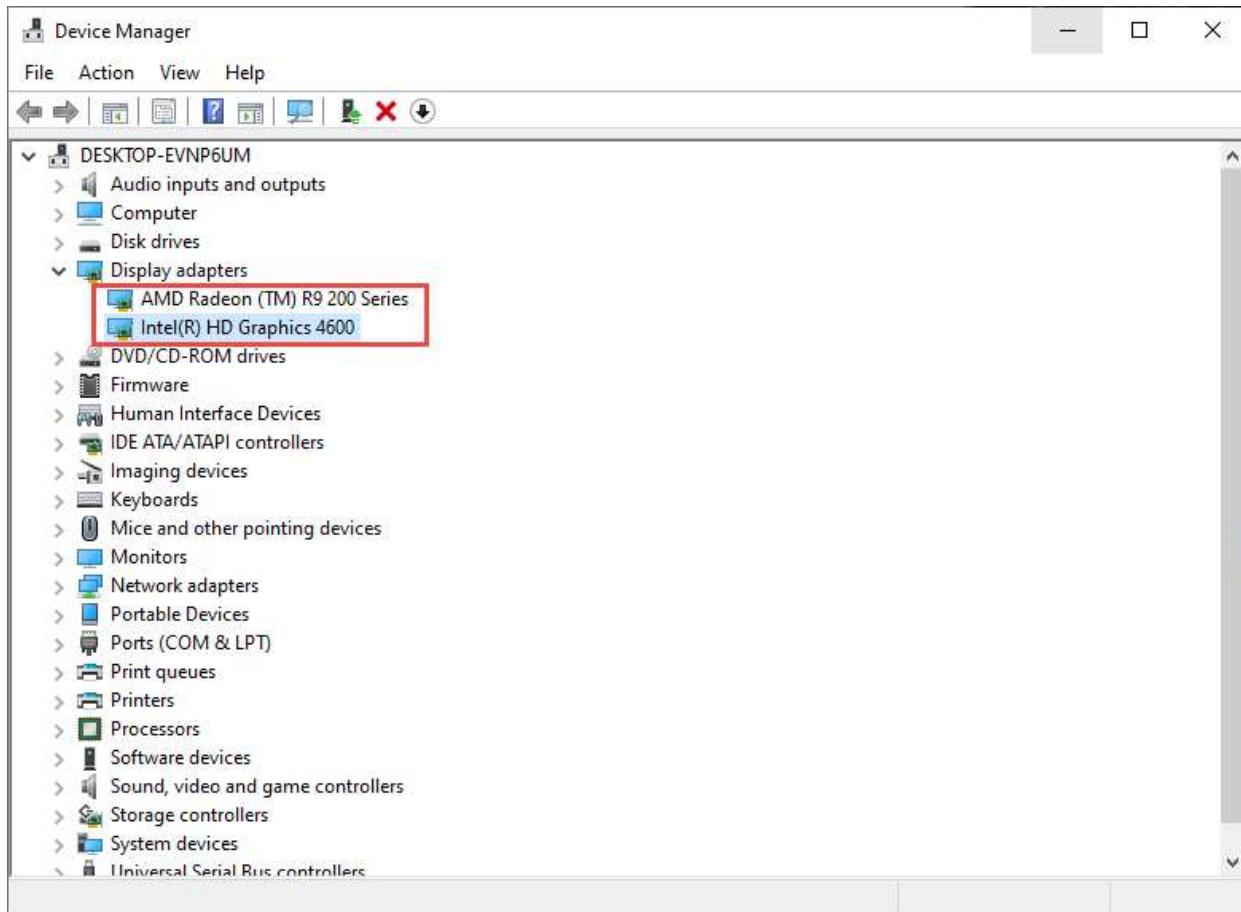


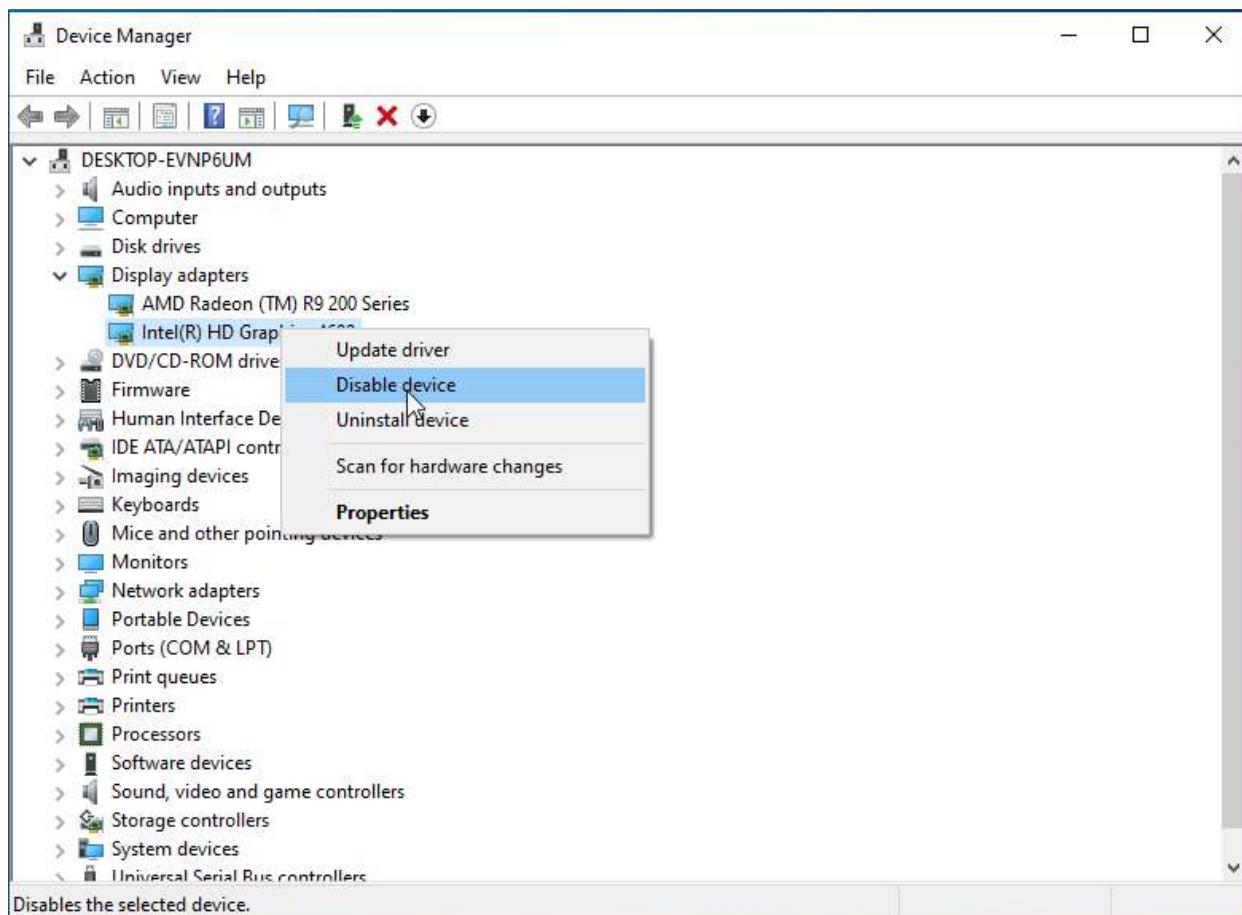
Fig.1-1. Two graphics cards installed on my PC.

```
Finished dev [unoptimized + debuginfo] target(s) in 0.98s
Running `target\debug\examples\cube.exe`
Using Intel(R) HD Graphics 4600 (Dx12)
There was a compiler error: Kernel capability not supported.
error: process didn't exit successfully: `target\debug\examples\cube.exe` (exit code: 0xc0000409, STATUS_STACK_BUFFER_OVERRUN)
```

Fig.1-2. A wgpu application cannot run on the default Intel graphics card.

You can see from Fig.1-2 that the *wgpu* application cannot run on the Intel HD Graphics 4600 even though it has a modern graphics API (DirectX 12), because it does not support kernel capability.

In order to run the application, we can try to use the AMD Radeon graphics card instead. Unfortunately, Windows 10 does not allow you to easily switch the default graphics card. One way to do this is to disable the Intel graphics card: from the *Device Manager* window, right click on Intel (R) HD Graphics 4600 and select *Disable device*, as shown in Fig.1-3. After this, my PC will use the AMD Radeon graphics card for its display.



*Fig.1-3. Disable the default Intel graphics card.*

Now, let us try to run our `wgpu` application again. This time, the application runs successfully with the following message in the terminal window, as shown in Fig.1-4.

```
Finished dev [unoptimized + debuginfo] target(s) in 0.49s
Running `target\debug\examples\cube.exe`
Using AMD Radeon (TM) R9 200 Series (Vulkan)
```

*Fig.1-4. The wgpu application runs on an AMD Radeon graphics card.*

You can see from Fig.1-4 that the `wgpu` application runs successfully on the AMD Radeon graphics card, which has a modern Vulkan API. In order to run `wgpu` Rust applications properly in both natively supported back-ends and the web via WebAssembly, I strongly encourage you to install a modern graphics card with a modern graphics API on your local machine.

## 1.2 Install C++ Build Tools

On Windows platforms, Rust requires certain C++ build tools. Either you can download the Microsoft C++ Build Tools at <https://visualstudio.microsoft.com/visual-cpp-build-tools/> or you might prefer to just install Microsoft Visual Studio (<https://visualstudio.microsoft.com/downloads/>).

While installing Visual Studio, there are several Windows workloads that I recommend you select:

- .NET desktop development.
- Desktop development with C++.
- Universal Windows Platform Development.

You might not need all three workloads, but it is likely that some dependencies will arise where they are required, so that it is just simpler to select all three.

New Rust projects default to using *Git*. So you will also need to add the individual component *Git for Windows* to the mix.

## 1.3 Install Rust

There are many ways to install Rust on your local machine. Here, we will use the official method of using *rustup* (<https://www.rust-lang.org/tools/install>). The website detects what platform you are running and offers 64- and 32-bit installers of the *rustup* tool.

The *rustup* installer installs the Rust programming language from the official release channels, enabling you to easily switch between stable, beta, and nightly compilers and keep them updated. It makes cross compiling simpler with binary builds of the standard library for common platforms.

*rustup* installs *rustc*, *cargo*, *rustup*, and other standard tools to Cargo's bin directory. This is the same directory where *cargo install* will install Rust programs and Cargo plugins. Now, in a command prompt window, you can issue the commands:

```
rustc --version
```

and

```
cargo --version
```

In my case both give version 1.55.0 as a result. If you see a version number printed, then it confirms that Rust installed correctly.

If you have installed *rustup* in the past, you can update your installation by running the following command in the command prompt window:

```
rustup update
```

This will update the Rust tools, such as *rustc* and *cargo*, to the latest version.

When the Rust installer is finished, you will be ready to program with Rust. However, you will not yet have a convenient IDE and you are not yet set up to call Windows APIs. Cargo is the name of the tool in the Rust development environment that manages and builds your projects and their dependencies.

## 1.4 Install Visual Studio Code

In this book, we will use Visual Studio Code as our text editor and IDE (integrated development environment) so we can take advantage of its language services like code completion, syntax highlighting, formatting, and debugging. Visual Studio Code also contains a built-in terminal that enables you to issue command-line commands.

Now, download Visual Studio Code from <https://code.visualstudio.com/> and install it. After installation, we also need to install the *rust-analyzer* extension. You can either install the *rust-analyzer* extension from the Visual Studio Marketplace (<https://marketplace.visualstudio.com/items?itemName=matklad.rust-analyzer>), or you can open Visual Studio Code and search for *rust-analyzer* in the extensions menu (Ctrl+Shift+X).

For debugging support, you can also install the *CodeLLDB* extension by searching for *CodeLLDB* in the extensions menu in Visual Studio Code.

## 1.5 Run a Rust Application

In this section, we will use Visual Studio Code to configure the window and dependencies that will be used for building *wgpu* applications. Now, let us start a new, simple “Hello, World!” Rust application. First, launch a command prompt window, and *cd* into the folder where you want to keep your Rust projects. Then, use *Cargo* to create a new Rust project with the following command:

```
cargo new wgpu_book
```

The argument we pass to the *cargo new* command is the name of the project that we want *Cargo* to create. Here, the project name is *wgpu\_book*. Rust recommends that you name your project using snake case: that is, words in the name are lowercase, with each space replaced by an underscore.

*Cargo* creates a project for you with the name you supply. In fact, the new project contains the source code for a very simple application that outputs a “Hello, World!” message. In addition to creating the *wgpu\_book* project, *Cargo* has created a folder named *wgpu\_book*, and has put the project’s source code files in there.

So now *cd* into that folder, and then launch Visual Studio Code from within the context of that folder:

```
cd wgpu_book
code .
```

In Visual Studio Code’s Explorer, open the *main.rs* file from the *src* folder. The *main.rs* file contains your application’s entry point (a function named *main*). Here is the code of this file:

```
fn main() {
    println!("Hello, world!");
}
```

You can tell from glancing at the code in *main.rs* that *main* is a function definition, and that it prints the string “Hello, World!” Note that when you first open a *.rs* file in VS Code, you may get a notification saying that some Rust components are not installed and asking whether you want to install them. Click *yes*, and VS Code will install the Rust language server.

Now, let us try running the application under the debugger. Put a breakpoint in line 2, and click *Run > Start Debugging* (or press F5). If you are running an application under the debugger for the first time,

## 12 | Practical GPU Graphics with wgpu and Rust

you will see a dialog box saying “Cannot start debugging because no launch configuration has been provided”. Click *OK* to see a second dialog box saying “*Cargo.toml* has been detected in this workspace. Would you like to generate launch configurations for its targets?” Click *Yes*. Then close the *launch.json* file and begin debugging again.

As you can see, the debugger breaks at line 2. Press F5 to continue, and the application runs to completion. In the terminal window, you will see the expected output “Hello, World!”

## 1.6 Configuration

For every Rust application, there is a *Cargo.toml* file in the root folder. This file is called a manifest. It is written in the TOML format, which aims to be the minimal configuration file format. Every manifest file can consist of different sections such as cargo-features, package, target tables, dependency tables, etc.

Note that in Rust, crates are similar to packages in some other languages. Crates compile individually. If a crate has child file modules, those files will get merged with the crate file and compile as a single unit. In the dependencies section of the *Cargo.toml* file, the dependencies usually contain different crates.

The examples in this book will use the following manifest file:

```
[package]
name = "wgpu_book"
version = "0.1.0"
authors = ["jack xu"]
edition = "2018"
resolver = "2"

[dependencies]
wgpu = "0.11"
cgmath = "0.18"
env_logger = "0.9"
futures = "0.3"
gfx-hal = "0.9"
image = "0.23"
log = "0.4"
pollster = "0.2"
winit = "0.25"
anyhow = "1.0"
```

Here, all the crates included in the *[dependencies]* section can be found at the <https://crates.io/> website:

- *wgpu*: Rusty WebGPU API Wrapper which allows you to access the *wgpu* API.
- *cgmath*: A linear algebra and mathematics library for computer graphics.
- *env\_logger*: A logging implementation for “log” which is configured via an environment variable.
- *futures*: An implementation of futures and streams featuring zero allocations, composability, and iterator-like interfaces.
- *gfx-hal*: *gfx-rs* hardware abstraction layer.
- *image*: Imaging library written in Rust, which provides basic filters and decoders for the most common image formats.
- *log*: A lightweight logging façade for Rust.

13

- *pollster*: A minimal async executor that lets you block on a future.
- *winit*: Cross-platform window creation library.
- *anyhow*: Provides *anyhow::Error*, a trait object-based error type for easy idiomatic error handling in Rust applications.

Depending on the requirements of your applications, you may need to add more crates to the `[dependencies]` section later.

In the `[package]` section, the *edition* field indicates which Rust edition your package is compiled with. Setting the *edition* field will affect all targets/crates in the package. Currently, the valid *edition* fields are 2015, 2018, and 2021.

In addition, as of version 0.11, *wgpu* requires cargo’s newest feature resolver. So, we set the *resolver* version to “2” in the `[package]` section. The version “1” *resolver* is the original resolver that shipped with Cargo up to version 1.50, and is the default if the resolver is not specified. The version “2” resolver introduces changes in feature unification, which means a different feature resolver is used that uses a different algorithm for unifying features. The version “2” resolver will avoid unifying features in situations where features for target-specific dependencies are not enabled if the target is not currently being built. On my Windows 10 PC, I will get the following compiled error if I do not set *resolver* = “2” in the *Cargo.toml* file:

```
compile_error!("Metal API enabled on non-Apple OS. ...")
```

Since the WGSL shading language will be used in our *wgpu* applications, we will add the WGSL extension to VS Code. Open VS Code, search for WGSL in the extension menu (Ctrl+Shift+X), and install it. This extension will provide syntax highlighting for our WGSL shaders.

In Cargo’s parlance, an example is nothing else but the Rust source code of a standalone executable that typically resides in a single `.rs` file. All such files should be placed in the `examples/` directory, at the same level as `src/` and the *Cargo.toml* manifest itself.

Here, we will use Cargo’s examples configuration. Now, add a new folder called *examples* to the root directory. We will add all our example source code to this folder. In this folder, we create two subfolders, *ch01\_test01* and *ch01\_test02*, each with its own `main.rs` file. Fig.1-5 shows the file structure of our *wgpu\_book* project



Fig.1-5. The file structure of our *wgpu\_book* project.

## 14 | Practical GPU Graphics with wgpu and Rust

The `main.rs` files in these two subfolders have the following content respectively:

```
fn main() {
    println!("Hello from ch01_test01!");
}
```

and

```
fn main() {
    println!("Hello from ch01_test02!");
}
```

We can then use the default syntax to run the programs in the `examples/` folder, as found in the Cargo documentation (<https://doc.rust-lang.org/cargo/reference/cargo-targets.html#examples>). For example, we can issue the following `cargo run` command in the terminal window to run the `ch01_test01` example:

```
cargo run --example ch01_test01
```

The output will be displayed on your terminal pane “*Hello from ch01\_test01!*” You can also run the `ch01_test02` in a similar manner.

As you can see, the way we run examples is very similar to how we would run `src/bin` binaries, which some people use as normal entry point to their Rust programs. The important thing is that you do not have to worry about what to do with your example code anymore. All you need to do is drop your code in the `examples/` directory, and let Cargo do the rest.

The `examples` configuration will work well for a simple project that contains only a few examples. However, if our project contains over fifty examples, the `examples/` folder will contain too many subfolders, which will make our project hard to manage.

Here, we will manually manage our example projects – we will create a subfolder named `ch01` that will contain all the Rust files used to create all the example projects for Chapter 1. Similarly, we will add `ch02`, `ch03`, ..., subfolders to the `examples/` folder. This way, our `examples/` folder contains only the book-chapter subfolders.

Now, we can copy the `main.rs` file in the `examples/ch01_test01/` folder, then paste it to the `examples/ch01/` folder, and rename it to `test01.rs`. Do the same thing for the `main.rs` file in the `examples/ch01_test02/` folder. Now, the `examples/ch01/` folder has two Rust files, `test01.rs` and `test02.rs`. We can then delete the `examples/ch01_test01/` and `examples/ch01_test02` folders.

One issue associated with this configuration is that we cannot use the default syntax to run programs in the `examples/` folder. Instead, we will have to manually configure our example files in the `Cargo.toml` file. Add the following code to the `Cargo.toml` file:

```
[[example]]
name = "ch01_test01"
path = "examples/ch01/test01.rs"

[[example]]
name = "ch01_test02"
path = "examples/ch01/test02.rs"
```

We can then run these two examples with the following command:

```
cargo run --example ch01_test01
cargo run --example ch01_test02
```

## 1.7 Creating a Window

In this book, we will use the *winit* crate to open up windows that will be used to display our *wgpu* graphics. *winit* is a cross-platform window creation and management library. It can create windows and lets you handle events produced by windows.

*winit* is designed to be a low-level brick in a hierarchy of libraries. Consequently, in order to show something in the window you need to use platform-specific getters provided by *winit*, or another library. Here, I will show you how to open an empty window.

Inside VS Code, open the *test02.rs* file in the *examples/ch01* folder and replace its content with the following code:

```
use winit::{
    event::{Event, WindowEvent},
    event_loop::{ControlFlow, EventLoop},
    window::Window,
};

fn main() {
    let event_loop = EventLoop::new();
    let window = Window::new(&event_loop).unwrap();
    window.set_title("my window");
    env_logger::init();

    event_loop.run(move |event, _, control_flow| {
        *control_flow = ControlFlow::Wait;
        match event {
            Event::WindowEvent {
                event: WindowEvent::CloseRequested,
                ..
            } => *control_flow = ControlFlow::Exit,
            _ => {}
        }
    });
}
```

All this code does is create a window that stays open until you close it. Now, execute the following command in your terminal pane:

```
cargo run --example ch01_test02
```

This will open up an empty window as shown in Fig.1-6.

## 1.8 Rust Basics

In this book, I assume that you are already somewhat familiar with Rust, as I will not go into much detail on Rust syntax. To help you feel more comfortable with Rust programming, here I will provide a brief review on Rust basics.

The goal of Rust is to be a language for creating highly reliable and fast software in a simple way. As you can see from Fig.1-7 by comparison to other languages, Rust is designed to provide speed and safety at the same time. It can be used to write high-level programs down to hardware-specific programs. In fact, Rust is a system programming language focused on three goals: safety, speed, and concurrency.

## 16 | Practical GPU Graphics with wgpu and Rust

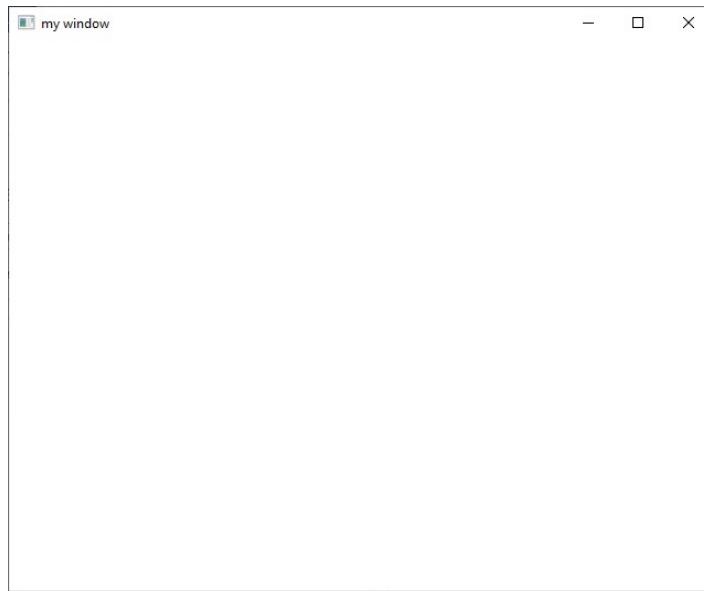


Fig.1-6. Empty window created using winit.

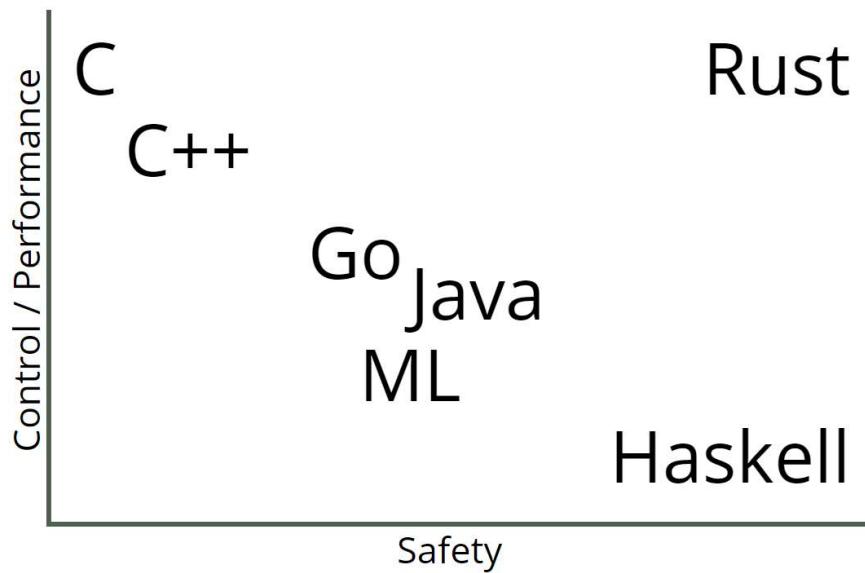


Fig.1-7. Safety and performance of Rust.

Rust is a young and modern language. It is a compiled programming language and it uses the LLVM compiler infrastructure on its backend. It is also a multi-paradigm language, which supports imperative, procedural, concurrent actor, object-oriented, and pure functional styles. It also supports generic programming and meta programming, in both static and dynamic styles. Rust does not have a garbage collector (GC) by design. This improves performance at runtime.

Rust also supports WebAssembly (WASM), which helps execute high computation intensive algorithms in browsers, on embedded devices, or anywhere else. It runs at the speed of native code. Rust can be compiled to WebAssembly for fast, reliable execution in web applications.

## 1.8.1 Data Types

In Rust, the Type System represents different types of values. It checks the validity of supplied values before they are stored or manipulated by the program. This ensures that the code behaves as expected. Rust is a statically typed language. Every value in Rust is of a certain data type. The compiler can automatically infer the data type of a variable based on the value assigned to it.

Rust uses the *let* keyword to declare a variable:

```
let first_name = "Joe"; // string type
let x = 0.5;           // float type
let is_true = false;   // boolean type
```

In the above example, the data type of the variables will be inferred from the values assigned to them.

Rust has four primary scalar types: integer, float-point, Boolean, and character.

Constants in Rust represent values that cannot be changed. Constants must be explicitly typed. For example:

```
const MY_INTEGER: i16 = 5; // declare an integer constant
const MY_FLOAT: f16 = 2.34; // declare a float constant
```

You can see that we use the *const* keyword to declare a constant, and that all characters in a constant name are usually in uppercase.

Note that a variable declared using the *let* keyword is immutable by default. You can set it as mutable by using the *mut* keyword:

```
let mut my_variable = 1.5;
```

While constants are always immutable.

Rust allows us to declare variables with the same name. In such a case, the new variable simply overrides the previous variable:

```
let _variable = 2.3;
let _variable = "Test";
println!("{}", _variable);
```

You can see that we re-declare not only the value but also the data type (from float to string). The output will be “Test”.

Note that unlike variables, constants cannot be re-declared using the same name.

## 1.8.2 Functions

Rust declares functions using the *fn* keyword. A function is a set of statements that perform a specific task. A function declaration tells the compiler about the function’s name, return type, and input arguments. Here is a *main* function example:

```
fn main() {
    println!("Hello from ch01_test02!");
}
```

Here is a function with input arguments:

```
fn add(x:f32, y:f32) {
```

## 18 | Practical GPU Graphics with wgpu and Rust

```
    println!("{}", x + y);
}
```

Here is a function with returned value:

```
fn add(x:f32, y:f32) -> f32 {
    x + y
}
```

This function uses shorthand syntax for the returned value; that is, no semicolon means  $x + y$  is returned. The following function has an explicit return statement:

```
fn add(x:f32, y:f32) -> f32 {
    return x + y;
}
```

This is actually a bad practice in Rust. It should be used only in conditional return statements, except if it is the last expression.

A function must be called to execute it. This process is called function invocation. For example, the *add* function can be called from the *main* function:

```
fn main() {
    let sum = add(2.5, 4.2);
    println!("sum = {}", sum);
}
```

This will give the output “*sum = 6.7*” on your terminal pane.

### 1.8.3 Control Flows

In Rust, control flows are the order in which certain code is executed or evaluated.

#### 1.8.3.1 If and if let

Rust implements *if* and *if let* expressions, which are conditional branches in program control. The syntax of an *if* expression is a conditional operand, followed by a consequent block, any number of *else if* conditions and blocks, and an optional trailing *else* block. Here is an example of using the *if* expression:

```
if x == 1 {
    println!("x is one");
} else if x == 2 {
    println!("x is two");
} else {
    println!("x is something else");
}
```

Here is another example:

```
let age = 18;
let is_child = if age < 18 { true } else { false };
println!("Is child: {}", is_child);
```

Note that we omit the semicolon behind the *true* and *false*. This allows us to return the value as a result of the *if* statement.

An *if let* expression is semantically similar to an *if* expression, but in place of a condition operand it expects the keyword *let* followed by a pattern, an *=*, and a scrutinee operand. For example:

```

19
let dish = ("Ham", "Eggs");

// this body will be skipped because the pattern is refuted
if let ("Bacon", b) = dish {
    println!("Bacon is served with {}", b);
} else {
    // This block is evaluated instead.
    println!("No bacon will be served");
}

// this body will execute
if let ("Ham", b) = dish {
    println!("Ham is served with {}", b);
}

```

### 1.8.3.2 Match

Rust also provides pattern matching via the *match* keyword. The *match* operator can be used to match a single value against a variety of patterns. For example

```

let shoes_size = 39;
let shoes_type = match shoes_size {
    35..37 => "S",
    38..40 => "M",
    41..43 => "L",
    _ => "Not available"
};
println!("{}", sheos_type);

```

The output is “*M*”. The *match* patterns must be exhaustive. If not all cases are explicitly handled, the last statement must be a “catch-all” without conditions.

The *match* blocks can also be used to replace a set of *if-else* statements:

```

let age = 15;
let description = match age {
    _ if age < 10 => "child",
    _ if age >= 18 => "adult",
    _ => "teenager",
};
println!("{}", description);

```

The output is “*teenager*”. The *\_s* in the above examples are actually variables that we ignore. In Rust, variables that are not used can be ignored with an *\_*-prefix, or simply an *\_*, like in these examples.

### 1.8.3.3 For Loops

The for-loop is a common way to repeat an action a number of times:

```

for count in 0..10 {
    print!("{} ", count)
}

```

Rust also supports custom step-sizes while iterating:

```

for count in (0..10).step_by(2) {
    print!("{} ", count)
}

```

## 20 | Practical GPU Graphics with wgpu and Rust

The *for*-loop can also work with arrays or vectors:

```
let companies = ["Apple", "Amazon", "Google", "Microsoft"];
for i in 0..companies.len() {
    println!("Current company: {}", companies[i]);
}
```

You can also turn an array into a simple iterator like below:

```
for company in companies.iter() {
    println!("Current company: {}", company);
}
```

### 1.8.3.4 While Loops

Like in other languages, the *while* keyword in Rust can be used to run a loop while a condition is true. Here is an example:

```
let mut num = 1;
while num <= 101 {
    println!("Current value : {}", num);
    num += 1;
}
```

Note that there is no `++` or `--` in Rust. You have to use `num += 1` in order to increment.

You can also use the *break* and *continue* keywords inside a while loop:

```
let mut num = 0;
while num < 10 {
    if num == 2 {
        println!("Skip value : {}", num);
        num += 1;
        continue;
    } else if num == 6 {
        println!("Break At : {}", num);
        break;
    }
    println!("Current value : {}", num);
    num += 1;
}
```

### 1.8.3.5 Loop Loops

Rust provides a *loop* keyword to indicate an infinite loop. The *break* statement can be used to exit a loop at any time, whereas the *continue* statement can be used to skip the rest of the iteration and start a new one.

```
let mut count = 0;
loop {
    count += 1;
    if count == 3 {
        println!("three");
        continue;
    }

    println!("{}", count);
    if count == 5 {
        println!("OK, Exit");
    }
}
```

```
21
    break;
}
}
```

This is in fact an infinite loop that skips the rest of the iteration when `count == 3` and exits the loop when `count == 5`.

## 1.8.4 Arrays, Vectors, and Slices

Rust implements several types to represent a sequence of elements, including arrays, vectors, and slices. An array in Rust is a fixed-size list of elements of the same type. By default, arrays are immutable:

```
let arr = [1, 2, 3];          // arr: [i32; 3]
let mut arr = [4, 5, 6];      // mut arr: [i32; 3]
```

There is a shorthand for initializing each element of an array to the same value. In the following example, each element of `arr` will be initialized to 5:

```
let arr = [5; 10];           // arr: [i32; 10]
```

Arrays have type `[T; N]`, where T represents a generic type.

You can get the number of elements in an array `arr` with `arr.len()`, and use `arr.iter()` to iterate over them with a loop. The following code snippet will print each element in order:

```
let arr = ["Yahoo", "Apple", "Amazon", "Google"];      // arr: [&str; 4]
println!("arr has {} elements", arr.len());
for ele in arr.iter() {
    println!("{}:", ele);
}
```

You can access a particular element of an array using subscript notation. For example, `arr[1]` gives the output “Apple”. Like in most programming languages, subscripts start from zero in Rust.

Unlike an array, a vector in Rust is a dynamic array, implemented as the standard library type `Vec<T>`. Vectors always allocate their data on the heap. You can create a vector in two ways: one way is to use the `new` keyword:

```
let mut v1 = Vec::new();
v1.push(1); // [1]: add an element at the end
v1.push(2); // [1, 2]
v1.push(3); // [1, 2, 3]
v1.pop();   // [1, 2]; remove an element from the end
```

The other way is to use the `vec!` macro:

```
let mut v2 = vec![1, 2, 3]; // v2: Vec<i32>
```

A *slice* is a reference to an array. It is useful for allowing safe, efficient access to a portion of an array without copying:

```
let arr = ["Yahoo", "Apple", "Amazon", "Google", "Microsoft"];
let middle = &arr[1..4];
```

Here, `middle` represents a *slice* of array `arr`: just the elements 1, 2, and 3:

```
for e in middle.iter() {
    println!("{}:", e);
}
```

## 22 | Practical GPU Graphics with wgpu and Rust

This prints the output: “Apple, Amazon, Google”.

You can also take a slice of a *vector*, *String*, and *&str*, because they are backed by arrays. Slice have type *&[T]*.

### 1.8.5 Structs

Structs are used to encapsulate related properties into one unified datatype. The members of a struct can be different data types. By convention, the name of the *struct* starts with a capital letter and follows *CamelCase*. There are three types of structs, which can be created using the *struct* keyword: classic C structs, tuple structs, and unit structs.

Here is a classic C-like struct:

```
struct Person {
    first_name: String,
    last_name: String,
    is_teenager: bool
}
```

This type of struct is the most commonly used. Each field has a name and a type, and once defined it can be accessed using the *example\_struct.field* syntax. The fields of a struct share its mutability, so:

```
let mut p = Person{
    first_name: "Jack".to_string(),
    last_name: "London".to_string(),
    is_teenager: false
};

p.first_name = "Joe".to_string();
println!("{} {}, is_teenager: {}", p.first_name, p.last_name, p.is_teenager);
```

The output is: “*Joe London, is\_teenager: false*”. Here we reset the first name, which is only possible if *p* is mutable.

The tuple struct is similar to the C-like struct, but its fields have no names. It is like a hybrid between a tuple and a struct. A tuple struct has a name but its member fields do not. It is declared with the *struct* keyword, and then with a name followed by a tuple. For example:

```
struct Color(u8, u8, u8);
struct Point(i32, i32, i32);
```

We can then use them in this way:

```
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Here, *black* and *origin* are not equal, even though they contain the same values.

Note that it is always better to use a classic struct than a tuple struct. We would write *Color* and *Point* like the following instead:

```
Struct Color {
    red: u8,
    green: u8,
    blue: u8
}

Struct Point {
```

23

```
x: i32,
y: i32,
z: i32
}
```

There is one case when a tuple struct is very useful, and that is when it has only one member. We call this the *newtype* pattern, because it allows you to create a new type that is distinct from its contained value and also expresses its own semantic meaning:

```
struct Kilometer(i32);
let distance = Kilometer(30);
let Kilometer(distance_in_km) = distance;
println!("The distance: {}", distance_in_km); // The distance: 30 km
```

Here, we extract the inner integer type through a destructuring *let*, as with regular tuples.

The other type of struct is the unit-like struct, which is rarely useful on its own. But in combination with other features, it can become useful. We can define a struct with no members at all:

```
struct Electron;
let x = Electron;
```

This *struct* is called a unit-like struct because it resembles the empty tuple, (), sometimes called *unit*. Like a tuple struct, it defines a new type.

## 1.8.6 Enums

An *enum* in Rust is a single type. It contains variants that are possible values of the *enum* at a given time. For example:

```
enum Equity {
    MSFT,
    AMZN,
    AAPL,
    IBM,
    SPY
}
```

We use the :: syntax to use the name of each variant: they are scoped by the name of the *enum* itself. This allows both of these to work:

```
Equity::MSFT;
```

Note that each enum variant can have no data (unit variant), unnamed ordered data (tuple variant), and named data (struct variant). For example:

```
enum Message {
    Success,           // a unit variant
    Warning {category: i32, message: String}, // a struct variant
    Error(String)     // a tuple variant
}
let message = Message::Warning {category: 3, message: String::from("Field x is required")};
```

A value of an *enum* type contains information about which variant it is, in addition to any data associated with that variant.

## 1.8.7 Generics

Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. In Rust, we can do this with generics. Instead of declaring a specific data type, we can use an uppercase letter  $T$  (or any other letter) as the data type, but we must first inform the compiler that that letter  $T$  is a generic type by writing  $<T>$ .

We can write functions that take generic types with a similar syntax:

```
fn take_anything<T>(x: T){
    // do something with x
}
```

Multiple arguments can have the same generic type:

```
fn take_two_of_something<T>(x: T, y: T){
    // do something with x and y
}
```

We can also write a version that takes multiple generic types:

```
fn take_two_things<T, U>(x: T, y: U){
    // do something with x and y
}
```

We can also store a generic type in a struct as well:

```
struct Point<T> {
    x: T,
    y: T,
}

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Similarly to functions,  $<T>$  is where we declare the generic parameters, and we then use  $x: T$  in the type declaration too.

The generic type can also be used in enums:

```
enum Option<T> {
    Some(T),
    None
}
```

The above *Option* type is a special generic type that is already defined in Rust's standard library; it can have the value of either *Some* value or *None*.

We can use the *Option* enum as follows:

```
let x: Option(i32) = Some(5);
let y: Option(f32) = Some(5.0);
```

You can see that a generic enum with a single definition can have multiple uses.

# 2 wgpu Basics

Computer graphics deals with all aspects of generating images using a computer. It has changed greatly over the years, progressing from static images to powerful animation and 3D graphics, GPU and gaming capabilities. All of these technologies are powered by a variety of graphics APIs, including OpenGL, WebGL, DirectX, Vulkan, Metal, etc. Now, the next-generation graphics API – *wgpu*, which is based on WebGPU, is expected to provide a new low-level graphics standard to leverage the computing powers and hardware acceleration of the latest GPU technologies.

In this chapter, I will use a simple triangle example to explain the key aspects of the *wgpu* API, including *wgpu* context, the rendering pipeline, the shader program, and rendering graphics on a native window surface.

## 2.1 First *wgpu* Example

Now, we are ready to start working towards our first *wgpu* example. I will show you how to create a simple triangle in *wgpu* with Rust. As mentioned previously, the examples in this book will be created using the “official” WGSL shader.

First, I will show you how to create a simple triangle, and then I will explain the code structure and render pipeline in this chapter.

### 2.1.1 WGSL Shaders

A typical *wgpu* application consists of two parts of code: one being the shader code that runs on the GPU, and the other being the control code written in the Rust programming language. The shader code is a little program that runs for each specific section of the graphics pipeline. You could say a shader is nothing more than a program that transforms inputs to outputs. Shaders are also isolated programs in that they are not allowed to communicate with each other; the only communication they have is via their input and output variables.

Let us add a new subfolder called *ch02/* to the *examples/* folder, and then add a new file, *first\_triangle.wgsl*, to this newly created folder with the following code:

```
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>, 3>(
        vec2<f32>(<0.0, 0.5),
        vec2<f32>(-0.5, -0.5),
```

## 26 | Practical GPU Graphics with `wgpu` and Rust

```
    vec2<f32>( 0.5, -0.5)
);
return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

[[stage(fragment)]]
fn fs_main() -> [[location(0)]] vec4<f32> {
    return vec4<f32>(1.0, 1.0, 0.0, 1.0);
}
```

This WGLS shader code consists of vertex and fragment shaders. Inside the `vs_main` function of the vertex shader, we first create a `vec2<f32>` array that contains three vertex points used to create our triangle. We then convert the `vec2<f32>` vertices into `vec4<f32>` points as returned built-in positions, which can be used by `wgpu` to render the triangle.

The fragment shader simply defines the color with a `vec4<f32>` type, which represents (R, G, B, A) color components: red, green, blue, and alpha. Here, the alpha component represents the transparency. Note that each color component has a value range [0.0, 1.0]. For `f32` type in WGLS, the zero or other digits after the decimal point are required. For example, the following variable

```
vec4<f32>(1, 1, 0, 1)
```

is invalid and the application will not work. You have to change it to

```
vec4<f32>(1.0, 1.0, 0.0, 1.0)
```

### 2.1.2 Common Rust Code

Next, we will discuss the control code of our `wgpu` application. The control code will be written in Rust. We will add a commonly used Rust code called `common.rs` to the `examples/ch02/` folder, which will be shared across different example applications. Type the following code in this `common.rs` file:

```
use wgpu::{IndexFormat, PrimitiveTopology, ShaderSource};
use winit::{
    event::{Event, WindowEvent},
    event_loop::{ControlFlow, EventLoop},
    window::Window,
};

pub struct Inputs<'a> {
    pub source : ShaderSource<'a>,
    pub topology: PrimitiveTopology,
    pub strip_index_format: Option<IndexFormat>,
}

pub async fn run(event_loop: EventLoop<()>, window: Window, inputs: Inputs<'_>, num_vertices: u32) {
    let size = window.inner_size();
    let instance = wgpu::Instance::new(wgpu::Backends::VULKAN);
    let surface = unsafe { instance.create_surface(&window) };
    let adapter = instance
        .request_adapter(&wgpu::RequestAdapterOptions {
            power_preference: wgpu::PowerPreference::Default(),
            compatible_surface: Some(&surface),
            force_fallback_adapter: false,
        })
        .await
        .expect("Failed to find an appropriate adapter");
```

```

let (device, queue) = adapter
    .request_device(
        &wgpu::DeviceDescriptor {
            label: None,
            features: wgpu::Features::empty(),
            limits: wgpu::Limits::default(),
        },
        None,
    )
    .await
    .expect("Failed to create device");

let format = surface.get_preferred_format(&adapter).unwrap();
let mut config = wgpu::SurfaceConfiguration {
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    format: format,
    width: size.width,
    height: size.height,
    present_mode: wgpu::PresentMode::Mailbox,
};
surface.configure(&device, &config);

// Load the shaders from disk
let shader = device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: None,
    source: inputs.source,
});

let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: None,
    bind_group_layouts: &[],
    push_constant_ranges: &[],
});
let render_pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: None,
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[format.into()],
    }),
    primitive: wgpu::PrimitiveState{
        topology:inputs.topology,
        strip_index_format:inputs.strip_index_format,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
event_loop.run(move |event, _, control_flow| {
    let _ = (&instance, &adapter, &shader, &pipeline_layout);
    *control_flow = ControlFlow::Wait;
    match event {

```

## 28 | Practical GPU Graphics with `wgpu` and Rust

```
Event::WindowEvent {
    event: WindowEvent::Resized(size),
    ..
} => {
    // Recreate the surface with the new size
    config.width = size.width;
    config.height = size.height;
    surface.configure(&device, &config);
}
Event::RedrawRequested(_) => {
    let frame = surface.get_current_texture().unwrap();
    let view = frame.texture.create_view(&wgpu::TextureViewDescriptor::default());
    let mut encoder =
        device.create_command_encoder(&wgpu::CommandEncoderDescriptor { label: None });
    {
        let mut rpass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: None,
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {r: 0.05, g:0.062, b:0.08, a:1.0}),
                    store: true,
                },
            }],
            depth_stencil_attachment: None,
        });
        rpass.set_pipeline(&render_pipeline);
        rpass.draw(0..num_vertices, 0..1);
    }
    queue.submit(Some(encoder.finish()));
    frame.present();
}
Event::WindowEvent {
    event: WindowEvent::CloseRequested,
    ..
} => *control_flow = ControlFlow::Exit,
_ => {}
});
```

The code first introduces the `winit` events and window that are used to display our triangle. It then defines the `Inputs` struct that is used to set the input parameters for the `run` function. A detailed explanation about the `wgpu` API used in the `run` function will be given in the following sections.

### 2.1.3 Rust Main Function

Now, add a new Rust file called `first_triangle.rs` to the `examples/ch02/` folder, and then enter the following content into it:

```
mod common;

use winit::{
    event_loop::{EventLoop},
};

use std::borrow::Cow;
```

```

fn main() {
    let event_loop = EventLoop::new();
    let window = winit::window::Window::new(&event_loop).unwrap();
    window.set_title("ch02_first_triangle");
    env_logger::init();

    let inputs = common::Inputs{
        source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("shader.wgsl"))),
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None
    };

    pollster::block_on( common::run(event_loop, window, inputs, 3));
}

```

Here, we first load the file modules from the *common.rs* file, and then introduce the *winit* window. Next, we define the input struct and pass it to the *run* function implemented in the *common.rs* file.

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch02_first_triangle"
path = "examples/ch02/first_triangle.rs"

```

We can then run this application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch02_first_triangle
```

This brings up a new window that displays a yellow-colored triangle as shown in Fig.2-1.

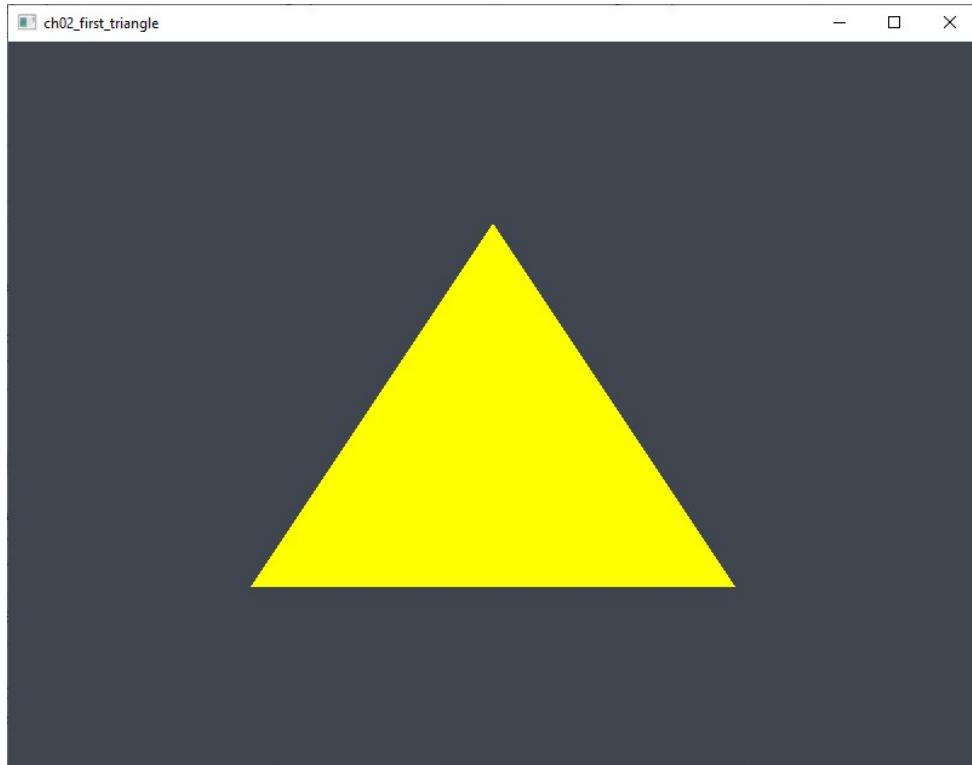


Fig.2-1. Yellow triangle created using wgpu.

## 2.2 *wgpu* API

From the triangle example presented in the preceding section, you can see the difference between *wgpu* and WebGL – *wgpu* separates resource management, work preparation, and submission to the GPU, while in WebGL, a single context object is responsible for everything, and contains many associated states. In the following subsections, we will walk through the example code to better understand the *wgpu* program structure and render pipeline.

### 2.2.1 *wgpu* Backend

You can see from the *common.rs* file in the common folder that the initial step of *wgpu* rendering context is to introduce a window that is used to display the rendered image. Here, we first set the physical size of the window and then create an instance of the *wgpu* backend that depends on the graphics card installed on your local machine. In my case, I have an AMD Radeon R9 200 series graphics card with a Vulkan API.

```
let size = window.inner_size();
let instance = wgpu::Instance::new(wgpu::Backends::VULKAN);
```

You can see here that we use *Backends::VULKAN*, which returns the Vulkan API. In fact, you can set it to a specific backend according to your machine. Here are some valid options:

- VULKAN: Supported on Windows, Linux/Android, and macOS/iOS via Vulkan Portability.
- GL: OpenGL, currently unsupported.
- METAL: Supported on macOS/iOS.
- DX12: Supported on Windows 10.
- DX11: Supported on Windows 7+.
- BROWSER\_WEBGPU: Supported when targeting the web through WebAssembly.
- PRIMARY: All the APIs that *wgpu* offers first tier support for (Vulkan + Metal + DX12 + Browser WebGPU).
- SECONDARY: All the APIs that *wgpu* offers second tier support for (OpenGL + DX11).
- all(): returns the set containing all backends supported by the *wgpu* API.

Here, we can also set it to PRIMARY or *all()*, and then rerun the application, which should generate the same result shown in Fig.2-1.

Next, we use the *unsafe* code keyword to create a rendered surface for a *winit* window:

```
let surface = unsafe { instance.create_surface(&window) };
```

This surface provides a drawing functionality for the platform supported by *winit*.

Since the *wgpu* API is *async* when interacting with the GPU, we need to place our rendering code inside the *async* function *run*, which is executed when the code is loaded. Within this *async run* function, we can access the GPU in *wgpu* by calling the *request\_adapter* function.

```
let adapter = instance
    .request_adapter(&wgpu::RequestAdapterOptions {
        power_preference: wgpu::PowerPreference::default(),
        compatible_surface: Some(&surface),
        force_fallback_adapter: false,
```

```

})
.await
.expect("Failed to find an appropriate adapter");

```

Here, it is required that the `compatible_surface` field is representable with the requested adapter. This does not create the surface, it only guarantees that the adapter can present said surface. The `force_fallback_adapter` field indicates that if it is `true`, only a fallback adapter can be returned. This is generally a “software” implementation on the system.

Once we have the GPU adapter, we can call the `adapter.request_device` function to create a GPU device:

```

let (device, queue) = adapter
    .request_device(
        &wgpu::DeviceDescriptor {
            label: None,
            features: wgpu::Features::empty(),
            limits: wgpu::Limits::default(),
        },
        None,
    )
    .await
    .expect("Failed to create device");

```

Here, the `features` attribute of `DeviceDescriptor` allows you to specify extra features that you may want. For this simple triangle example, we do not use any extra features. Note that the device configuration of your machine may limit the features you can use. If you want to use certain features, you may need to limit what devices you support, or provide work arounds.

The `limits` attribute describes the limit of certain types of resources we create. Here, we use the default, which supports most devices.

## 2.2.2 Surface

As mentioned in the preceding section, we use the `unsafe` code keyword to create a rendered surface, which is the part of the window that we will draw on. We need this surface in order to write the output from the fragment shader directly to the screen.

Next, we use the `wgpu` adapter to configure the surface with the following code snippet:

```

let format = surface.get_preferred_format(&adapter).unwrap();
let mut config = wgpu::SurfaceConfiguration {
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    format: format,
    width: size.width,
    height: size.height,
    present_mode: wgpu::PresentMode::Mailbox,
};
surface.configure(&device, &config);

```

Here, we first introduce the `format` parameter that defines how the surface texture will be stored on the GPU. Different displays prefer different formats, so we call the `surface.get_preferred_format` function to figure out the best format to use based on the display we are using.

Inside the `SurfaceConfiguration` struct, the `usage` attribute describes how the surface textures will be used. `RENDER_ATTACHMENT` specifies that the textures will be used to write to the surface defined in the window.

## 32 | Practical GPU Graphics with `wgpu` and Rust

The `width` and `height` attributes define the size of the surface texture, which should usually be the width and height of the `winit` window. Make sure that the width and height of the surface texture are not zero. Otherwise, it may cause your application to crash.

The `present_mode` attribute uses the `wgpu::PresentMode::Mailbox` enum, which determines how to synchronize the surface with the display. The `PresentMode` enum has three options:

- *Immediate*: The presentation engine does not wait for a vertical blanking period and the request is presented immediately. This is a low-latency presentation mode, but visible tearing may be observed. Will fallback to *Fifo* if unavailable on the selected platform and backend. Not optimal for mobile.
- *Mailbox*: The presentation engine waits for the next vertical blanking period to update the current image, but frames may be submitted without delay. This is a low-latency presentation mode and visible tearing will not be observed. Will fallback to *Fifo* if unavailable on the selected platform and backend. Not optimal for mobile.
- *Fifo*: The presentation engine waits for the next vertical blanking period to update the current image. The framerate will be capped at the display refresh rate, corresponding to the *VSync*. Tearing cannot be observed. Optimal for mobile.

As mentioned, we use the `Mailbox` enum here.

### 2.2.3 Load Shaders

Next, we load the shaders that we will implement in the `first_triangle.wgsl` file using the `create_shader_module` function:

```
let shader = device.create_shader_module(&wgpu::ShaderModuleDescriptor {  
    label: None,  
    source: inputs.source,  
});
```

The `create_shader_module` function can create a shader module from either SPIR-V or WGSL source code. In this book, we will use the official WGSL shader module.

The `inputs.source` is defined in the `first_triangle.rs` file with the following code snippet:

```
let inputs = common::Inputs {  
    source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("first_triangle.wgsl"))),  
    topology: wgpu::PrimitiveTopology::TriangleList,  
    strip_index_format: None  
};
```

Here, the type `Cow` is a smart pointer providing clone-on-write functionality: it can enclose and provide immutable access to borrowed data, and clone the data lazily when mutation or ownership is required. The type is designed to work with general borrowed data via the `Borrow` trait.

### 2.2.4 Rendering Pipeline

What we have done so far are basic initialization steps, including setting up the *GPU adapter*, *GPU device*, and surface texture. All of these objects are common to any `wgpu` application and they do not need to be reset or changed. However, after this initialization, the rendering pipeline and the shading program will differ depending on the application.

The pipeline in the `wgpu` API describes all the actions the GPU will perform when acting on a set of data. It usually consists of two programmable states, the vertex shader and fragment shader, similar to WebGL. The `wgpu` API also adds support for compute shaders, which exist outside the rendering pipeline.

To render our triangle, we need to configure this pipeline, create the shaders, and specify vertex attributes. In `wgpu`, the `render_pipeline` object is used to specify the different pieces of the pipeline. The configuration of the components of this pipeline, such as the shaders, vertex state, and render output state, are fixed, allowing the GPU to better optimize rendering for the pipeline. Meanwhile, the buffer or textures bound to the corresponding inputs or outputs can be changed.

Here, we first need to create a `pipeline_layout`, which is useful when we need to use the GPU buffer. For our simple triangle example, it is just a placeholder without any meaningful content:

```
let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: None,
    bind_group_layouts: &[],,
    push_constant_ranges: &[],,
});
```

We can now create the rendering pipeline that combines the shaders, vertex, fragment, and the output configuration by calling the `device.create_render_pipeline` function:

```
let render_pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: None,
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],,
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[format.into()],
    }),
    primitive: wgpu::PrimitiveState{
        topology: inputs.topology,
        strip_index_format: inputs.strip_index_format,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
```

The pipeline requires the `vertex` and `fragment` attributes, which correspond to the vertex shader and fragment shader respectively. Here, we can specify which function inside of the shader should be called, which is known as the `entry_point`. We assign the `vs_main` and `fs_main` functions, implemented previously in the `first_triangle.wgsl` file, as the `entry_point` attributes for the `vertex` and `fragment` shader respectively.

The `buffers` attribute inside the `vertex` section tells `wgpu` what type of vertices we want to pass to the vertex shader. Here, for our simple triangle example, we define the vertices in the vertex shader directly, so we simply leave this attribute empty. We will need to place something here when we use the GPU buffer to store vertex data.

The `fragment` attribute is technically optional, so we have to wrap it inside the `Some()` enum variant. We only need it if we want to store color data to the surface texture. The `targets` attribute inside the `fragment`

## 34 | Practical GPU Graphics with `wgpu` and Rust

section tells `wgpu` which color output format it should set up. Here, we simply use the surface *format*, which specifies the set of output slots and the texture format. Our fragment shader has a single output slot for the color, which we will write directly to the surface texture image. Thus, we specify a single color state for an image with this format.

The *primitive* attribute describes how to interpret our vertices when converting them into triangles. Here, we set this attribute as an input parameter defined in the `first_triangle.rs` file:

```
let inputs = common::Inputs {
    source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("first_triangle.wgsl"))),
    topology: wgpu::PrimitiveTopology::TriangleList,
    strip_index_format: None
};
```

You can see that we set the topology attribute to `PrimitiveTopology::TriangleList`, indicating that every three vertices will correspond to one triangle. Here, we also set the `strip_index_format` to `None` because we want to create a `TriangleList` but not a `TriangleStrip` primitive. The `primitive` field also contains other attributes that we set to their default values.

We are not using the `depth_stencil` attribute for our current example, so we leave it as `None`. The `multisample` attribute determines how many samples this pipeline will use. Here, we simply use its default value.

### 2.2.5 Rendering Output

The final step is to record and submit GPU commands using a GPU command encoder. Note that the `GPUCommandEncoder` is directly derived from the `MTLCommandEncoder` in Metal; while in DirectX 12 and Vulkan, it is called `GraphicsCommandList` and `CommandBuffer` respectively. Since the GPU is an independent coprocessor, all GPU commands are executed asynchronously. This is why GPU commands are built up into a list and sent in batches when needed. In `wgpu`, the GPU command encoder returned by the `device.create_command_encoder` function is a Rust object that builds a batch of buffered commands that will be sent to the GPU at some point:

```
let frame = surface.get_current_texture().unwrap();
let view = frame.texture.create_view(&wgpu::TextureViewDescriptor::default());

let mut encoder = device.create_command_encoder(&wgpu::CommandEncoderDescriptor { label: None });
{
    let mut rpass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: None,
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &frame.view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {r: 0.05, g:0.062, b:0.08, a:1.0}),
                store: true,
            },
        }],
        depth_stencil_attachment: None,
    });
    rpass.set_pipeline(&render_pipeline);
    rpass.draw(0..num_vertices, 0..1);
}

queue.submit(Some(encoder.finish()));
frame.present();
```

You can see that inside the `device.create_command_encoder` function, we open a render pass by calling the `encoder.begin_render_pass` function. We name it `rpass` and make it mutable. This render pass accepts a parameter of type `RenderPassDescriptor` as a render pass option that has several attributes: one is a required field called `color_attachments`, which is an array attached to the current render channel that stores image information. In our example, it stores the background color as

```
ops: wgpu::Operations {
    load: wgpu::LoadOp::Clear(wgpu::Color {r: 0.05, g: 0.062, b: 0.08, a: 1.0}),
    store: true,
},
```

Here, `Color {r: 0.05, g: 0.062, b: 0.08, a: 1.0}`) represents [R, G, B, alpha] in the `load` attribute, while in the `view` attribute, the rendering result is stored on the current image of the frame texture, i.e., `frame.texture.create_view`.

The other attribute in the `RenderPassDescriptor` is called `depth_stencil_attachment`, which stores the depth information of the rendering pass and the template information. Our triangle example does not need depth information because it is actually drawing a 2D flat triangle with no depth, so here, we set it to `None`.

Next, we assign the pipeline to the render pass and call the `rpass.draw` function to draw our triangle with 3 vertices and 1 instance, since the input parameter `num_vertices` is set to 3 in our `first_triangle.rs` file. This is where the built-in `vertex_index` defined in the vertex shader comes from.

Note that in order to present the `SurfaceTexture` called `frame` returned by the `get_current_texture` method, we need to first submit all instructions to the queue of the GPU device for execution, with some work rendering to this texture, and then call the `frame.present()` method to schedule the texture to be presented on the surface.

In the `main` function, we use the `block_on` function of the `pollster` crate to call our async `run` function. The `pollster` crate is a minimal async executor that lets us block on a future. After running the command, our triangle will be written to the surface and displayed in the `winit` window, as shown in Fig.2-1.

In the above discussion, I showed you the basic steps for creating a simple triangle using the `wgpu` API. In the following sections, I will explain how the shader program works in `wgpu`.

## 2.3 Shader Program

The shader code used in `wgpu` has to be specified in a binary format as opposed to a human-readable syntax like WGSL or GLSL. This binary format is called SPIR-V, which can be used to write graphics and compute shaders. However, this does not mean that we need to write this binary code by hand. In fact, we can write shaders in `wgpu` using either GLSL or WGSL and then convert them into SPIR-V binary.

In WebGPU, we can convert GLSL code into SPIR-V using the `npm` package called `@webgpu/glslang`. Furthermore, some modern browsers, such as Chrome Canary and Microsoft Edge Canary, come with a built-in compiler that can internally compile text-based WGSL shader code into SPIR-V binary.

In `wgpu`, we can use the `shaderc` utility crate to compile GLSL shaders into the SPIR-V binary. For WGSL shaders, the `wgpu` API converts them into SPIR-V internally, so we do not need to do anything manually. In this book, we will use the official WGSL shaders to build our example applications.

### 2.3.1 Why Use WGSL Shaders?

As mentioned previously, we can use GLSL to write shaders and then compiled them into SPIR-V binary. This seems like an easy approach for *WebGPU* or *wgpu* shading. So, why did the WebGPU working group decide to create a new WGSL shading language? At the moment, WGSL is so new that it is still in the draft stage (<https://www.w3.org/TR/WGSL/>).

There has been a bit of controversy regarding the choice to create a new WGSL instead of going for the widely supported GLSL. The rationale behind WGSL is as follows: first, we would not be able to directly use GLSL in its current state because we need separate textures and samplers in WebGPU or *wgpu*. If the extension tried to fit the existing GLSL syntax as much as possible, we would have to construct *sample2D* and other things on the fly every time we needed to sample or fetch any texels. This is more like a hack than it is a design anyone would want to buy. Secondly, GLSL was made for a world where each shader only has a single entry point and execution model. This does not match any one of SPIR-V, HLSL, or Metal. We could try to stretch GLSL apart and attempt to support multiple entry points, but then we would hit other problems like conflicting locations and binding points. What we would then end up with would not be nearly the same GLSL for either users or tooling.

The goals of WGSL can be listed as follows:

- Trivially convertible to SPIR-V.
- Constructs are defined as normative references to their SPIR-V counterparts.
- All features in WGSL are directly translatable to SPIR-V (no polymorphism, no general points, no overloads, etc.).
- Features and semantics are exactly the ones of SPIR-V.
- Each item in the WGSL spec must provide the mapping to SPIR-V for the construct.

### 2.3.2 Writing Shader Code

Here, I will show you how to use WGSL to create a vertex shader and a fragment shader. In addition, the *wgpu* API also adds support for compute shaders that do not exist in WebGL.

Programs implemented in WGSL have a *main* function that is invoked for every object. The language includes many features to aid in graphics programming, such as built-in vector and matrix primitives. It also includes functions for operations like cross products, matrix-vector products, and reflections around a vector. The vector type in WGSL is called *vec* plus a number indicating the number of elements. For example, a 3D position would be stored in a *vec3*. It is possible to access single components through members like *.x*, but it is also possible to create a new vector from multiple components at the same time. For example, the expression *vec3<f32>(1.0, 2.0, 3.0).xy* would result in *vec2<f32>*. The constructors of vectors can also take combinations of vector objects and scalar values. For example, a *vec3* can be constructed with *vec3<f32>(vec2<f32>(1.0, 2.0), 3.0)*.

As mentioned, we need to implement a vertex shader and a fragment shader in order to create a simple triangle in the *winit* window. A vertex shader processes each incoming vertex. It takes attributes, such as world position, color, normal, and texture coordinates as inputs. Its outputs are the final position in clip coordinates and attributes such as color and texture coordinates, which need to be passed on to the fragment shader. These values will then be interpolated over the fragments to produce a smooth color gradient.

Note that a clip coordinate is a 4D vector from the vertex shader that is subsequently converted into a normalized device coordinate by dividing the whole vector by its last component. These normalized device coordinates are homogeneous coordinates that map the framebuffer to a  $[-1, 1]$  by  $[-1, 1]$  coordinate system, as shown in Fig.2-2.

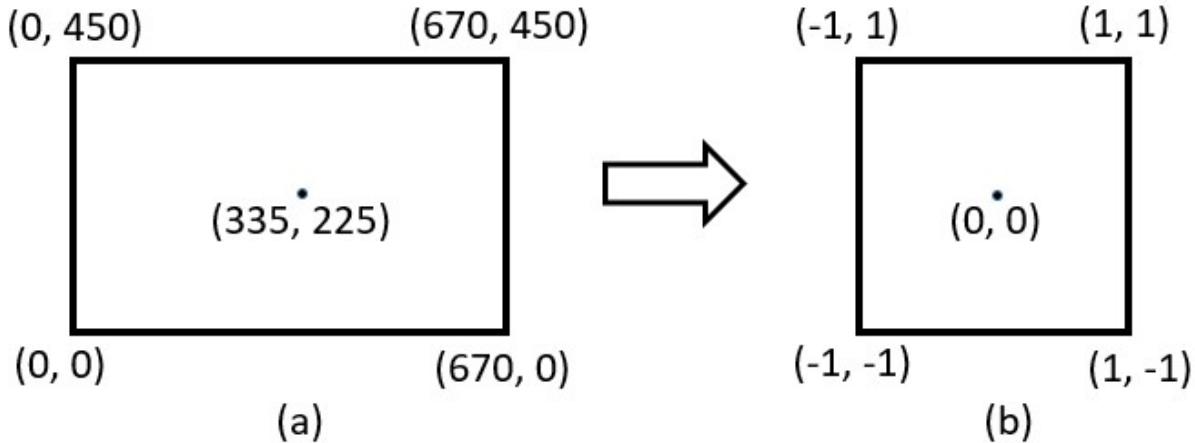


Fig.2-2. Framebuffer coordinates (a) and normalized device coordinates (b).

You may already be familiar with these coordinates if you have previously worked on computer graphics. Note that the z-coordinate in the device coordinate system uses the range  $[0, 1]$ .

For a triangle without any applied transformations, we can directly specify the positions of the three vertices as device coordinates, resulting in the triangle shape shown in Fig.2-3.

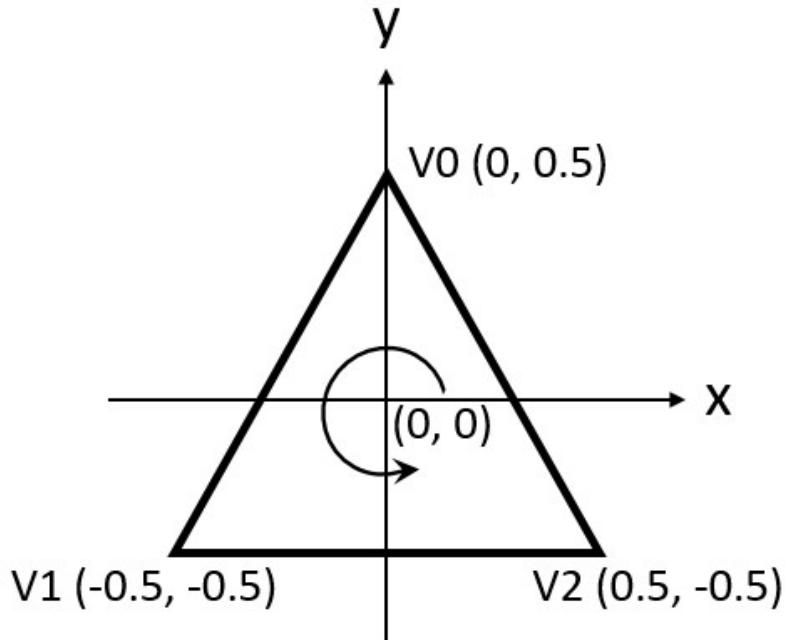


Fig.2-3. Vertex positions of a triangle in normalized device coordinates. The vertices should be arranged in counterclockwise order.

## 38 | Practical GPU Graphics with `wgpu` and Rust

From Fig.2-3, we can directly write down normalized device coordinates by outputting them as clip coordinates from the vertex shader with the last component set to 1. This way, the division to transform clip coordinates to normalized device coordinates will not change anything.

Normally these coordinates would be stored in a vertex buffer, but here we can specify them directly in the vertex shader since all we are making is the simple triangle shown in Fig.2-3. In later chapters, I will show you how to use both the vertex buffer and fragment buffer to create complex `wgpu` applications. The following is the vertex shader code for our triangle example:

```
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,3>(
        vec2<f32>( 0.0, 0.5),
        vec2<f32>(-0.5, -0.5),
        vec2<f32>( 0.5, -0.5)
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}
```

The `vs_main` function is invoked for every vertex shader. The built-in `vertex_index` variable contains the index of the current vertex. This is usually the index into the vertex buffer, but in our simple triangle example, it will be the index into a hardcoded array of vertex data. The position of each vertex is accessed from the `pos` array in the shader and combined with dummy `z` and `w` components to produce a position in clip coordinates. The output from the `vs_main` function is the built-in variable `position`.

The other part of the shader code is the fragment shader. The triangle defined by the positions from the vertex shader fills an area on the surface of the `winit` window with fragments. The fragment shader is invoked on three fragments to produce a color and depth for the framebuffer. A simple hardcoded fragment shader that generates a yellow-colored triangle looks like this:

```
[[stage(fragment)]]
fn fs_main() -> [[location(0)]] vec4<f32> {
    return vec4<f32>(1.0, 1.0, 0.0, 1.0);
}
```

The `fs_main` function is called for every fragment, just like the `vs_main` function in the vertex shader. Here, the color in the fragment shader is a four-component vector (`vec4<f32>`) representing the R, G, B, and alpha channels within the range [0, 1]. Unlike the built-in `position` in the vertex shader, there is no built-in variable to output a color for the current fragment. We need to specify our own output variable for each framebuffer where the `layout(location = 0)` modifier specifies the index of the framebuffer. The yellow color is written to the returned variable that is linked to the first (and only) framebuffer at index 0.

Compared with GLSL, WGLS shaders look verbose and cumbersome. This is why some people do not like the WGLS syntax. However, if you want to access the new features of WebGPU or `wgpu`, you should go with WGLS because it will become the “official” shading language in the future.

The advantage of using WGLS is that you do not need to compile shaders into binary because the `wgpu` API can directly take WGLS code to render `wgpu` graphics.

## 2.4 Triangle with Different Vertex Colors

After going through the code structure and shader code for a typical `wgpu` application, here we will build another `wgpu` application – one that creates a triangle with different vertex colors.

## 2.4.1 Shader Code

Add a new shader file called *triangle\_vertex\_color.wgsl* to the *examples/ch02/* folder with the following code:

```
struct VOutput{
    [[location(0)]] v_color: vec4<f32>;
    [[builtin(position)]] position: vec4<f32>;
};

[[stage(vertex)]]
fn vs_main(
    [[builtin(vertex_index)]] in_vertex_index: u32) -> VOutput {
    var pos = array<vec2<f32>,3>(
        vec2<f32>( 0.0, 0.5),
        vec2<f32>(-0.5, -0.5),
        vec2<f32>( 0.5, -0.5)
    );
    var color = array<vec3<f32>,3>(
        vec3<f32>(1.0, 0.0, 0.0),
        vec3<f32>(0.0, 1.0, 0.0),
        vec3<f32>(0.0, 0.0, 1.0)
    );
    var out: VOutput;
    out.position = vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
    out.v_color = vec4<f32>(color[in_vertex_index], 1.0);
    return out;
}

[[stage(fragment)]]
fn fs_main(in: VOutput) -> [[location(0)]] vec4<f32> {
    return in.v_color;
}
```

Here, we first define a struct called *VOutput* that includes two fields: one is *v\_color*, which represents the output vertex color, and the other is the built-in vertex positions. Inside the *vs\_main* function, we create three vertices using a *vec2* array, and three vertex colors using a *vec3* array. These three red, green, and blue colors correspond to three different vertices respectively. The *vs\_main* function returns the *out* variable that is a *VOutput*-type.

The fragment function *fs\_main* takes the *VOutput* as its input argument and sets its return value to the output vertex color *v\_color* from the vertex shader. This is different from the fragment shader used in the previous example where we set the output color to a manually coded yellow color.

## 2.4.2 Rust Code

The Rust code in the example is very simple because we can reuse the modules defined in the *common.rs* file. Add a new Rust file called *triangle\_vertex\_color.rs* to the *examples/ch02/* folder with the following code:

```
mod common;

use winit::{
    event_loop::{EventLoop},
};
use std::borrow::Cow;
```

## 40 | Practical GPU Graphics with wgpu and Rust

```
fn main() {
    let event_loop = EventLoop::new();
    let window = winit::window::Window::new(&event_loop).unwrap();
    window.set_title("ch02_triangle_vertex_color");
    env_logger::init();

    let inputs = common::Inputs{
        source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("triangle_vertex_color.wgsl"))),
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None
    };

    pollster::block_on( common::run(event_loop, window, inputs, 3));
}
```

In fact, this *main* function is also identical to that used in the previous example. The only difference between the examples is the shader file.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch02_triangle_vertex_color"
path = "examples/ch02/triangle_vertex_color.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch02_triangle_vertex_color
```

This produces a triangle with different vertex colors, as shown in Fig.2-4.

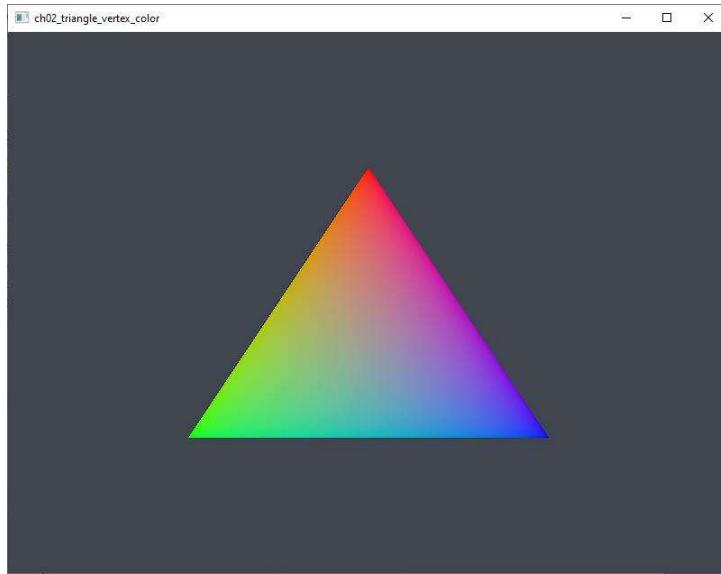


Fig.2-4. Triangle with different vertex colors.

You can see from the figure that the color changes from one vertex to another gradually and smoothly. This is because color in *wgpu* is interpolated internally over fragments to produce a smooth color gradient.

# 3 wgpu Primitives

In the preceding chapter, I explained the basic procedure for creating a simple triangle in *wgpu*. In fact, the *wgpu* API can draw only a few basic shapes, including points, lines, and triangles. These basic shapes are referred to as primitives. There is no built-in support for curves or curved surfaces; they must be approximated by primitives.

Currently, the *wgpu* API consists of five primitives, which can be specified using the *primitive.topology* attribute inside the rendering pipeline:

```
enum wgpu::PrimitiveTopology {
    PointList,
    LineList,
    LineStrip,
    TriangleList,
    TriangleStrip,
};
```

Suppose we have six vertices: v1, v2, v3, v4, v5, and v6. They will generate different shapes depending on the primitive type:

- *PointList*: Six point primitives are composed at each vertex position.
- *LineList*: Line primitives are composed from (v1, v2), then (v3, v4), then (v5, v6). Each subsequent primitive takes two vertices.
- *LineStrip*: Line primitives are composed from (v1, v2), then (v2, v3), then (v3, v4), then (v4, v5), and then (v5, v6). Each subsequent primitive takes one vertex.
- *TriangleList*: Triangle primitives are composed from (v1, v2, v3), then (v4, v5, v6). Each subsequent primitive takes three vertices.
- *TriangleStrip*: Triangle primitives are composed from (v1, v2, v3), then (v2, v3, v4), then (v3, v4, v5), then (v4, v5, v6). Each subsequent primitive takes one vertex.

In this chapter, I will show you how to use the *wgpu* API to create various primitives, which will help you become more familiar with the *wgpu* rendering pipeline and drawing images on the *winit* window.

## 3.1 Creating Points

The *point-list* is the simplest primitive in `wgpu`. Here we will use six vertices, as shown in Fig.3-1, to create six points.

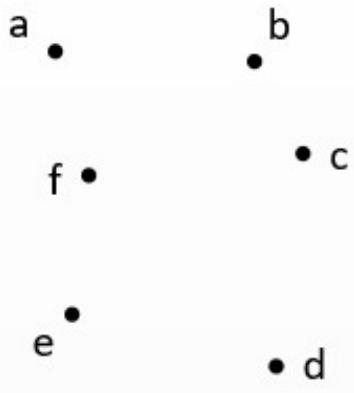


Fig.3-1. Six vertices used to create points.

### 3.1.1 Rust Code

Add a new folder called `ch03` to the `examples/` folder, and then add a file named `point_line.rs` to this newly created folder with the following code:

```
#[path = "../ch02/common.rs"]
mod common;

use winit::{
    event_loop::EventLoop,
};

use std::borrow::Cow;

fn main() {
    let mut primitive_type = "point-list";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        primitive_type = &args[1];
    }

    let mut topology = wgpu::PrimitiveTopology::PointList;
    let mut index_format = None;
    if primitive_type == "line-list" {
        topology = wgpu::PrimitiveTopology::LineList;
        index_format = None;
    } else if primitive_type == "line-strip" {
        topology = wgpu::PrimitiveTopology::LineStrip;
        index_format = Some(wgpu::IndexFormat::UInt32);
    }

    let inputs = common::Inputs{
        source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("point_line.wgsl"))),
        topology: topology,
    }
}
```

```

    strip_index_format: index format
};

let event_loop = EventLoop::new();
let window = winit::window::Window::new(&event_loop).unwrap();

window.set_title(&format!("{}: {}", "Primitive", primitive type));
env_logger::init();
pollster::block_on( common::run(event_loop, window, inputs, 6));
}

```

Here, we first introduce the `common.rs` file implemented in the `examples/ch02/` folder by specifying the path explicitly because the file is located in a different folder. This means that we will use the same common file as the previous chapter in our current example.

Inside the `main` function, we first set the `primitive_type` to “point-list”. In order to avoid code duplication, we make the program accept command line arguments with the code snippet:

```

let args: Vec<String> = std::env::args().collect();
if args.len() > 1 {
    primitive_type = &args[1];
}

```

Here, we use a function from Rust’s standard library, `std::env::args`, which returns an iterator of the command line arguments that were given to our `main` program. This iterator produces a series of values, and we can call the `collect` function on the iterator to convert it into a collection, `Vec<string>`, that contains all the elements the iterator produces. We set the `primitive_type` to the second element of the collection (`args[1]`) because the first element represents the project name.

There are three possible command line arguments: “*point-list*”, “*line-list*”, and “*line-strip*”. The default setting is “*point-list*”.

The rest of the code is similar to that used in the preceding examples.

### 3.1.2 Shader Code

We now need to add a new shader called `point_line.wgsl` to the `examples/ch03/` folder. The following is the code for this new shader:

```

[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,6>(
        vec2<f32>(-0.5, 0.7),
        vec2<f32>(+0.3, 0.6),
        vec2<f32>(+0.5, 0.3),
        vec2<f32>(+0.4, -0.5),
        vec2<f32>(-0.4, -0.4),
        vec2<f32>(-0.3, 0.2)
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

[[stage(fragment)]]
fn fs_main() -> [[location(0)]] vec4<f32> {
    return vec4<f32>(1.0, 1.0, 0.0, 1.0);
}

```

## 44 | Practical GPU Graphics with `wgpu` and Rust

Here, we also create vertex data directly in the shader program as we did for our triangle example in the preceding chapter. We define six vertex positions using a `vec2<f32>` array, which will be used to generate our point list, line list, and line strip.

### 3.1.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]  
name = "ch03_point_line"  
path = "examples/ch03/point_line.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch03_point_line "point-list"
```

Here, “`point-list`” is the input argument for our `primitive_type` variable. This produces six points shown in Fig.3-2.

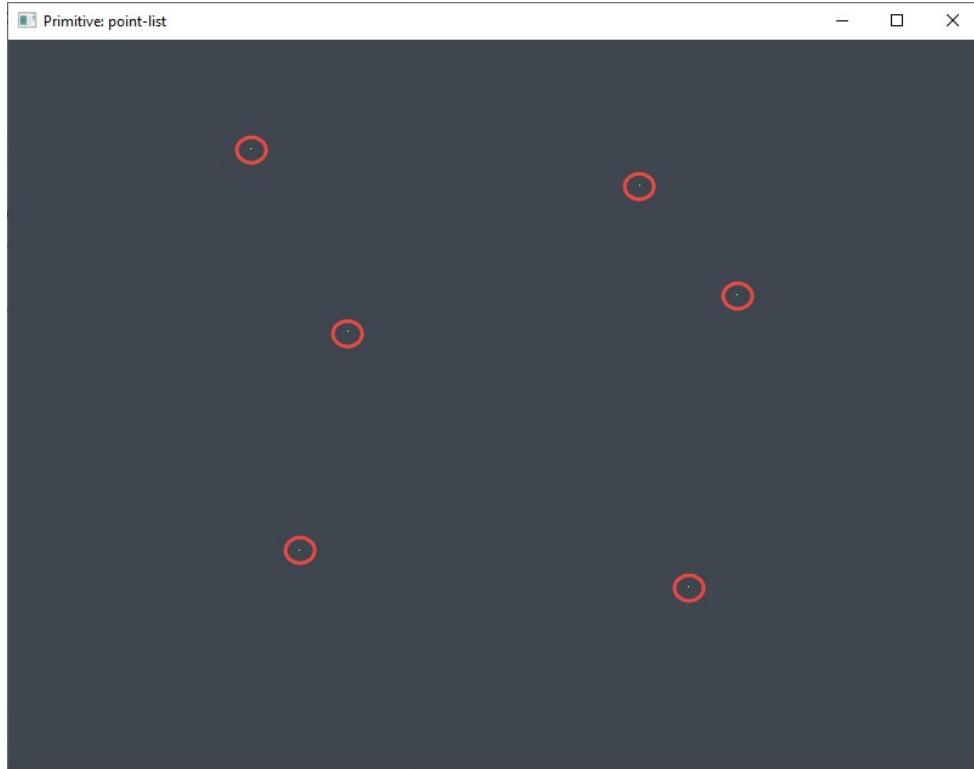
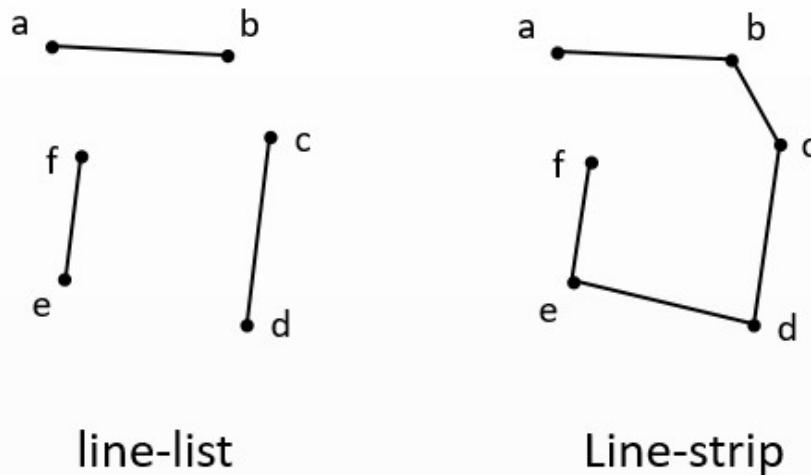


Fig.3-2. Create Points.

You may notice from the figure that the size of the points is only `1px`, which is too small to be seen in the figure. This is why I have highlighted those points using red circles. Unfortunately, the current `wgpu` API and WebGPU do not support changing the size of points and the thickness of lines; they are all set to `1px` by default. Hopefully, there will be an extension in the future, which will allow us to change point size and line width.

## 3.2 Creating Lines

In this section, I will show you how to create line segments in *wgpu* using the same set of vertices used in our previous point example. Here is what you get if you use the same six vertices with two different line primitives shown in Fig.3-3.



*Fig.3-3. Lines we will create using the line-list and line-strip primitives.*

We can create the line-list and line-strip primitives using the same code that was used to create the point-list primitive in the preceding example.

Note that for the line-strip primitive, we have to set the *index\_format* attribute:

```
if primitive_type == "line-list" {
    topology = wgpu::PrimitiveTopology::LineList;
    index_format = None;
} else if primitive_type == "line-strip" {
    topology = wgpu::PrimitiveTopology::LineStrip;
    index_format = Some(wgpu::IndexFormat::UInt32);
}
```

You can see here that we set the *index\_format* attribute to *None* for the line-list primitive, while we set it to “*UInt32*” for the line-strip primitive.

Now, we can run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch03_point_line "line-list"
```

Here, “*line-list*” is the input argument for our *primitive\_type* variable. This produces the result shown in Fig.3-4.

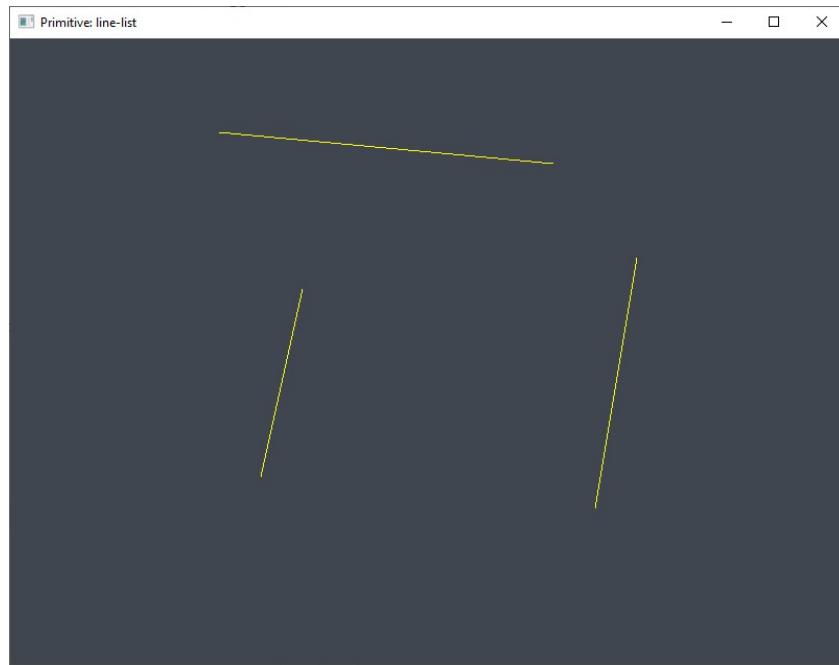


Fig.3-4. Line-list primitive.

Now, execute the following `cargo run` command in the terminal window:

```
cargo run --example ch03_point_line "line-strip"
```

Here, “`line-strip`” is the input argument for our `primitive_type` variable. This produces the result shown in Fig.3-5.

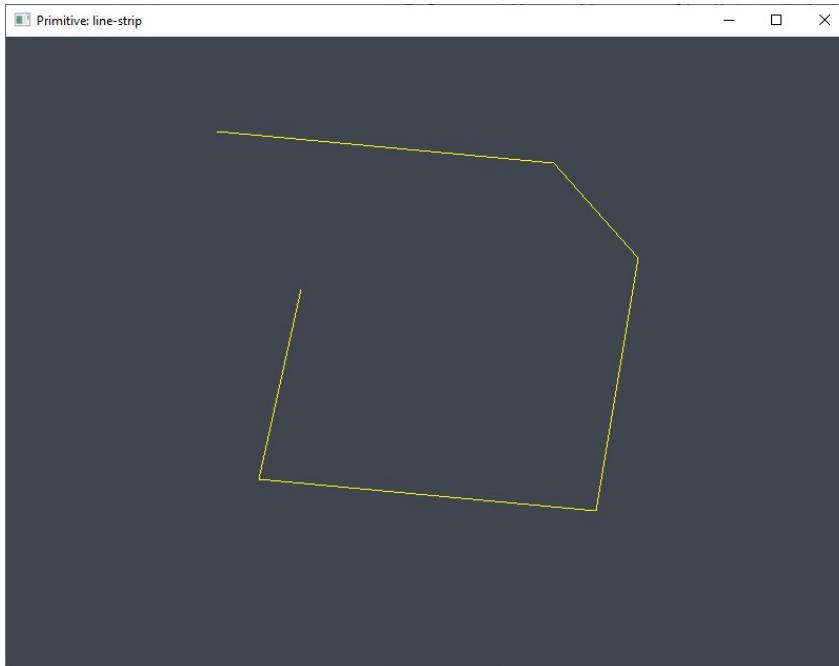


Fig.3-5. Line-strip primitive.

## 3.3 Creating Triangles

In this section, I will show you how to use the *triangle-list* and *triangle-strip* primitives to create multiple triangles in *wgpu*. Here is what you get if you use the same set of nine vertices with two different triangle primitives, shown in Fig.3-6.

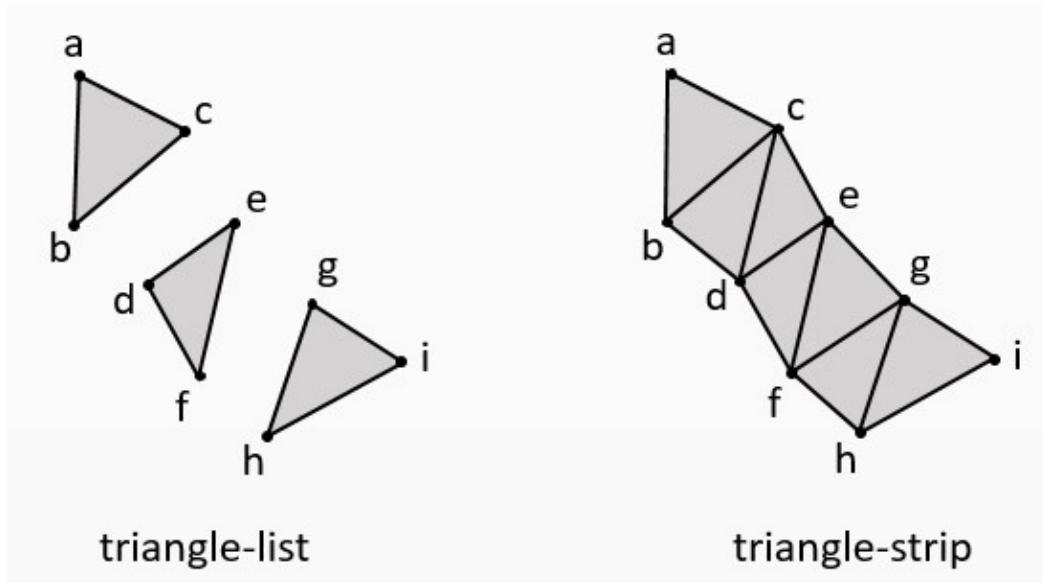


Fig.3-6. Triangles we will create using the triangle-list and triangle-strip primitives.

### 3.3.1 Rust Code

Add a new Rust file called *triangles.rs* to the *examples/ch03/* folder with the following code:

```
#[path = "../ch02/common.rs"]
mod common;

use winit::{
    event_loop::{EventLoop},
};

use std::borrow::Cow;

fn main() {
    let mut primitive_type = "triangle-list";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        primitive_type = &args[1];
    }

    let mut topology = wgpu::PrimitiveTopology::TriangleList;
    let mut index_format = None;
    if primitive_type == "triangle-list" {
        topology = wgpu::PrimitiveTopology::TriangleList;
        index_format = None;
    } else if primitive_type == "triangle-strip" {
        topology = wgpu::PrimitiveTopology::TriangleStrip;
        index_format = Some(wgpu::IndexFormat::Uint32);
    }
}
```

## 48 | Practical GPU Graphics with wgpu and Rust

```
let inputs = common::Inputs{
    source: wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("triangles.wgsl"))),
    topology: topology,
    strip_index_format: index_format
};
let event_loop = EventLoop::new();
let window = winit::window::Window::new(&event_loop).unwrap();

window.set_title(&format!("{}: {}", "Primitive", primitive_type));
env_logger::init();
pollster::block_on( common::run(event_loop, window, inputs, 9));
}
```

This code is similar to that used in the preceding example. Inside the `main` function, we first set the `primitive_type` to “`triangle-list`”. In order to avoid code duplication, we also make the program accept command line arguments as the input value for the `primitive_type` variable. Here, there are two possible command line arguments: “`triangle-list`” and “`triangle-strip`”. The default setting is “`triangle-list`”.

### 3.3.2 Shader Code

Here, we need to add a new shader file called `triangles.wgsl` to the `/examples/ch03` folder. Here is the code for this new shader:

```
struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_color : vec4<f32>;
};

[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> Output {
    var pos : array<vec2<f32>, 9> = array<vec2<f32>, 9>(
        vec2<f32>(-0.63, 0.80),
        vec2<f32>(-0.65, 0.20),
        vec2<f32>(-0.20, 0.60),
        vec2<f32>(-0.37, -0.07),
        vec2<f32>( 0.05, 0.18),
        vec2<f32>(-0.13, -0.40),
        vec2<f32>( 0.30, -0.13),
        vec2<f32>( 0.13, -0.64),
        vec2<f32>( 0.70, -0.30)
    );

    var color : array<vec3<f32>, 9> = array<vec3<f32>, 9>(
        vec3<f32>(1.0, 0.0, 0.0),
        vec3<f32>(0.0, 1.0, 0.0),
        vec3<f32>(0.0, 0.0, 1.0),
        vec3<f32>(1.0, 0.0, 0.0),
        vec3<f32>(0.0, 1.0, 0.0),
        vec3<f32>(0.0, 0.0, 1.0),
        vec3<f32>(1.0, 0.0, 0.0),
        vec3<f32>(0.0, 1.0, 0.0),
        vec3<f32>(0.0, 0.0, 1.0),
    );

    var output: Output;
    output.position = vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
    output.v_color = vec4<f32>(color[in_vertex_index], 1.0);
}
```

```

        return output;
    }

[[stage(fragment)]]
fn fs_main([[location(0)]] v_color: vec4<f32>) -> [[location(0)]] vec4<f32> {
    return v_color;
}

```

We first define a struct called *Output* that contains two output variables: one is the built-in position, and the other is the *v\_color* that represents the output vertex color. Inside the *vs\_main* function, we define nine vertices using a *vec2<f32>* array, and then define nine vertex colors using a *vec3<f32>* array. These nine colors correspond to nine different vertices respectively. The *vs\_main* function finally returns the *output* variable, which is an *Output* type.

The fragment function *fs\_main* takes the *v\_color* as its input argument and sets its return value to the output vertex color *v\_color* from the vertex shader.

### 3.3.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch03_triangles"
path = "examples/ch03/triangles.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch03_triangles "triangle-list"
```

Here, “*triangle-list*” is the input argument for our *primitive\_type* variable. This produces three triangles, as shown in Fig.3-7.

Now, execute the following *cargo run* command in the terminal window:

```
cargo run --example ch03_triangles "triangle-strip"
```

Here, “*triangle-strip*” is the input argument for our *primitive\_type* variable. This produces the results shown in Fig.3-8.

In this chapter, I demonstrated how to create all five built-in primitives in *wgpu*. In fact, any complex 3D shape can be constructed using these primitives. In particular, the *triangle-list* is the preferred primitive for 3D object composition since most GPUs specifically accelerate triangles. The *triangle-list* primitive can be used in real-time computer graphics because it is planar and can be rasterized very quickly in hardware. Several triangles can form a mesh, and meshes can be used to represent just about anything in a 3D scene.

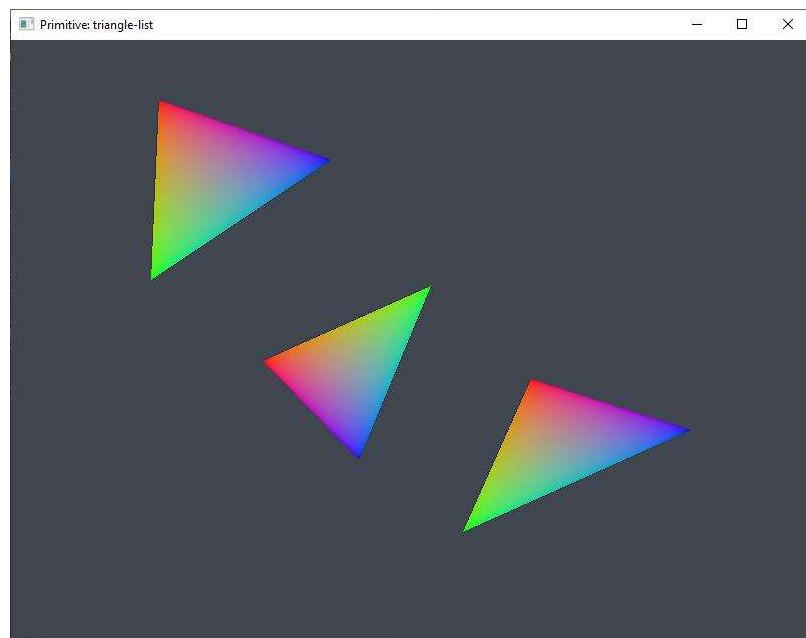


Fig.3-7. Triangles created using the triangle-list primitive.

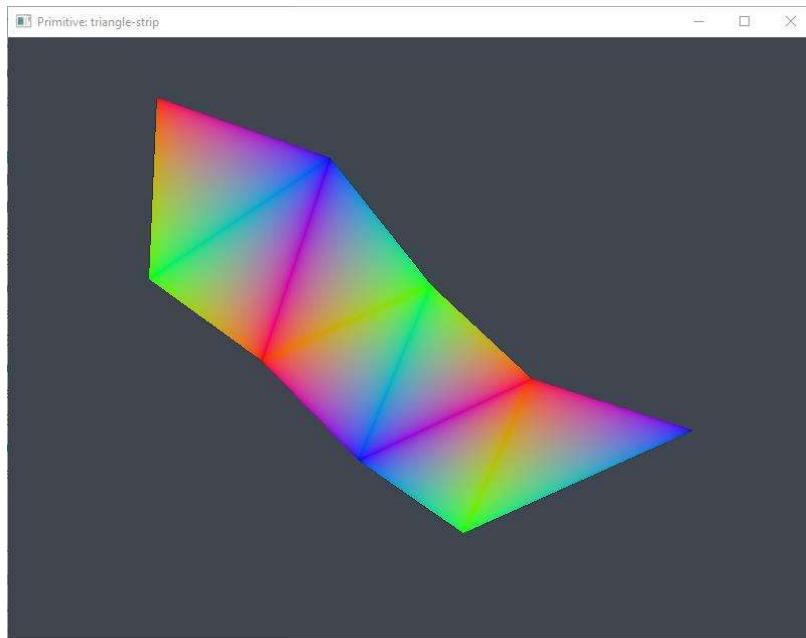


Fig.3-8. Triangles created using the triangle-strip primitive.

# 4 GPU Buffers

In the preceding chapter, we wrote the vertex data directly in the vertex shader when we created the primitive shapes. This is only possible for simple primitive shapes. We usually do not use this direct approach for complex 3D objects with different colors and textures because recompiling the shader code whenever we need to update the model would slow down our program. In this chapter, I will introduce GPU buffers, which hold vertex data and color information, and explain how to use GPU buffers to create colorful triangles and squares with each vertex having a distinct color.

## 4.1 GPU Buffer

A GPU buffer in *wgpu* represents a block of memory used for storing arbitrary data that can be read by the graphics card. It can be used to store vertex data, but it can also be used for many other purposes such as storing global transformation data. Unlike *wgpu* objects, GPU buffers do not automatically allocate memory for themselves. The *wgpu* API requires you to control the memory management yourself.

A GPU buffer in *wgpu* is guaranteed to be contiguous, meaning that all the data is stored sequentially in memory. This indicates that each byte of the allocation can be addressed by its offset from the start of the GPU buffer, subject to alignment restrictions depending on the operation. Some GPU buffers can be mapped so that the block of memory is accessible via an array buffer.

In *wgpu*, we usually use the *device.create\_buffer\_init* function to create a GPU buffer with data to initialize it. This function is a utility function that may not belong to the main *wgpu* API. To access it in *wgpu::device*, we need to import the *DeviceExt* extension trait:

```
use wgpu::util::DeviceExt;
```

If we want to use a GPU buffer to create a vertex buffer to hold the vertex data for our triangle with vertices at [0.0, 0.5], [-0.5, -0.5], and [0.5, -0.5], as well as the color data at each vertex: [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], and [0.0, 0.0, 1.0], we first need to define a *Vertex* struct:

```
#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 2],
    color: [f32; 3],
}
```

Note that we use an alternative representation *repr(C)*, which simply tells our struct to do exactly what C would do, down to the order, size and alignment of fields. Here, we use it for the representation of our

## 52 | Practical GPU Graphics with wgpu and Rust

user-defined *struct* to specify the layout of the type. Since the *create\_buffer\_init* function expects a `&[u8]` type, we will need to use the *bytemuck* crate, which provides functions that let us cast between plain data types. In our case, we will use the *bytemuck::cast\_slice*.

To install the *bytemuck* crate, we need to add the following line to the `[dependencies]` section of our *Cargo.toml* file:

```
bytemuck = { version = "1.4", features = ["derive"] }
```

Here we also need to implement two traits to get *bytemuck* to work: *bytemuck::Pod* and *bytemuck::Zeroable*. *Pod* means that our *Vertex* is “plain old data”, and thus can be interpreted as a `&[u8]` type. *Zeroable* indicates that we can use the `std::mem::zeroed()` function.

Next, we need to generate the vertex and color data for our triangle with the following code snippet:

```
const VERTICES: &[Vertex] = &[
    Vertex { // vertex a
        position: [0.0, 0.5],
        color:[1.0, 0.0, 0.0]
    },
    Vertex { // vertex b
        position: [-0.5, -0.5],
        color: [0.0, 1.0, 0.0]
    },
    Vertex { // vertex c
        position: [0.5, -0.5],
        color:[0.0, 0.0, 1.0]
    },
];
```

Here, we arrange the vertices in counterclockwise order. Now that we have our vertex data, we need to store it in a GPU buffer with the following code:

```
let vertex_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(VERTICES),
    usage: wgpu::BufferUsages::VERTEX,
});
```

You can see that the input argument of the *create\_buffer\_init* function is the *BufferInitDescriptor*, which describes a GPU buffer when allocating. *BufferInitDescriptor* contains three fields:

- *label: Label<'a>* type. This is simply a debug label for the GPU buffer.
- *contents: &'a [u8]* type. This describes the initial contents of the GPU buffer. Here, we use the *bytemuck::cast\_slice* function to convert our vertex data into a `&'a [u8]` type.
- *usage: BufferUsages* type. This constrains the usages of the GPU buffer. Here, we set it to *VERTEX*.

## 4.2 Creating a Colored Triangle

In this section, I will show you how to use GPU buffers to create a colored triangle: each vertex has a different color. In the preceding chapter, we created vertices and colors directly within the vertex and fragment shaders using WGSL.

## 4.2.1 Rust Code

Add a new sub-folder called `ch04` to the `examples/` folder, and then add a new Rust file called `triangle.rs` to this newly created folder with the following content:

```
use std:: {iter, mem};
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 2],
    color: [f32; 3],
}

const VERTICES: &[Vertex] = &[
    Vertex { // vertex a
        position: [0.0, 0.5],
        color:[1.0, 0.0, 0.0]
    },
    Vertex { // vertex b
        position: [-0.5, -0.5],
        color: [0.0, 1.0, 0.0]
    },
    Vertex { // vertex c
        position: [0.5, -0.5],
        color:[0.0, 0.0, 1.0]
    },
];

impl Vertex {
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &[
                wgpu::VertexAttribute {
                    offset: 0,
                    shader_location: 0,
                    format: wgpu::VertexFormat::Float32x2,
                },
                wgpu::VertexAttribute {
                    offset: mem::size_of::<[f32; 2]>() as wgpu::BufferAddress,
                    shader_location: 1,
                    format: wgpu::VertexFormat::Float32x3,
                },
            ],
        }
    }
}

struct State {
    surface: wgpu::Surface,
    device: wgpu::Device,
```

## 54 | Practical GPU Graphics with wgpu and Rust

```
queue: wgpu::Queue,
config: wgpu::SurfaceConfiguration,
size: winit::dpi::PhysicalSize<u32>,
pipeline: wgpu::RenderPipeline,
vertex_buffer: wgpu::Buffer,
}

impl State {
    async fn new(window: &Window) -> Self {
        let size = window.inner_size();
        let instance = wgpu::Instance::new(wgpu::Backends::all());
        let surface = unsafe { instance.create_surface(window) };
        let adapter = instance
            .request_adapter(&wgpu::RequestAdapterOptions {
                power_preference: wgpu::PowerPreference::default(),
                compatible_surface: Some(&surface),
                force_fallback_adapter: false,
            })
            .await
            .unwrap();

        let (device, queue) = adapter
            .request_device(
                &wgpu::DeviceDescriptor {
                    label: None,
                    features: wgpu::Features::empty(),
                    limits: wgpu::Limits::default(),
                },
                None, // Trace path
            )
            .await
            .unwrap();

        let config = wgpu::SurfaceConfiguration {
            usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
            format: surface.get_preferred_format(&adapter).unwrap(),
            width: size.width,
            height: size.height,
            present_mode: wgpu::PresentMode::Fifo,
        };
        surface.configure(&device, &config);

        let shader = device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("shader.wgsl").into()),
        });

        let pipeline_layout =
            device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
                label: Some("Render Pipeline Layout"),
                bind_group_layouts: &[],
                push_constant_ranges: &[],
            });

        let pipeline = device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
            label: Some("Render Pipeline"),
            layout: Some(&pipeline_layout),
            vertex: wgpu::VertexState {
                module: &shader,
                entry_point: "vs_main",
            }
        });
    }
}
```

```

        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format:None,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
```

```

let vertex_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(VERTICES),
    usage: wgpu::BufferUsages::VERTEX,
});
```

```

Self {
    surface,
    device,
    queue,
    config,
    size,
    pipeline,
    vertex_buffer,
}
```

```

}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.size = new_size;
        self.config.width = new_size.width;
        self.config.height = new_size.height;
        self.surface.configure(&self.device, &self.config);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.surface.get_current_texture()?;
    let view = output
        .texture

```

## 56 | Practical GPU Graphics with wgpu and Rust

```
.create_view(&wgpu::TextureViewDescriptor::default());  
  
let mut encoder = self  
.device  
.create_command_encoder(&wgpu::CommandEncoderDescriptor {  
    label: Some("Render Encoder"),  
});  
  
{  
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {  
        label: Some("Render Pass"),  
        color_attachments: &[wgpu::RenderPassColorAttachment {  
            view: &view,  
            resolve_target: None,  
            ops: wgpu::Operations {  
                load: wgpu::LoadOp::Clear(wgpu::Color {  
                    r: 0.2,  
                    g: 0.247,  
                    b: 0.314,  
                    a: 1.0,  
                }),  
                store: true,  
            },  
        }],  
        depth_stencil_attachment: None,  
    });  
  
    render_pass.set_pipeline(&self.render_pipeline);  
    render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));  
    render_pass.draw(0..3, 0..1);  
}  
  
self.queue.submit(iter::once(encoder.finish()));  
output.present();  
Ok()  
}  
}  
  
fn main() {  
    env_logger::init();  
    let event_loop = EventLoop::new();  
    let window = WindowBuilder::new().build(&event_loop).unwrap();  
    window.set_title(&format!("{}", "ch04-triangle"));  
    let mut state = pollster::block_on(State::new(&window));  
  
    event_loop.run(move |event, _, control_flow| {  
        match event {  
            Event::WindowEvent {  
                ref event,  
                window_id,  
            } if window_id == window.id() => {  
                if !state.input(event) {  
                    match event {  
                        WindowEvent::CloseRequested  
                        | WindowEvent::KeyboardInput {  
                            input:  
                                KeyboardInput {  
                                    state: ElementState::Pressed,  
                                    virtual_keycode: Some(VirtualKeyCode::Escape),  
                                    ..  
                                }  
                        }  
                    }  
                }  
            }  
        }  
    })  
}
```

```

        },
        ..
    } => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!(":{}:", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

To better organize our code, here we use the code structure suggested by sotrh in his excellent Learn Wgpu tutorial (<https://sotrh.github.io/learn-wgpu/>).

You should already be familiar with the above code listing because most of the code is similar to what we used in the examples from the preceding chapter. The difference is that we now use the GPU buffers to store the vertex data.

Here, we first define the *Vertex* struct. Each vertex contains a position and a color. The position represents the *x* and *y* coordinates in 2D space. The color is the red, green, and blue components for the vertex. We need the *Vertex* struct to be copyable so that we can create the GPU buffer with it.

Next, we set the actual data that will make up our triangle. We then write an implementation block that creates a static method called *desc<'a>()*, which returns *VertexBufferLayout*:

```

impl Vertex {
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &[
                wgpu::VertexAttribute {
                    offset: 0,
                    shader_location: 0,
                    format: wgpu::VertexFormat::Float32x2,
                },
                wgpu::VertexAttribute {
                    offset: mem::size_of::<[f32; 2]>() as wgpu::BufferAddress,
                    shader_location: 1,
                    format: wgpu::VertexFormat::Float32x3,
                },
            ],
        }
    }
}

```

## 58 | Practical GPU Graphics with wgpu and Rust

```
        ],
    }
}
```

We use *VertexBufferLayout* to tell the render pipeline how to read the data from the GPU buffer. A *VertexBufferLayout* defines how a GPU buffer is laid out in memory. Without this layout, the render pipeline cannot map the GPU buffer in the shader code. Within the *VertexBufferLayout*, there are several attributes need to be specified, including:

- *array\_stride*: This defines the stride, in bytes, between vertices. When the shader goes to read the next vertex, it will skip over *array\_stride* number of bytes. In our case, *array\_stride* will be 20 bytes, with 8 bytes for the position ([f32; 2]) and 12 bytes for the color ([f32; 3]).
- *step\_mode*: This tells the render pipeline how often it should move to the next vertex. This seems redundant in our case, but we can set it to *wgpu::VertexStepMode::Instance* if we want to change vertices when we start drawing a new instance. We will discuss instancing later.
- *attributes*: This is a two-element array, which describes the individual parts of the vertex. Each element is a *VertexAttrib* that includes three fields: *offset*, *shader\_location*, and *format*. We set the *offset*, in bytes, to zero in the first element and to the collective *size\_of* of the previous attribute data in the second element. The *shader\_location* is another important attribute that tells the shader at which location to store this attribute. For example, *[[location(0)]] position: vec2<f32>* in the vertex shader would correspond to the position field of the struct, while *[[location(1)]] color: vec3<f32>* would be the color field. The *format* attribute tells the shader the shape of the attribute. Here, the *Float32x2* and *Float32x3* correspond to [f32; 2] and [f32; 3] in the shader code respectively. The maximum value we can store in an attribute is *Float32x4*.

You can see that specifying the *attributes* as we did here is quite verbose. We can use a *wgpu* macro called *vertex\_attr\_array* to simplify this definition. With *vertex\_attr\_array*, we can rewrite the above code snippet in the following:

```
impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttrib; 2] = wgpu::vertex_attr_array![0=>Float32x2, 1=>Float32x3];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}
```

Here, the *ATTRIBUTES* constant specifies a list of two *VertexAttrib*, each with the given *shader\_location* and *format*. Offsets are calculated automatically. We will use this simplified version in future examples. While in the current example, we will use the original code because it shows how the data is mapped.

We then pack all the fields required for drawing our graphics shape into another struct called *State*, and create corresponding methods on it:

```
struct State {
    surface: wgpu::Surface,
    device: wgpu::Device,
    queue: wgpu::Queue,
    config: wgpu::SurfaceConfiguration,
    size: winit::dpi::PhysicalSize<u32>,
    pipeline: wgpu::RenderPipeline,
```

```

        vertex_buffer: wgpu::Buffer,
    }
impl State {
    async fn new(window: &Window) -> Self {
        ... // code omitted for brevity
    }

    fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
        ... // code omitted for brevity
    }

    fn input(&mut self, event: &WindowEvent) -> bool {
        ... // code omitted for brevity
    }

    fn update(&mut self) {
        ... // code omitted for brevity
    }

    fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
        ... // code omitted for brevity
    }
}
}

```

The `State::new()` `async` method consists of code for creating the adapter, device, surface, configuration, shader module, and pipeline layout, which are required to set up the render pipeline. This part of the code is similar to that used in our previous examples, except that within the pipeline, the `vertex` attribute associated with the `Vertex::desc()` function:

```

vertex: wgpu::VertexState {
    module: &shader,
    entry_point: "vs_main",
    buffers: &[Vertex::desc()],
},

```

In addition, the `State::new()` function also includes the `vertex_buffer` created using the `device.create_buffer_init` function. After configuring the surface and pipeline, we then add these new fields at the end of the `State::new()` function:

```

Self {
    surface,
    device,
    queue,
    config,
    size,
    pipeline,
    vertex_buffer,
}

```

We will call the `State::new()` function in our `main` function before entering the event loop:

```
let mut state = pollster::block_on(State::new(&window));
```

To support resizing in our application, we need to configure the surface every time the window's size changes. Inside the `resize` function, we store the physical `size` and the `config` used to configure the surface. If the window's size changes, we use the new `size` to reconfigure the surface.

Next, the `input()` function returns a Boolean result to indicate whether an event has been fully processed. If it returns true, the main loop will not process the event any further. Here, it just returns false because for now there are no input events to capture.

## 60 | Practical GPU Graphics with `wgpu` and Rust

The `update()` function is empty in our current example because there is nothing to update yet. We will add some code here later on when we want to move some objects around.

The `render()` function defines the render pass, sets the pipeline and vertex buffer to the render pass, and calls the draw function to create our triangle:

```
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.draw(0..3, 0..1);
```

Here, `set_vertex_buffer` takes two parameters: the first is the buffer slot and the second is the slice of buffer to use. You can store as many objects in a buffer as your hardware allows, so `slice` allows you to specify which portion of the buffer to use. Here, we use `..` to indicate the entire buffer.

Inside the `main` function, we need to update the event loop to call the `render()` function.

### 4.2.2 Shader Code

Before our changes can have any effect, we need to update our vertex shader to get its data from the vertex buffer. Add a new `triangle.wgsl` file to the `examples/ch04/` folder with the following content:

```
struct VertexInput {
    [[location(0)]] pos: vec2<f32>;
    [[location(1)]] color: vec3<f32>;
};

struct VertexOutput {
    [[builtin(position)]] position: vec4<f32>;
    [[location(0)]] color: vec4<f32>;
};

[[stage(vertex)]]
fn vs_main(in: VertexInput) -> VertexOutput {
    var out: VertexOutput;
    out.color = vec4<f32>(in.color, 1.0);
    out.position = vec4<f32>(in.pos, 0.0, 1.0);
    return out;
}

[[stage(fragment)]]
fn fs_main(in: VertexOutput) -> [[location(0)]] vec4<f32> {
    return in.color;
}
```

Note that every `shader_location` in the Rust code must be unique in order to generate the correct shaders in WGLS. Here, `[[location(0)]]` corresponds to `shader_location = 0` while `[[location(1)]]` corresponds to `shader_location = 1`. This way, the shaders can use the GPU buffer data for vertex and color information.

### 4.2.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch04_triangle"
path = "examples/ch04/triangle.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch04_triangle
```

This produces a colorful triangle, as shown in Fig.4-1.

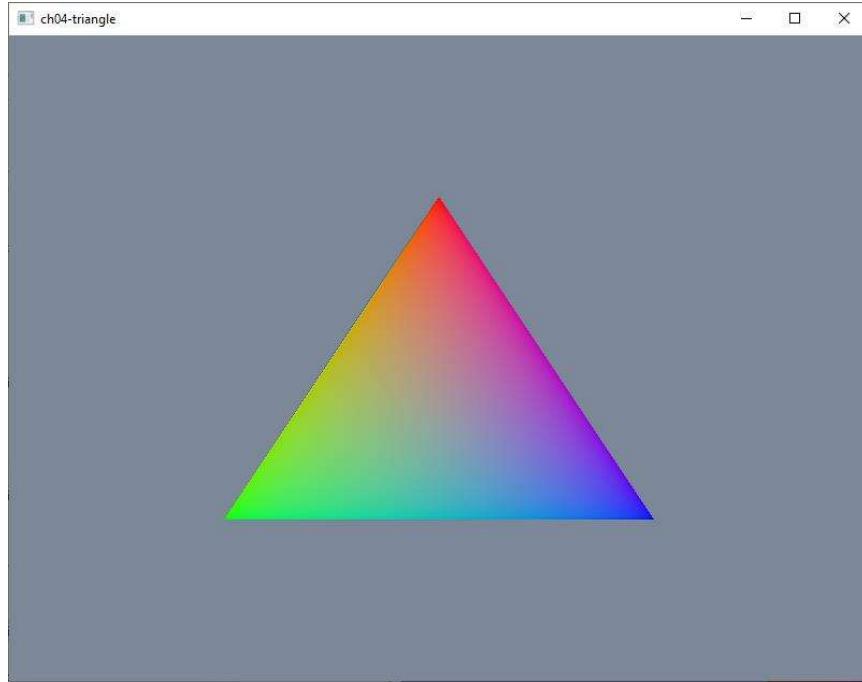


Fig.4-1. Colored triangle created using GPU buffers.

### 4.3 Creating a Colored Square

In this section, we will create a colored square using the *triangle-list* primitive. We can divide a square  $\square abcd$  diagonally into two triangles:  $\Delta abd$  and  $\Delta dbc$ , as shown in Fig.4-2.

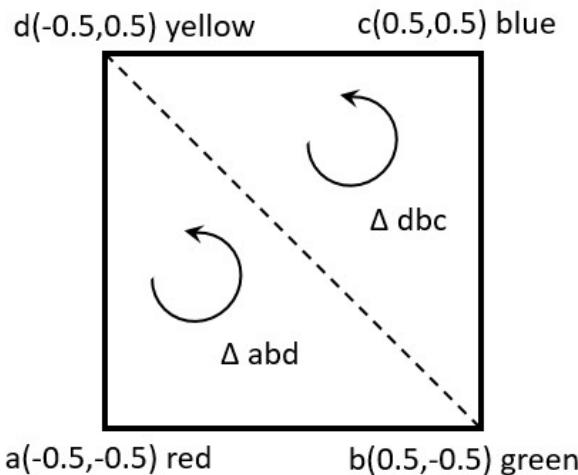


Fig.4-2. Divide a square shape into two triangles.

## 62 | Practical GPU Graphics with `wgpu` and Rust

The vertices of each triangle must be arranged in the counterclockwise order. In the figure, the vertex positions and corresponding color for each vertex are also specified.

### 4.3.1 Rust Code

Add a new Rust file called `square.rs` to the `examples/ch04/` folder. For the most part, the code for this file is identical to that used in the preceding example, except that we need to replace the vertex data for the triangle with the vertex data for the square:

```
const VERTICES: &[Vertex] = &[
    Vertex { // vertex a
        position: [-0.5, -0.5],
        color:[1.0, 0.0, 0.0]
    },
    Vertex { // vertex b
        position: [0.5, -0.5],
        color: [0.0, 1.0, 0.0]
    },
    Vertex { // vertex d
        position: [-0.5, 0.5],
        color:[1.0, 1.0, 0.0]
    },
    Vertex { // vertex d
        position: [-0.5, 0.5],
        color:[1.0, 1.0, 0.0]
    },
    Vertex { // vertex b
        position: [0.5, -0.5],
        color: [0.0, 1.0, 0.0]
    },
    Vertex { // vertex c
        position: [0.5, 0.5],
        color: [0.0, 0.0, 1.0]
    },
];
```

We also need to change the number of vertices from 3 to 6 in the `draw` function because our square consists of two triangles:

```
render_pass.draw(0..6, 0..1);
```

The shader code is the same as that used in the preceding example, so we use the same shader file, `triangle.wgsl`:

```
let shader = device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!("triangle.wgsl").into()),
});
```

We do not need to make any changes to the rest of the code because it is the same as that used in the preceding example.

### 4.3.2 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

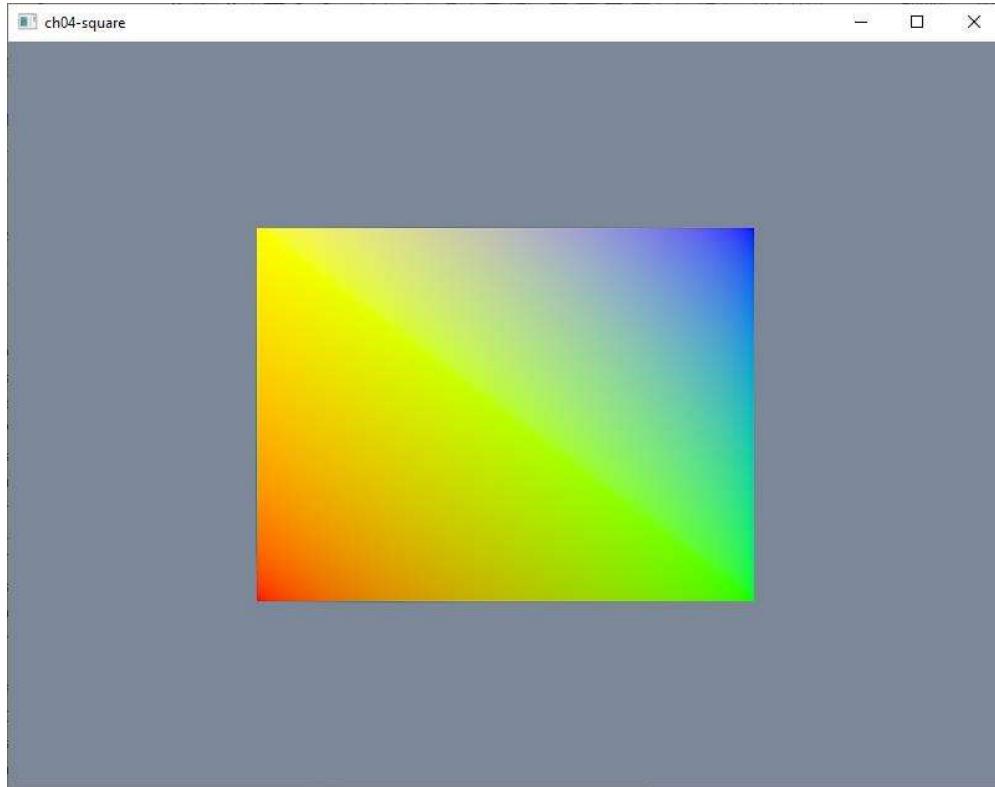
```
[[example]]
```

```
name = "ch04_square"
path = "examples/ch04/square.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch04_square
```

This produces a colorful square, as shown in Fig.4-3.



*Fig.4-3. Colorful square created using GPU buffers.*

## 4.4 Creating a Square with an Index Buffer

In the preceding example, we created the square with two triangles, three vertices each, for a total of six vertices. However, our square only has four unique vertices, meaning that we ended up duplicating two vertices. For complicated 3D shapes, this duplication of vertex data may significantly increase our CPU/GPU memory consumption.

To avoid such duplication, *wgpu* implements a function called *draw\_indexed* that allows us to draw graphics with an index GPU buffer.

### 4.4.1 Rust Code

Add a new Rust file called *square\_index.rs* to the *examples/ch04/* folder. Most of the code for this file is identical to that used in the preceding example, except that we need to replace the vertex data with indexed vertex data for the square:

## 64 | Practical GPU Graphics with wgpu and Rust

```
const VERTICES: &[Vertex] = &[
    Vertex { // vertex a, index = 0
        position: [-0.5, -0.5],
        color:[1.0, 0.0, 0.0]
    },
    Vertex { // vertex b, index = 1
        position: [0.5, -0.5],
        color: [0.0, 1.0, 0.0]
    },
    Vertex { // vertex c, index = 2
        position: [0.5, 0.5],
        color: [0.0, 0.0, 1.0]
    },
    Vertex { // vertex d, index = 3
        position: [-0.5, 0.5],
        color:[1.0, 1.0, 0.0]
    }
];
const INDICES: &[u16] = &[0, 1, 3, 3, 1, 2];
```

Here, we just store all of the unique vertices in *VERTICES* and then create another buffer to store indices to the elements in *VERTICES* to create the two triangles.

Originally, we needed six vertices to create two triangles, which needed 120 bytes of memory to store their data. Now, with the index buffer setup, our *VERTICES* take up 80 bytes (for four vertices) and *INDICES* is only 12 bytes given that *u16* is 2 bytes wide. Altogether, our square with the index buffer requires 92 bytes in total. This means we saved 28 bytes. For our simple square example, this is not much of a savings of memory space. However, for a complex scene, we would need to repeatedly use the same vertex, meaning we would have to rewrite the same vertex coordinates many times if we do not use a vertex index. By using a vertex index, we only need to write each vertex coordinate once, and *wgpu* will find the specific vertices to create the corresponding triangles by calling the *render\_pass.draw\_indexed* method.

There are a couple of things, which we need to change in order to use indexing. The first thing we need to do is create an index buffer to store the indices. Within the *State::new()* method, we can create the *index\_buffer* after creating the *vertex\_buffer*:

```
let vertex_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: bytemuck::cast_slice(VERTICES),
    usage: wgpu::BufferUsages::VERTEX,
});
let index_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Index Buffer"),
    contents: bytemuck::cast_slice(INDICES),
    usage: wgpu::BufferUsages::INDEX,
});
let indices_len = INDICES.len() as u32;
```

Here, we do not need to implement *Pod* and *Zeroable* for our indices because *bytemuck* has already implemented them for basic types such as *u16*. This means we can simply add *index\_buffer* and *indices\_len* to the *State* struct:

```
Self {  
    surface,  
    device,  
    queue,
```

```

    config,
    size,
    pipeline,
    vertex_buffer,
    index_buffer,
    indices_len
}

```

Finally, we need to update the `render()` method to use the `index_buffer`:

```

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_index_buffer(self.index_buffer.slice(..), wgpu::IndexFormat::Uint16);
render_pass.draw_indexed(0..self.indices_len, 0, 0..1);

```

Note that when we use the index buffer, we need to use the `draw_indexed` function to create our square. The original `draw` method ignores the index buffer. Also, we must make sure we use the number of indices (`indices_len`) instead of the number of vertices when we call the `draw_indexed` method.

## 4.4.2 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```

[[example]]
name = "ch04_square_index"
path = "examples/ch04/square_index.rs"

```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch04_square_index
```

This produces a colorful square, as shown in Fig.4-4.

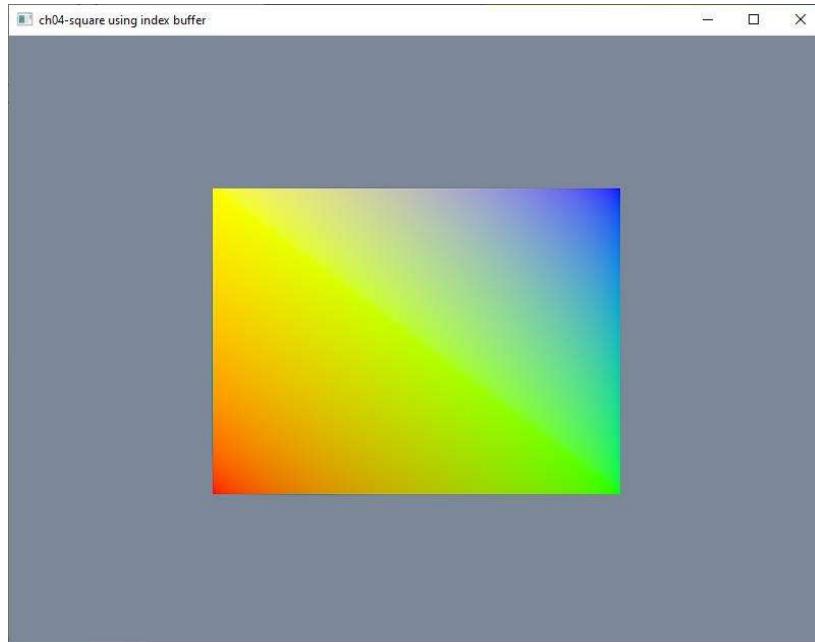


Fig.4-4. Square created using an index GPU buffer.



# 5 3D Transformations

In the preceding chapters, we discussed the basics of *wgpu* and explained how to set up the development environment and tools used to build *wgpu* applications. We also presented a step-by-step procedure for creating built-in primitives. However, in order to create real 3D objects in *wgpu* applications, you will need to learn the mathematical basics of 3D matrices and transformations.

Transformations in 3D are analogous to those in 2D, but the increase in complexity that comes with the third dimension is substantial. In this chapter, I will show you how to perform basic 3D transformations, including translation, scaling, and rotation. I will also describe how to construct matrices that represent viewing and projection, allowing you to view 3D graphics objects on a 2D screen.

## 5.1 Basics of 3D Matrices and Transformations

Matrix representations play an important role in transformations and operations on graphics objects. A matrix is a multi-dimensional array. This section explains the basics of 3D matrices and transformations. General 3D transformations are quite complicated, but as is the case of 2D, you can build more useful transformations with combinations of simple basic transformations, including translation, scaling, rotation, and projection. The following sections describe these fundamental transformations. Once you understand how to construct these basic 3D transformations, you can always compose them to create general transformations.

Like WebGL and WebGPU, the *wgpu* API does not provide any functions for working with transformations. In WebGL and WebGPU, we usually use the JavaScript package *glMatrix* to perform 3D transformation and matrix operations. In Rust, the *cgmath* crate is a more popular linear algebra and mathematics library for computer graphics. In this book, I will use the *cgmath* library to create 3D matrix operations and transformations in *wgpu* applications. Of course, you are free to choose any of the other Rust crates used in computer graphics, such as *gl\_matrix*, *nalgebra*, etc.

### 5.1.1 Introducing *cgmath*

The *cgmath* crate provides various structs such as *Vector1*, *Vector2*, *Vector3*, and *Vector4* for working with vectors of 1, 2, 3, and 4 numbers. It also defines *Matrix2*, *Matrix3*, and *Matrix4* for working with 2-by-2, 3-by-3, and 4-by-4 matrices, respectively. Although *cgmath* is a Rust crate, a *Matrix4* matrix in *cgmath* can be directly passed to the shader in *wgpu* to specify the value of a WGSL *mat4*, and the same is true for the other vector and matrix types.

## 5.1.2 3D Vector and Matrix Operations

You can perform 3D matrix operations in a homogeneous coordinate system. Here, a point or vector is represented by a column matrix. In this notation, most 3D transformation matrices contain  $(0, 0, 0, 1)$  in their last row. You can use this special structure to speed up matrix operations. This approach works for most transformation matrices, except for the perspective transformation described later in this chapter, which does not contain  $(0, 0, 0, 1)$  in its last row. Therefore, in this book, we will not assume that transformation matrices contain  $(0, 0, 0, 1)$  in their last row.

The *Vector4* struct defined in *cgmath* represents a vector with four numbers. You can create a new *Vector4* initialized with four given values using the following command:

```
let my_vec = Vector4::new(4, 3, 2, 1);
```

The *Vector4* struct in *cgmath* consists of many functions that allow you to manipulate vectors.

The matrix format in *cgmath* is column major. The matrix element  $cirj$  represents the component in the column  $i$  and row  $j$  position; that is, the first index always denotes the column and the second index the row. It can be created from an array of values. For example, a *Matrix2* matrix can be created using the following statement:

```
Matrix2::new(c0r0, c0r1, c1r0, c1r1);
```

Here, the first two numbers  $c0r0, c0r1$  represent the first column and  $c1r0, c1r1$  represent the second column of the matrix. The following command creates a 4-by-4 identity matrix:

```
let identity_mat:Matrix4<f32> = Matrix4::identity();
```

We usually use the *Vector4* and *Matrix4* structs to perform matrix operations and transformations in the 3D homogeneous coordinate system ( $x, y, z, w$ ). The  $w$ -dimension is called the projective space and the coordinates in the projective space are called homogeneous coordinates. For the purpose of 3D software, the terms “projective” and “homogeneous” are *basically* interchangeable with “4D”.

The *Matrix4* contains sixteen numbers representing sixteen elements, including  $c0r0, c0r1, c0r2, c0r3$  (first column),  $c1r0, c1r1, c1r2, c1r3$  (second column),  $c2r0, c2r1, c2r2, c2r3$  (third column), and  $c3r0, c3r1, c3r2, c3r3$  (fourth column).

Each vector or matrix in *cgmath* has a function called *clone()* that creates a copy of itself. For example, if we create a matrix called *my\_mat*, we can make a clone of it by using the command:

```
let clone_mat = my_mat.clone();
```

We can also create a new matrix using vectors as columns:

```
let v0 = Vector4::new(1, 0, 0, 1);
let v1 = Vector4::new(0, 1, 0, 1);
let v2 = Vector4::new(0, 0, 1, 1);
let v3 = Vector4::new(0, 0, 0, 1);
let m1 = Matrix4::from_cols(v0, v1, v2, v3);
```

There are functions in *cgmath* for performing standard transformations such as scaling, translation, and rotation. For example, we can use the following functions to construct a scale matrix and a translation matrix:

```
let scale_mat = Matrix4::from_scale(0.5)
let trans_mat = Matrix4::from_translation(Vector3::new(1.0, 0.5, 1.5));
```

*Matrix4*, *Vector4*, and *Vector3* also contain various functions and operations that allow you to perform matrix operations and transformations in 3D space.

### 5.1.3 Scaling

To scale or stretch an object in the  $x$  direction, you need to multiply the  $x$  coordinates of each of the object's points by the scaling factor,  $s_x$ . In the same way, you can scale an object in the  $y$  and  $z$  directions. In the standard 3D Cartesian coordinate system, a scaling transformation can be represented in the form:

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \end{pmatrix} \quad (5.1)$$

The 3D scaling transformation can be generalized in the homogeneous coordinate system:

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad (5.2)$$

For example, suppose that you have a 3D point  $(1, 2, 3, 1)$  in homogeneous coordinates, and you want to apply a scaling matrix to this point. This scaling matrix shrinks  $x$  and  $y$  uniformly by a factor of two, and stretches  $z$  by a factor of three-halves. This scaling operation can easily be computed using Equation (5.2):

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1.5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 1 \\ 4.5 \\ 1 \end{pmatrix} \quad (5.3)$$

You can also perform two successive scaling transformations. For example, say the first scaling matrix is the same as that used in the above example, while the second scaling matrix has scaling factors of  $s_x = 1$ ,  $s_y = 0.5$ , and  $s_z = 0.3$ . We can calculate the two successive scaling transformations using matrix multiplication. The final scaled vector after these two scaling operations will be  $(0.5, 0.5, 1.35, 1)$ .

In the following, I will show you how to perform 3D transformations using *cgmath*. Before using the *cgmath* crate, make sure you add it to the `[dependencies]` section in the *Cargo.toml* file:

```
cgmath = "0.18"
```

For scaling, *cgmath* has two functions that allow you to construct a scaling transformation matrix with the scale factors  $s$ , or  $s_x$ ,  $s_y$ , and  $s_z$ :

```
Matrix4::from_scale(s)
Matrix4::from_nonuniform_scale(sx, sy, sz).
```

The first function creates a uniform scaling along the  $x$ ,  $y$ , and  $z$  directions, while the second function allows you to create different scaling for different directions.

In the following example, I will show you how to perform scaling transformations. Add a new sub-folder called *ch05* to the *examples/* folder, and then add a Rust file named *scaling.rs* to this newly created folder. Enter the following code into the *scaling.rs* file:

```
use cgmath::{Matrix4, Vector4};

fn main() {
    //create original vector
```

## 70 | Practical GPU Graphics with wgpu and Rust

```
let my_vec = Vector4::new(1.0, 2.0, 3.0, 1.0);

// create scale matrix
let my_mat = Matrix4::from_nonuniform_scale(0.5, 0.5, 1.5);

// get the scaled vector
let scaled_vec = my_mat * my_vec;

println!("Original vector: \n{:?}", my_vec);
println!("Scaling matrix: \n{:?}", my_mat);
println!("Vector after scaling: scaled_vec = my_mat * my_vec = \n{:?}", scaled_vec);

// two successive scaling transforms:
// get total scaling matrix after another scaling transformation:
let scaling_mat = my_mat * Matrix4::from_nonuniform_scale(1.0, 0.5, 0.3);

// get final scaled vector
let final_vec = scaling_mat * my_vec;

println!("Scaling matrix after two scalings: \n{:?}", scaling_mat);
println!("Vector after two scalings: scaled_vec = scaling_mat * my_vec = \n{:?\n", final_vec);
}
```

Here, we first create our original vector (1.0, 2.0, 3.0, 1.0) with the *Vector4<f64>* type, and then create a scaling matrix by calling the *Matrix4::from\_nonuniform\_scale* method. Finally, we perform a scaling operation by applying the scaling transformation by a direct multiplication *my\_mat \* my\_vec*.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch05_scaling"
path = "examples/ch05/scaling.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch05_scaling
```

This produces results in the terminal window, as shown in Fig.5-1.

```
Original vector:
Vector4 [1.0, 2.0, 3.0, 1.0]
Scaling matrix:
Matrix4 [[0.5, 0.0, 0.0, 0.0], [0.0, 0.5, 0.0, 0.0], [0.0, 0.0, 1.5, 0.0],
[0.0, 0.0, 0.0, 1.0]]
Vector after scaling: scaled_vec = my_mat * my_vec =
Vector4 [0.5, 1.0, 4.5, 1.0]

Scaling matrix after two scalings:
Matrix4 [[0.5, 0.0, 0.0, 0.0], [0.0, 0.25, 0.0, 0.0], [0.0, 0.0, 0.4499999999999996, 0.0],
[0.0, 0.0, 0.0, 1.0]]
Vector after two scalings: scaled_vec = scaling_mat * my_vec =
Vector4 [0.5, 0.5, 1.3499999999999999, 1.0]
```

Fig.5-1. 3D scaling transformations.

You can see that the result in Fig.5-1 is consistent with Equation (5.3), as expected.

We can also demonstrate how to perform two successive scaling transformations using *cgmath*, by using the following command to calculate the total scaling matrix:

```
let scaling_mat = my_mat * Matrix4::from_nonuniform_scale(1.0, 0.5, 0.3);
```

Here, *my\_mat* is the first scaling matrix and *scaling\_mat* is the total scaling matrix after another scaling transformation with *sx* = 1.0, *sy* = 0.5, and *sz* = 0.3.

## 5.1.4 Translation

To translate a point by a distance of *dx* in the *x*-direction, *dy* in the *y*-direction, and *dz* in the *z*-direction, you simply multiply the point by a transformation matrix in homogeneous coordinates:

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{pmatrix} \quad (5.4)$$

The results of  $x_1 = x + dx$ ,  $y_1 = y + dy$ , and  $z_1 = z + dz$  are indeed the correct translation of a point  $(x, y, z, 1)$ .

For translation, *cgmath* has the following function:

```
Matrix4::from_translation(v);
```

This function creates a translation matrix from a vector *v* translation.

In the following example, I will show you how to perform translations using the above function. Add a new Rust file called *translation.rs* to the *examples/ch05/* folder. Enter the following code into this *translation.rs* file:

```
use cgmath::*;

fn main() {
    // create original vector
    let my_vec = Vector4::new(1.0, 2.0, 3.0, 1.0);

    // create first translation matrix
    let my_mat = Matrix4::from_translation(Vector3::new(2.0, 2.5, 3.0));

    // get total translation matrix after another translation
    let trans_mat = my_mat * Matrix4::from_translation(Vector3::new(-3.0, -2.0, -1.0));

    // get final translated vector
    let trans_vec = trans_mat * my_vec;

    println!("Original vector: my_vec = {:?}", my_vec);
    println!(
        "Total translation matrix after two translations: trans_mat: \n{:?}",
        trans_mat
    );
    println!(
        "Vector after two translations: trans_vec = trans_mat * my_vec = \n{:?}\n",
        trans_vec
    );
}
```

Now, add the following code snippet to the *Cargo.toml* file:

## 72 | Practical GPU Graphics with wgpu and Rust

```
[[example]]
name = "ch05_scaling"
path = "examples/ch05/scaling.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch05_translation
```

This produces results in the terminal window, as shown in Fig.5-2.

```
Original vector: my_vec =
Vector4 [1.0, 2.0, 3.0, 1.0]
Total translation matrix after two translations: trans_mat:
Matrix4 [[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0],
[-1.0, 0.5, 2.0, 1.0]]
Vector after two translations: trans_vec = trans_mat * my_vec =
Vector4 [0.0, 2.5, 5.0, 1.0]
```

*Fig.5-2. Vector after two successive translations.*

### 5.1.5 Rotation

Rotating an object around an arbitrary axis in 3D is much more complicated than doing so in 2D. However, rotating an object around the  $x$ -,  $y$ -, or  $z$ -axis is still quite simple. For example, to rotate a point around the  $z$ -axis, we can simply ignore the  $z$  coordinate of the point and handle the rotation as if it were taking place in 2D. This is because its coordinate remains unchanged when we rotate the point around the  $z$ -axis. We can easily write down this rotation matrix in homogeneous coordinates:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.5)$$

Similarly, we can represent rotation matrices around the  $y$  and  $x$  directions:

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.6)$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (5.7)$$

The `cgmath` crate has several functions that allow us to perform various rotations:

```
Matrix4::from_axis_angle(axis, rad);
```

```
Matrix4::from_angle_x(rad);
Matrix4::from_angle_y(rad);
Matrix4::from_angle_z(rad);
```

The first function creates a rotation matrix for a given angle (in radians) around a given axis. The last three functions rotate a matrix about the  $x$ -,  $y$ -, or  $z$ -axis.

In the following example, we will create two rotations around the  $z$ -axis: one 20-degree rotation followed by one 25-degree rotation. The total effect of these two successive rotations should be identical to a 45-degree rotation around the  $z$ -axis.

Now, add a new Rust file called *rotation.rs* to the *examples/ch05/* folder. Enter the following code into this file:

```
use std::f32::consts::PI;
use cgmath:: { Matrix4, Vector4, Rad };

fn main() {
    // create original vector
    let my_vec = Vector4::new(1.0, 2.0, 3.0, 1.0);

    // create a rotation matrix around the z axis by 20 degrees:
    let rot_mat_z = Matrix4::from_angle_z(Rad( 20.0 * PI / 180.0));

    // get total rotation matrix after another rotation around the z axis by 25 degrees
    let rot_mat = rot_mat_z * Matrix4::from_angle_z(Rad( 25.0 * PI / 180.0));

    // get final rotated vector
    let rot_vec = rot_mat * my_vec;

    println!("Original vector: my_vec = \n{:?}", my_vec);
    println!(
        "Total rotation matrix after two rotations: rot_mat = \n{:?}",
        rot_mat
    );
    println!(
        "Vector after two rotations: rot_vec = rot_mat * my_vec = \n{:?\n}",
        rot_vec
    );
}
```

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch05_rotation"
path = "examples/ch05/rotation.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch05_rotation
```

This produces results in the terminal window, as shown in Fig.5-3.

```

Original vector: my_vec =
Vector4 [1.0, 2.0, 3.0, 1.0]
Total rotation matrix after two rotations: rot_mat =
Matrix4 [[0.70710677, 0.70710677, 0.0, 0.0], [-0.70710677, 0.70710677,
0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0, 0.0, 0.0, 1.0]]
Vector after two rotations: rot_vec = rot_mat * my_vec =
Vector4 [-0.70710677, 2.1213202, 3.0, 1.0]

```

*Fig.5-3. Vector after two successive rotations.*

You can see that we end up with a final vector (-0.707, 2.121, 3, 1). This result can easily be confirmed using a direct matrix multiplication approach [see Equation (5.5)].

### 5.1.6 Combining Transformations

In the preceding sections, we discussed how to perform individual transformation in 3D space. Now, what if we want to move, rotate, and scale our 3D objects? To do this, we need to combine the individual transformation matrices into a single *Matrix4* matrix using matrix multiplication. Note that when combining transformation matrices, we must multiply them in the reverse order of how we want them to be applied. This is critical; we will get completely wrong transformations otherwise. For example, if we want to scale our object, then rotate it and finally translate it, we must multiply the translation matrix by the rotation matrix by the scaling matrix, in that order.

Suppose we want to construct a combined transformation matrix by performing transformations in the following order: translation first, then rotation, and finally scaling. This can be achieved by adding a new Rust file called *transforms.rs* to the *examples/common/* folder with the following code:

```

#![allow(dead_code)]
use cgmath::{ Matrix4, Vector3, Rad};

pub fn create_transforms(translation:[f32; 3], rotation:[f32; 3], scaling:[f32; 3]) -> Matrix4<f32> {

    // create individual transformation matrices
    let trans_mat = Matrix4::from_translation(Vector3::new(translation[0],
        translation[1], translation[2]));
    let rotate_mat_x = Matrix4::from_angle_x(Rad(rotation[0]));
    let rotate_mat_y = Matrix4::from_angle_y(Rad(rotation[1]));
    let rotate_mat_z = Matrix4::from_angle_z(Rad(rotation[2]));
    let scale_mat = Matrix4::from_nonuniform_scale(scaling[0], scaling[1], scaling[2]);

    // combine all transformation matrices together to form a final transform matrix: model matrix
    let model_mat = trans_mat * rotate_mat_z * rotate_mat_y * rotate_mat_x * scale_mat;

    // return final model matrix
    model_mat
}

```

Here, the *rotation* input parameter is a vector  $[f32, 3]$  type with its components representing the rotation (in radians) about the *x*-, *y*-, and *z*-axes, respectively.

In addition, we also add a crate-level attribute `#![allow(dead_code)]`. The Rust compiler provides a *dead\_code* lint that will warn about unused code. This attribute will disable the *dead\_code* lint. But in

real applications, you should eliminate dead code. In our examples, we will allow the dead code in the `examples/common/` folder because of the interactive nature of our examples.

## 5.2 Projections and Viewing

In the preceding sections, we discussed basic 3D transformations, including scaling, translation, and rotation. Because our computer screen is two dimensional, it cannot directly display 3D objects. In order to view 3D objects on a 2D screen, we have to project our objects from 3D to 2D.

### 5.2.1 Transforming Coordinates

The traditional rendering pipeline in 3D graphics includes two main stages of vertex transformation, each with its own transformation matrix, as shown in Fig.5-4.

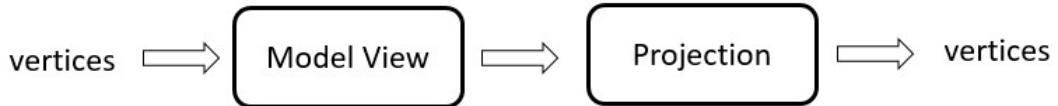


Fig.5-4. Two-stage vertex transformations.

Each vertex in the scene passes through two main stages of transformations: a model-view transformation followed by a projection. The model-view transformation includes basic transformations like scaling, translation, and rotation, as well as 3D viewing transformations. Projection can be either perspective or orthographic. We use these transformations to transform coordinates when drawing 3D primitives.

When a 3D object is rendered on a 2D computer screen, it actually invokes several coordinate systems, as shown in Fig.5-5.

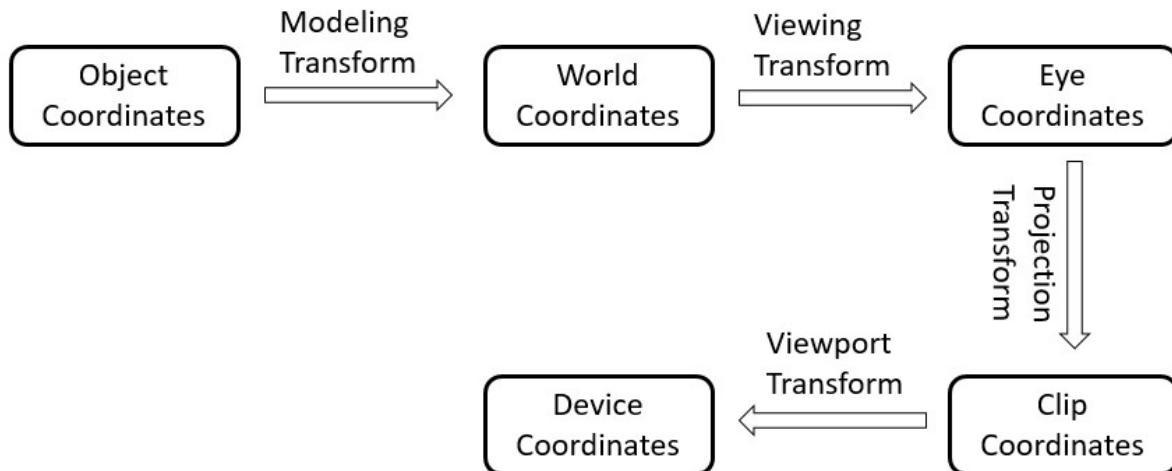


Fig.5-5. Coordinate systems and transformations.

## 76 | Practical GPU Graphics with `wgpu` and Rust

In order to draw a primitive like a point or triangle in a scene, it needs to go through several transformations and operations. We first specify the vertices that define the primitive in object coordinates, which are used to define the vertices of the object. We then perform a modeling transformation on the vertices to map them from object coordinates onto world coordinates. The modeling transform defines the position, orientation, and scale of a primitive in the world coordinate system. Different objects can be placed in the world coordinate system to form a scene.

We then convert the vertices from world coordinates into eye coordinates using a viewing transform. The viewing transform specifies the position and viewing direction of the “viewer” within the scene. The eye coordinates are the local system defined by the point of view onto the scene. The position of the view, the line of sight, and the upwards direction of the view define a coordinate system relative to the world coordinate system. We must draw the objects of a scene relative to the eye coordinate system in order to see them from the viewing position.

Next, we apply a projection transform to map the view volume seen by the viewer onto the clip coordinate space. If the transformed object lies outside the clip space, it will not be part of the image, and the processing stops. If part of the object lies inside and part of it outside, the part that lies outside is clipped away and discarded, and only the part that remains is processed further.

Finally, we use a viewport transform to convert the clip coordinates into a normalized device coordinates (NDC), which will be used to actually draw the object on the computer screen.

In WebGL or `wgpu`, we usually do not keep track of separate modeling and viewing transforms. Instead, we simply combine these two transforms into a single transform called a model-view transform. Thus, we can apply the model-view transform to map the object coordinates directly onto eye coordinates.

Alternatively, the model-view and projection transforms can be multiplied together to get a matrix that represents the combined transformation; object coordinates can then be multiplied by this matrix to transform them directly into clip coordinates.

Sometimes, eye coordinates are also called camera space. In this case, the viewing transform controls the transition from world coordinates into camera space, determining the position of the camera in the world coordinate system. The projection transform changes the geometry of 3D objects in camera space into clipping space and applies perspective distortion to the objects. Clipping space refers to the way the geometry is clipped to the view volume during this transformation. Finally, the geometry in clipping space is transformed onto the 2D screen. This transform is controlled by the viewport settings.

The `cgmath` crate provides a variety of functions that allow you to perform various transformations on 3D objects. It also constructs transform matrices for viewing and projection and performs the corresponding transformations. Understanding the mathematical basis of transform matrices and knowing how to construct them from camera and viewport settings is vital to developing professional 3D graphics applications.

In the following sections, you will learn how to construct viewing and projection transform matrices that can be used in `wgpu`.

### 5.2.2 Viewing Transform

In order to display 3D objects on a 2D computer screen, you must first decide how you want to position the objects in the scene, and you must choose a vantage point from which to view the scene. In the process, you will need to think in 3D coordinates while making the decisions that determine what is drawn on the screen.

The transformation process used to create the scene for viewing is analogous to taking a photograph with a camera. As such, using a viewing transform would be analogous to positioning and aligning the camera by changing the position and orientation of the viewport.

The viewing transform locates the viewer in world space and transforms 3D objects into camera space. In camera space, the camera, or viewer, is at the origin, looking in the negative  $z$  direction. Recall that *wgpu* uses a right-handed coordinate system, so  $z$  is negative in a scene. The viewing matrix relocates the objects in the world coordinate system around the camera's position – specifically, the origin of the orientation of the camera.

There are many ways to construct a viewing matrix, but in every case, we use the logical position and orientation of the camera in world space as our starting point. The viewing matrix will be applied to the objects in a scene, translating and rotating them in order to place them in camera space, where the camera is at the origin. One way to construct a viewing matrix is to combine a translation matrix with rotation matrices for each axis. Here are steps we would use to do this:

- Translate the given camera position to the origin.
- Rotate about the  $y$ -axis in world space to bring the  $z$ -axis of the camera coordinates into the  $y$ - $z$  plane of world coordinates
- Rotate about the world coordinate  $x$ -axis until the  $z$ -axis of the world and camera spaces are aligned.
- Rotate about the world coordinate  $z$ -axis to align the  $y$ -axis of the world and camera spaces.

Through the above analysis, we can then construct the viewing matrix:

$$V = T \cdot R(y) \cdot R(x) \cdot R(z)$$

In this formula,  $V$  is the viewing matrix;  $T$  is a translation matrix that repositions objects (or the camera) in world space; and  $R(x)$ ,  $R(y)$  and  $R(z)$  are rotation matrices that rotate objects around the  $x$ ,  $y$ , and  $z$  axes. The translation and rotation matrices are based on the camera's logical position and orientation in world space. So, if the camera's logical position in the world space is  $(10, 20, 30)$ , the purpose of the translation matrix is to move objects  $-10$  units along the  $x$ -axis,  $-20$  units along the  $y$ -axis, and  $-30$  units along the  $z$ -axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how much the axes of the camera space are rotated out of alignment with world space.

In practice, the relationship between the camera and the object is specified by three sets of arguments: the camera position  $P(x, y, z)$ ; the look at vector  $N$  that defines the camera's viewing direction; and the up vector  $U$  which indicates which direction is up for the camera. We choose the camera position to yield the desired view of the scene, typically somewhere in the middle of the scene.

Here, I will describe how we construct the viewing transform matrix using these three parameters:

- Create three vectors in 3D space:  $XScale$ ,  $YScale$ , and  $ZScale$ .
- Normalize  $N$  and  $V$ , i.e., let both  $N$  and  $V$  be unit vectors.
- Let  $ZScale = N$ , i.e., set  $ZScale$  to the look at direction.
- Compute  $YScale$  using the formula:  $Yscale = \frac{\vec{U} - (\vec{U} \cdot \vec{N})\vec{N}}{\sqrt{1 - (\vec{U} \cdot \vec{N})^2}}$ .
- Compute  $XScale$  using the formula:  $Xscale = \frac{\vec{N} \times \vec{U}}{\sqrt{1 - (\vec{U} \cdot \vec{N})^2}}$ .
- Construct an  $M$  matrix in the 3D homogeneous coordinate system:

## 78 | Practical GPU Graphics with wgpu and Rust

$$M = \begin{pmatrix} XScale.X & YScale.X & ZScale.X & 0 \\ XScale.Y & YScale.Y & ZScale.Y & 0 \\ XScale.Z & YScale.Z & ZScale.Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

- Translate the camera position to the origin and reflect it about the  $z$ -axis. Therefore, the viewing transform matrix  $V$  is given by:  $V = T(-x, -y, -z) \cdot M \cdot S(1, 1, -1)$ , where  $T$  is a translation matrix and  $S$  is a scaling matrix with a scaling factor of  $sz = -1$ , which corresponds to the reflection about the  $z$ -axis.

By following the above steps, we can easily construct the viewing matrix by relating the  $XScale$ ,  $YScale$ , and  $ZScale$  parameters to the camera position, up direction, and look at direction.

In practice, we do not need to implement such a viewing matrix ourselves because the *cgmath* library already has a function called *Matrix4::look\_at\_rh* that can be used to generate the viewing matrix for us:

```
Matrix4::look_at_rh(eye, center, up);
```

Here, *eye* is a *Point3* point in 3D space representing the camera position, *center* is a *Point3* indicating the camera's look at direction, and *up* is a *Vector3* vector pointing to the camera's *up* direction. This method creates a homogeneous view matrix, which results in a vector pointing from eye to center oriented according to the up direction.

I will use an example to show you how to use the *Matrix4::look\_at\_rh* function to create a viewing matrix. Add a new Rust file called *view\_projection.rs* to the *examples/ch05/* folder. Enter the following code into this file:

```
use std::f32::consts::FRAC_PI_6;
use cgmath::*;

fn main() {
    // position of the viewer
    let eye: Point3<f32> = Point3::new(3.0, 4.0, 5.0);
    //point the viewer is looking at
    let center: Point3<f32> = Point3::new(-3.0, -4.0, -5.0);
    // vector pointing up
    let up: Vector3<f32> = Vector3::new(0.0, 1.0, 0.0);

    // construct view matrix:
    let view_mat = Matrix4::look_at_rh(eye, center, up);

    println!("position of viewer: {:?}", eye);
    println!("point the viewer is looking at: {:?}", center);
    println!("up direction: {:?}", up);
    println!("view matrix: {:?}\n ", view_mat);
}
```

In this example, we set the camera's location to [3, 4, 5], the look at direction to [-3, -4, -5], and the up direction to the *y* direction [0, 1, 0].

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch05_view_projection"
path = "examples/ch05/view_projection.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch05_view_projection
```

This produces the results in the terminal window shown in Fig.5-6.

```
position of viewer: Point3 [3.0, 4.0, 5.0]
point the viewer is looking at: Point3 [-3.0, -4.0, -5.0]
up direction: Vector3 [0.0, 1.0, 0.0]
view matrix: Matrix4 [[0.8574929, -0.29104275, 0.42426407, 0.0], [0.0, 0.824
62114, 0.56568545, 0.0], [-0.51449573, -0.48507127, 0.7071068, 0.0], [-0.0,
-0.0, -7.071068, 1.0]]
```

*Fig.5-6. Viewing matrix.*

### 5.2.3 Perspective Projection

The preceding section explained how to compose a viewing transform matrix so that the correct modeling and viewing transforms can be applied. In this section, I will explain how to define a perspective projection matrix, which is needed to transform the vertices in a scene.

The purpose of the projection transformation is to define a view volume, called the *view frustum*, which is used in two ways: it determines how an object is projected onto the screen, and it defines which objects or portions of objects are clipped out of the final image.

The key feature of perspective projection is foreshortening: the farther an object is from the camera, the smaller it appears on the screen. This occurs because the frustum for a perspective projection is a pyramid – specifically, a truncated pyramid whose top has been cut off by a plane parallel to its base. Objects that fall within the frustum are projected toward the apex of the pyramid, where the camera is located. Objects that are closer to the camera appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away. This method of projection is commonly used in 3D computer graphics and visual simulation, because it is similar to how a camera works.

Remember that the viewing volume is used to clip objects that lie outside of it. The four sides of the frustum, plus its top and its base, correspond to the six clipping planes of the viewing volume, as shown in Fig.5-7. Objects or parts of objects outside these planes are clipped from the final image.

Our task now is to construct a perspective transform matrix. A perspective projection maps the *x*- and *y*-coordinates onto the projection plane while maintaining depth information, which can be achieved by mapping the view frustum onto a cube. This cube is the projection into 3D space of what we call the clip space, and it is centered at the origin and extends from  $-1$  to  $1$  on each of the *x*-, *y*-, and *z*-axes.

Let  $P = (x, y, z, 1)$  be a point in camera space that lies inside the view frustum. We will try to construct a projection matrix using the parameters of this view frustum. Let us start by defining the rectangle carved out of the near plane by the four side planes of the view frustum as having its left edge at  $x = l$ , its right edge at  $x = r$ , its bottom edge at  $y = b$ , and its top edge at  $y = t$ .

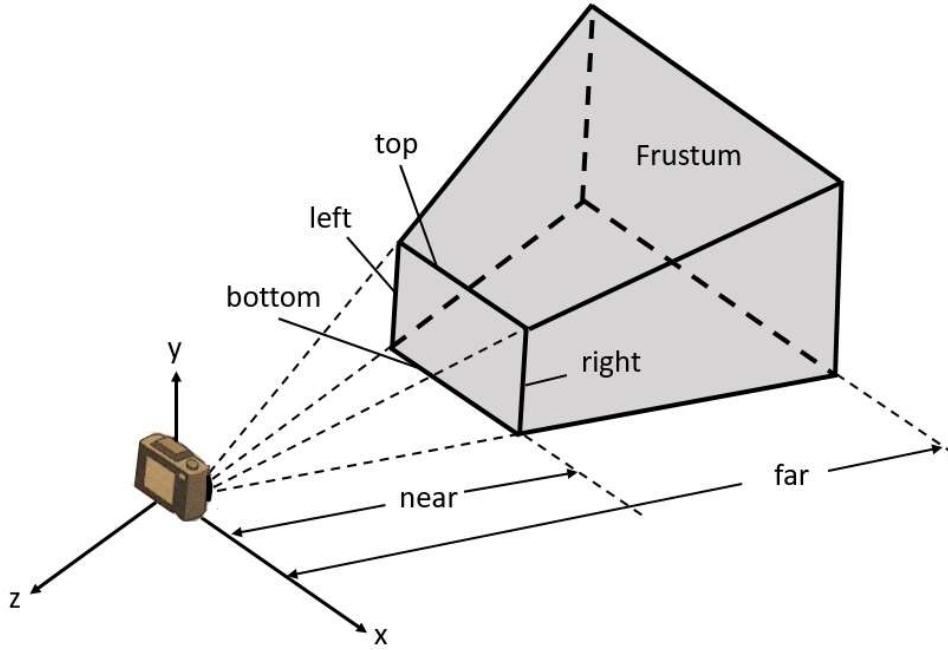


Fig.5-7. Projective view frustum.

The near plane and far plane are located at  $z = -zn$  and  $-zf$  in the right-hand coordinate system. So, we can now calculate the projected  $x1$  and  $y1$  coordinates of point  $P$  on the near plane:

$$x1 = -\frac{zn}{z}x, \quad y1 = -\frac{zn}{z}y \quad (5.8)$$

Note that  $z < 0$  because the camera points in the negative  $z$  direction. We want to map these coordinates into the range  $[-1, 1]$ , which we must do in order to fit the view frustum into the clip space. This can be done using the following relationships:

$$x2 = (x1 - l) \frac{2}{r - l} - 1, \quad y2 = (y1 - b) \frac{2}{t - b} - 1 \quad (5.9)$$

Substituting  $x1$  and  $y1$  in the above equation using (5.8) yields the result shown below:

$$x2 = \left( -\frac{x}{z} \right) \frac{2 \cdot zn}{r - l} - \frac{r + l}{r - l}, \quad y2 = \left( -\frac{y}{z} \right) \frac{2 \cdot zn}{t - b} - \frac{t + b}{t - b} \quad (5.10)$$

I should point out here that the range used to map the projected Z-component is different for different technologies. In OpenGL, the mapping range is from  $-1$  to  $+1$ . However, Microsoft uses a mapping range of  $[0, 1]$  for Direct3D. These two mapping ranges do give slightly different projection matrices, but this will not affect the final image displayed on your screen.

Here, we will use the range  $[0, 1]$  for mapping the projected  $z$ -component. This mapping involves a somewhat more complex computation. Since point  $P$  lies inside the view frustum, its  $z$ -component must be in the range  $[-zf, -zn]$ . We need to find a function that maps  $-zn \rightarrow 0$  and  $-zf \rightarrow 1$ . Let us assume that the mapping function has the following form:

$$z_2 = \frac{C1}{z} + C2 \quad (5.11)$$

We can solve for the coefficients  $C1$  and  $C2$  by plugging in the known mappings  $-zn \rightarrow 0$  and  $-zf \rightarrow 1$  to obtain

$$0 = \frac{C1}{-zn} + C2, \quad 1 = \frac{C1}{-zf} + C2$$

Solving the above equations yields these results:

$$C1 = \frac{zn \cdot zf}{zf - zn}, \quad C2 = \frac{zf}{zf - zn}$$

Substituting the above equations into (5.11), we have

$$z_2 = \left( \frac{1}{z} \right) \left( \frac{zn \cdot zf}{zf - zn} \right) + \frac{zf}{zf - zn} \quad (5.12)$$

You can see from equations (5.10) and (5.12) that  $x_2$ ,  $y_2$ , and  $z_2$  all contain a division by  $-z$ , so the 3D point  $P2 = (x_2, y_2, z_2)$  is equivalent to a 4D point in the homogeneous coordinate system:

$$P2 = (-x_2 \cdot z, -y_2 \cdot z, -z_2 \cdot z, -z)$$

From (5.10) and (5.12) we obtain the values of  $-x_2 z$ ,  $-y_2 z$ , and  $-z_2 z$  through the following equations:

$$\begin{aligned} -x_2 \cdot z &= \frac{2 \cdot zn}{r - l} x + \frac{r + l}{r - l} z \\ -y_2 \cdot z &= \frac{2 \cdot zn}{t - b} y + \frac{t + b}{t - b} z \\ -z_2 \cdot z &= -\frac{zf}{zf - zn} z - \frac{zn \cdot zf}{zf - zn} \end{aligned}$$

$$w = -z$$

The above equations can be rewritten in matrix form:

$$P2 = P \cdot M_{\text{perspective}} = (x \ y \ z \ 1) \begin{pmatrix} \frac{2 \cdot zn}{r - l} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot zn}{t - b} & 0 & 0 \\ \frac{r + l}{r - l} & \frac{t + b}{t - b} & \frac{zf}{zn - zf} & -1 \\ 0 & 0 & \frac{zn \cdot zf}{zn - zf} & 0 \end{pmatrix} \quad (5.13)$$

$M_{\text{perspective}}$  in the above equation is the perspective projection matrix. This matrix is in a general form and applies whether the view frustum is symmetric or not.

## 82 | Practical GPU Graphics with wgpu and Rust

For a symmetric view frustum, we can specify the width ( $w$ ) and height ( $h$ ) of the near plane, and use the following relationships:

$$l = -w/2, \quad r = w/2, \quad b = -h/2, \quad t = h/2$$

Then the perspective matrix in equation (5.13) reduces to

$$M_{\text{perspective}} = \begin{pmatrix} \frac{2 \cdot zn}{w} & 0 & 0 & 0 \\ 0 & \frac{2 \cdot zn}{h} & 0 & 0 \\ 0 & 0 & \frac{zf}{zn - zf} & -1 \\ 0 & 0 & \frac{zn \cdot zf}{zn - zf} & 0 \end{pmatrix} \quad (5.14)$$

Sometimes, it is convenient to specify the view frustum using the field of view ( $\text{fov}$ ) in the  $y$  direction and the aspect ratio, instead of the width and height. In this case, we have the relationships:

$$\text{yscale} = \frac{zn}{h/2} = \frac{1}{\tan(\text{fov}/2)}, \quad \text{xscale} = \frac{\text{yscale}}{\text{aspectRatio}}$$

Here  $\text{aspectRatio} = w/h$ . Thus, we can express the perspective matrix in terms of the field of view angle:

$$M_{\text{perspective}} = \begin{pmatrix} \text{xscale} & 0 & 0 & 0 \\ 0 & \text{yscale} & 0 & 0 \\ 0 & 0 & \frac{zf}{zn - zf} & -1 \\ 0 & 0 & \frac{zn \cdot zf}{zn - zf} & 0 \end{pmatrix} \quad (5.15)$$

The `cgmath` library has several functions that allow you to compute the frustum and projection matrices. For example:

```
cgmath::frustum(left, right, bottom, top, near, far);
```

This function will generate a frustum matrix with the given bound parameters. Meanwhile, the following function allows you to calculate a perspective projection matrix with the given bounds:

```
cgmath::perspective(fovy, aspect, near, far);
```

Here,  $\text{fovy}$  is the vertical field of view in radians. Note that the  $\text{far}$  bound of the frustum can be null or infinity. Passing a value of `null/undefined/no` for  $\text{far}$  will generate an infinite projection matrix.

Now, add the following code to the `view_projection.rs` file in the `examples/ch05/` folder:

```
use std::f32::consts::FRAC_PI_6;
use cgmath::*;

fn main() {
```

```
// ... code omitted for brevity

// frustum and perspective parameters
let left = -3.0;
let right = 3.0;
let bottom = -5.0;
let top = 5.0;
let near = 1.0;
let far = 100.0;
let fovy = FRAC_PI_6;
let aspect = 1.5;

// construct the frustum matrix
let frustum_mat = frustum(left, right, bottom, top, near, far);

// construct perspective projection matrix
let persp_mat = perspective(Rad(fovy), aspect, near, far);

println!("\nfrustum matrix: {:?}", frustum_mat);
println!("perspective matrix: {:?}", persp_mat);
}
```

Rerun the application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch05_view_projection
```

This produces the results in the terminal window shown in Fig.5-8.

```
frustum matrix: Matrix4 [[0.33333334, 0.0, 0.0, 0.0], [0.0, 0.2, 0.0, 0.0],
[0.0, 0.0, -1.020202, -1.0], [0.0, 0.0, -2.020202, 0.0]]

perspective matrix: Matrix4 [[2.488034, 0.0, 0.0, 0.0], [0.0, 3.732051, 0.0,
0.0], [0.0, 0.0, -1.020202, -1.0], [0.0, 0.0, -2.020202, 0.0]]
```

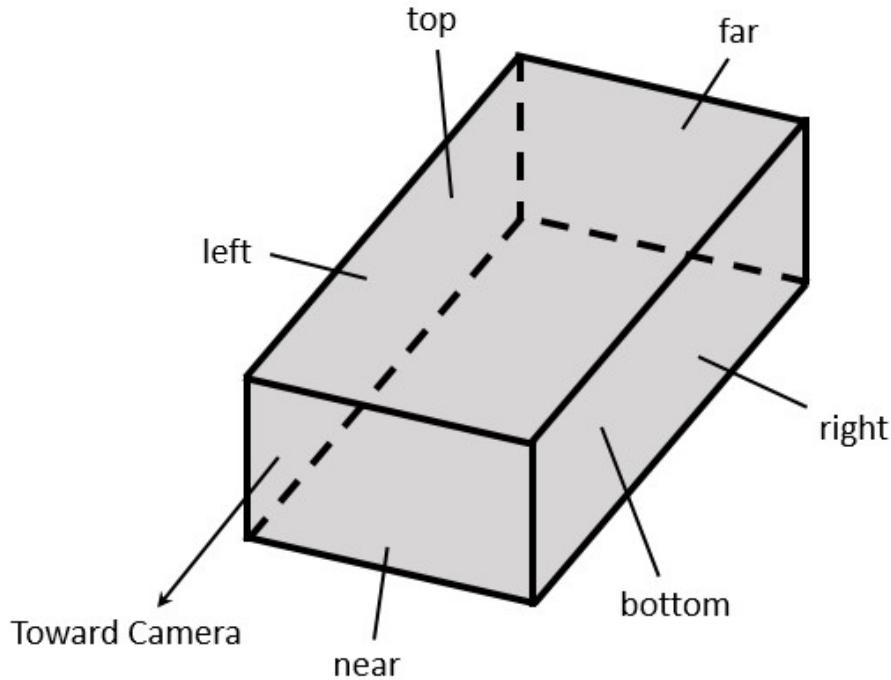
*Fig.5-8. Frustum and perspective projection matrices.*

## 5.2.4 Orthographic Projection

The viewing volume for an orthographic projection is a rectangular parallelepiped, or a box, as shown in Fig.5-9.

Unlike perspective projection, the size of the viewing volume does not change from one end to the other, so the distance from the camera does not affect how large an object appears. Namely, no perspective distortion occurs in orthographic projections. Points in camera space are always mapped to the projection plane by casting rays that are parallel to the camera's viewing direction. This type of projection is usually used in applications for architecture and computer-aided design (CAD), where it is important to maintain the actual sizes of objects and the angles between them as they are projected.

Orthographic projection transforms a viewing volume like the one shown in Fig.5-9 into a box with the parameters  $-1 \leq x \leq 1$ ,  $-1 \leq y \leq 1$ , and  $0 \leq z \leq 1$ . In other technologies, including OpenGL, the range  $[-1, 1]$  is used for mapping in the  $z$  direction.



*Fig.5-9. Orthographic viewing volume.*

Since there is no perspective distortion, an orthographic projection must be a linear transformation. Therefore, the transform can be written in the following form:

$$x_1 = Cx_1 \cdot x + Cx_2$$

$$y_1 = Cy_1 \cdot y + Cy_2$$

$$z_1 = Cz_1 \cdot z + Cz_2$$

Using the mapping relationships described above leads to the results:

$$-1 = Cx_1 \cdot l + Cx_2, \quad +1 = Cx_1 \cdot r + Cx_2$$

$$-1 = Cy_1 \cdot b + Cy_2, \quad +1 = Cy_1 \cdot t + Cy_2$$

$$0 = Cz_1 \cdot zn + Cz_2, \quad +1 = Cz_2 \cdot zf + Cz_2$$

We can obtain an orthographic transform by solving the above equation:

$$x_1 = \frac{2}{r - l}x - \frac{r + l}{r - l}$$

$$y_1 = \frac{2}{t - b}y - \frac{r + l}{r - l}$$

$$z_1 = \frac{1}{zf - zn}z - \frac{zn}{zf - zn}$$

We can then get the orthographic projection matrix by rewriting the above equations in matrix form:

$$P_1 = P \cdot M_{orthographic} = (x, y, z, 1) \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{1}{zf-zn} & 0 \\ \frac{l+r}{l-r} & \frac{b+t}{b-t} & \frac{zn}{zn-zf} & 1 \end{pmatrix} \quad (5.16)$$

$M_{orthographic}$  is the orthographic projection matrix, which is valid for general cases, including cases where the viewing volume is asymmetric or off center.

For symmetric cases, we can specify the width ( $w$ ) and height ( $h$ ) of the near plane, and use the following relationships:

$$l = -w/2, \quad r = w/2, \quad b = -h/2, \quad t = h/2$$

The orthographic projection matrix becomes

$$M_{orthographic} = \begin{pmatrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & \frac{1}{zn-zf} & 0 \\ 0 & 0 & \frac{zn}{zn-zf} & 1 \end{pmatrix} \quad (5.17)$$

The `cgmath` library has implemented the following function for calculating an orthographic projection matrix:

```
cgmath::ortho(left, right, bottom, top, near, far);
```

Now, add the following code to the `view_projection.rs` file in the `examples/ch05/` folder:

```
use std::f32::consts::FRAC_PI_6;
use cgmath::*;

fn main() {
    // ... code omitted for brevity

    // construct orthographic projection matrix
    let mut ortho_mat = [0.0; 16];
    let ortho_mat = ortho(left, right, bottom, top, near, far);
    println!("orthographic matrix: {:?}", ortho_mat);
}
```

Rerun the application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch05_view_projection
```

This produces the results in the terminal window shown in Fig.5-10.

```
orthographic matrix: Matrix4 [[0.33333334, 0.0, 0.0, 0.0], [0.0, 0.2, 0.0, 0  
.0], [0.0, 0.0, -0.02020202, 0.0], [-0.0, -0.0, -1.020202, 1.0]]
```

*Fig.5-10. Orthographic projection matrix.*

## 5.3 Transformations in `wgpu`

The purpose of the model-view transformation and projection is to transform coordinates when drawing primitives. In `wgpu`, these transformations are usually done in the vertex shader. In the shader program, we can pass arbitrary attributes to the vertex shader for each vertex. But what about global variables? We could include global transformations such as model-view transformations, projection, material parameters, and texture data as vertex data, but that would be a waste of memory and it would require us to update the vertex buffer whenever the transformation changes, which could easily happen every single frame.

The right way to tackle transformations is to use WGSL uniform variables of type `mat4`. Shaders can use either separate model-view and projection matrices or a combined matrix. Sometimes, we require a separate model-view transform matrix because certain lighting calculations are done in camera space.

In `wgpu`, we use a uniform buffer object (UBO) to pass uniform variables. The uniform buffer is a global buffer that needs to set its value once, and can then be reused in rendering your graphics. If your transformation changes, you only need to modify the corresponding data in the uniform buffer.

In the following chapter, I will show you how to use uniform buffers and bind groups in `wgpu` to create 3D objects.

# 6 3D Shapes and Camera

In the preceding chapter, I explained various 3D transformations and projections, and demonstrated how to construct corresponding matrices using the *cgmath* library. In this chapter, I will show you how to use transformation and projection matrices to create a few real 3D shapes – a 3D line and two cubes, one with distinct face colors and the other with distinct vertex colors.

From these examples, you will learn two very important concepts in *wgpu*: bind groups and uniform buffer objects (UBOs). We will use UBOs to represent transformation and projection matrices, and then use bind groups to pass the uniform buffers to our vertex shader. This allows us to add camera and mouse controls to our 3D shapes.

## 6.1 Uniform Buffers and Bind Groups

If you have a lot of information, such as model-view transformations, projections, and material parameters, the number of calls that must be set on a per frame level can become enormous, generating a heavy amount of undesired overhead. In a shader program, we typically consider this information as global inputs, which cannot simply be passed as standard vertex attributes. Therefore, in *wgpu* we use a uniform buffer object to pass these global variables.

A uniform buffer is a global buffer that needs to set its value once, and can then be reused in rendering graphics. If your transformations change, you only need to modify the corresponding data in the uniform buffer.

In *wgpu*, you will find that model-view and projection matrices usually use the same values across different programs. A UBO allows you to share uniform information among applications. Thus, you can use a uniform buffer to store uniform data, and then tell your programs to use a particular buffer object to find the appropriate uniform data.

To map global variables to parameters, we use bind groups. You can think of bind groups as roughly analogous to the vertex buffers we explored in Chapter 4, but instead of containing vertex data, a bind group specifies an array of buffers and textures to pass to our shader programs, plus the parameter binding indices that allow the shader to access this data.

The *line3d.rs* example presented in the following section uses a uniform buffer object to store a combined model-view and projection matrix named *mvpMat* (i.e., model-view-projection matrix). We typically use uniform buffers to pass smaller buffers containing constant data to shaders. If we want to pass larger or shader writable buffers, *wgpu* provides storage buffers that we can use instead.

## 88 | Practical GPU Graphics with wgpu and Rust

In WGLS, we can bind multiple bind groups to different input slots, which are known as groups. Within each input slot group, each bind group contains parameter bindings that correspond to specific shader parameters. We annotate uniform shader parameters with the corresponding group and binding that provide them with the underlying data. In our case, we want a vertex shader to apply a transform to our vertices, which we will do by passing a uniform buffer containing a transform matrix to the shader. We will have a corresponding group called *group(0)*, which will bind the uniform buffer to the corresponding binding *binding(0)*, as shown below:

```
[[block]] struct Uniforms {
    mvpMatrix : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Output {
    [[builtin(position)]] Position : vec4<f32>;
    [[location(0)]] vColor : vec4<f32>;
};

[[stage(vertex)]]
fn main([[location(0)]] pos: vec4<f32>, [[location(1)]] color: vec4<f32>) -> Output {
    var output: Output;
    output.Position = uniforms.mvpMatrix * pos;
    output.vColor = color;
    return output;
}
```

Here, the highlighted code defines a uniform struct named *Uniforms*. A uniform struct is a series of uniform definitions that get their data from a uniform buffer, rather than having it stored in the shader program object itself. In our case, this struct has only one member, called *mvpMat*, of the *mat4x4<f32>* type. Nothing else in the vertex shader needs to be changed.

The bind group layout is created using *device.create\_bind\_group\_layout*:

```
let uniform_bind_group_layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
    entries: &[wgpu::BindGroupLayoutEntry {
        binding: 0,
        visibility: wgpu::ShaderStages::VERTEX,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    }],
    label: Some("Uniform Bind Group Layout"),
});
```

Here, the *dynamic* field indicates whether this buffer will change its size or not. This will be useful if we want to store an array of things in the uniform buffer.

Next, we need to create a uniform buffer to store our combined transformation. We will define its usage to include usage as both a uniform buffer and a copy destination buffer:

```
let uniform_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Uniform Buffer"),
    contents: bytemuck::cast_slice(mvp_ref),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});
```

We use `bind_group` to specify the actual buffers or textures that will be passed to our shaders:

```
let uniform_bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});
```

Finally, we need to register our `uniform_bind_group_layout` with the render pipeline:

```
let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
```

In the following section, I will show you how to use uniform buffers and bind groups in `wgpu` to create a simple 3D line object.

## 6.2 Creating a 3D Line

In this section, we will create a `wgpu` application that draws a 3D spiral line using a set of parametric equations. First, I will place the commonly used code into one common file that we can reuse in our 3D shape examples for the rest of this book.

### 6.2.1 Common Code

In the preceding chapter, we added a `transforms.rs` file to the `examples/common/` folder and implemented a method called `create_transforms` in this file. Here, we will place all `wgpu` initialization-related code into a struct called `InitWgpu` and an async method called `init_wgpu` by adding the following code to this file:

```
use std::f32::consts::PI;
use cgmath::*;
use winit::window::Window;

pub struct InitWgpu {
    pub surface: wgpu::Surface,
    pub device: wgpu::Device,
    pub queue: wgpu::Queue,
    pub config: wgpu::SurfaceConfiguration,
    pub size: winit::dpi::PhysicalSize<u32>,
}

impl InitWgpu {
    pub async fn init_wgpu(window: &Window) -> Self {
        let size = window.inner_size();
        let instance = wgpu::Instance::new(wgpu::Backends::all());
        let surface = unsafe { instance.create_surface(window) };
        let adapter = instance
            .request_adapter(&wgpu::RequestAdapterOptions {
                power_preference: wgpu::PowerPreference::default(),
                compatible_surface: Some(&surface),
            })
            .await
            .expect("Failed to find GPU adapter");
        let device = adapter
            .request_device(&wgpu::DeviceDescriptor {
                features: wgpu::Features::empty(),
                limits: wgpu::Limits::empty(),
                label: None,
            })
            .await
            .expect("Failed to create GPU device");
        let queue = device.create_queue();
        let config = wgpu::SurfaceConfiguration {
            width: size.width,
            height: size.height,
            usage: wgpu::Usage::OUTPUT_ATTACHMENT,
        };
        surface.configure(device, config);
        Self {
            surface,
            device,
            queue,
            config,
            size,
        }
    }
}
```

## 90 | Practical GPU Graphics with wgpu and Rust

```
        force_fallback_adapter:false,
    })
    .await
    .unwrap();

let (device, queue) = adapter
    .request_device(
        &wgpu::DeviceDescriptor {
            label: None,
            features: wgpu::Features::empty(),
            limits: wgpu::Limits::default(),
        },
        None, // Trace path
    )
    .await
    .unwrap();

let config = wgpu::SurfaceConfiguration {
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    format: surface.get_preferred_format(&adapter).unwrap(),
    width: size.width,
    height: size.height,
    present_mode: wgpu::PresentMode::Fifo,
};
surface.configure(&device, &config);

Self{
    surface,
    device,
    queue,
    config,
    size,
}
}
```

This initialization code will be shared across most of our *wgpu* applications.

In Chapter 5, we already implemented the *create\_transforms* method in the *transforms.rs* file, which allows us to combine scaling, translation, and rotation together to form a combined transform matrix. Now, we add to this file another function, *create\_view\_projection*, which returns various view-projection matrices by view and projection operations. Here is the code for this function:

```
use std::f32::consts::PI;
use winit::window::Window;
use cgmath::*;

#[rustfmt::skip]
#[allow(unused)]
pub const OPENGL_TO_WGPU_MATRIX: Matrix4<f32> = Matrix4::new(
    1.0, 0.0, 0.0, 0.0,
    0.0, 1.0, 0.0, 0.0,
    0.0, 0.0, 0.5, 0.0,
    0.0, 0.0, 0.5, 1.0,
);

pub fn create_view_projection(camera_position: Point3<f32>, look_direction: Point3<f32>,
    up_direction: Vector3<f32>, aspect:f32, is_perspective:bool) ->
    (Matrix4<f32>, Matrix4<f32>, Matrix4<f32>) {
```

```
// construct view matrix
let view_mat = Matrix4::look_at_rh(camera_position, look_direction, up_direction);

// construct projection matrix
let project_mat:Matrix4<f32>;
if is_perspective {
    project_mat = OPENGL_TO_WGPU_MATRIX * perspective(Rad(2.0*PI/5.0), aspect, 0.1, 100.0);
} else {
    project_mat = OPENGL_TO_WGPU_MATRIX * ortho(-4.0, 4.0, -3.0, 3.0, -1.0, 6.0);
}

// construct view-projection matrix
let view_project_mat = project_mat * view_mat;

// return various matrices
(view_mat, project_mat, view_project_mat)
}
```

Here, we first introduce a constant matrix called `OPENGL_TO_WGPU_MATRIX`, which we use because OpenGL and `wgpu` have different coordinate systems. The coordinate system in `wgpu` is based on DirectX and Metal's coordinate system, meaning that in normalized device coordinates (NDC), the  $x$  axis and  $y$  axis are in the range  $[-1.0, 1.0]$ , while the  $z$  axis is in the range  $[0.0, 1.0]$ . Meanwhile, the `cgmath` crate is built for OpenGL's coordinate system where the  $x$ ,  $y$ , and  $z$ -axes are all in the range  $[-1.0, 1.0]$ . This constant matrix will scale and translate our scene to convert it from OpenGL's coordinate system into `wgpu`'s coordinate system. Of course, we do not explicitly need this matrix, but without it, models centered at  $(0, 0, 0)$  will be halfway inside the clipping area.

The `create_view_projection` function takes `camera_position`, `look_direction`, and `up_direction` as its input arguments, and creates a viewing matrix using the `Matrix4::look_at_rh` function implemented in the `cgmath` crate. It also accepts the aspect ratio and `is_perspective` as input arguments. Note that our projection matrix is multiplied by the `OPENGL_TO_WGPU_MATRIX`. This function returns three `Matrix4<f32>` matrices that represent the view, projection, and view-projection matrices respectively.

The Boolean input parameter `is_perspective` in the `create_view_projection` method lets you specify whether the projection is perspective (`is_perspective = true`) or orthographic (`is_perspective = false`).

If we want our 3D objects to behave correctly when resizing, we may need to separate the view matrix and the projection matrix. Add the following two methods to the `transforms.rs` file:

```
pub fn create_view(camera_position: Point3<f32>, look_direction: Point3<f32>,
    up_direction: Vector3<f32>) -> Matrix4<f32> {
    Matrix4::look_at_rh(camera_position, look_direction, up_direction)
}

pub fn create_projection(aspect:f32, is_perspective:bool) -> Matrix4<f32> {
    let project_mat:Matrix4<f32>;
    if is_perspective {
        project_mat = OPENGL_TO_WGPU_MATRIX * perspective(Rad(2.0*PI/5.0), aspect, 0.1, 100.0);
    } else {
        project_mat = OPENGL_TO_WGPU_MATRIX * ortho(-4.0, 4.0, -3.0, 3.0, -1.0, 6.0);
    }
    project_mat
}
```

You can use these two methods to create separate view and projection matrices.

## 6.2.2 Rust Code

Add a new sub-folder called `ch06` to the `examples/` folder, and then add a new Rust file called `line3d.rs` to the `examples/ch06/` folder with the following content:

```

use std::iter, mem;
use wgpu::util::DeviceExt;
use cgmath::*;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};
#[path = "../common/transforms.rs"]
mod transforms;

const IS_PERSPECTIVE:bool = false;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 3],
}

fn create_vertices() -> [Vertex; 300]{
    let mut vertices = [Vertex{position:[0.0,0.0,0.0]}; 300];
    for i in 0..300 {
        let t = 0.1*(i as f32)/30.0;
        let x = (-t).exp()*(30.0*t).sin();
        let z = (-t).exp()*(30.0*t).cos();
        let y = 2.0*t-1.0;
        vertices[i] = Vertex{position:[x, y, z]};
    }
    vertices
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 1] = wgpu::vertex_attr_array![0=>Float32x3];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_buffer: wgpu::Buffer,
    uniform_bind_group: wgpu::BindGroup,
    model_mat: Matrix4<f32>,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
}
impl State {

```

```

async fn new(window: &Window) -> Self {
    let init = transforms::InitWgpu::init_wgpu(window).await;

    let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
        label: Some("Shader"),
        source: wgpu::ShaderSource::Wgsl(include_str!("line3d.wgsl").into()),
    });

    // uniform data
    let camera_position = (1.5, 1.0, 3.0).into();
    let look_direction = (0.0, 0.0, 0.0).into();
    let up_direction = cgmath::Vector3::unit_y();

    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0]);
    let (view_mat, project_mat, view_project_mat) =
        transforms::create_view_projection(camera_position, look_direction, up_direction,
            init.config.width as f32 / init.config.height as f32, false);
    let mvp_mat = view_project_mat * model_mat;

    let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
    let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
        label: Some("Uniform Buffer"),
        contents: bytemuck::cast_slice(mvp_ref),
        usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    });

    let uniform_bind_group_layout =
        init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
            entries: &[wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            }],
            label: Some("Uniform Bind Group Layout"),
        });
}

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {

```

## 94 | Practical GPU Graphics with wgpu and Rust

```
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::LineStrip,
        strip_index_format: Some(wgpu::IndexFormat::UInt32),
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
```

```
let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&create_vertices()),
    usage: wgpu::BufferUsages::VERTEX,
});
```

```
Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_buffer,
    uniform_bind_group,
    model_mat,
    view_mat,
    project_mat,
}
```

```
}
```

```
pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
        let mvp_mat = self.project_mat * self.view_mat * self.model_mat;
        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
    }
}
```

```
#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
```

```

        false
    }

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: None,
        });

        render_pass.set_pipeline(&self.pipeline);
        render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
        render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
        render_pass.draw(0..300, 0..1);
    }

    self.init.queue.submit(iter::once(encoder.finish()));
    output.present();
}

Ok(())
}

fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}", "ch06-3d-line"));
    let mut state = pollster::block_on(State::new(&window));

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,

```

## 96 | Practical GPU Graphics with wgpu and Rust

```

        window_id,
    } if window_id == window.id() => {
        if !state.input(event) {
            match event {
                WindowEvent::CloseRequested
                | WindowEvent::KeyboardInput {
                    input:
                        KeyboardInput {
                            state: ElementState::Pressed,
                            virtual_keycode: Some(VirtualKeyCode::Escape),
                            ..
                        },
                    ..
                } => *control_flow = ControlFlow::Exit,
                WindowEvent::Resized(physical_size) => {
                    state.resize(*physical_size);
                }
                WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                    state.resize(**new_inner_size);
                }
                _ => {}
            }
        }
    }
    Event::RedrawRequested(_) => {
        state.update();
        match state.render() {
            Ok(_) => {}
            Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
            Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
            Err(e) => eprintln!("{}:", e),
        }
    }
    Event::MainEventsCleared => {
        window.request_redraw();
    }
    _ => {}
});}
}

```

Here, we first define a *Vertex* struct with only one *position* field because we want to draw a 3D line with a single color that will be specified in the fragment shader. We then implement a method called *create\_vertices* that uses a mathematical parametric function to represent a 3D spiral line. This function returns the vertex data of 300 vertices. This way, you can define any arbitrary line function to create your own 3D lines in *wgpu*.

We then use a *wgpu* macro called *vertex\_attr\_array* to write an implementation block that creates a static method called *desc<'a>()*, which returns *VertexBufferLayout*:

```

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 1] = wgpu::vertex_attr_array![0=>Float32x3];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

```

Here, the `ATTRIBUTES` constant specifies a list that contains only one `VertexAttrib` with a `shader_location` of zero and a `format` of `Float32x3` for the vertex position.

Next, we pack all the fields required for drawing our graphics shape into another struct called `State`, and create some methods on it:

```
struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_buffer: wgpu::Buffer,
    uniform_bind_group: wgpu::BindGroup,
    model_mat: Matrix4<f32>,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
}
```

The `State` struct consists of a field called `init` that is of the `transforms::InitWgpu` type. This field contains various attributes associated with `wgpu` initialization, such as `device`, `surface`, `config`, etc.

Inside the `new` function in the `State` implementation, we initialize `wgpu` by calling the `transforms::init_wgpu` method, which avoids duplication of the `wgpu` initialization code:

```
let init = transforms::InitWgpu::init_wgpu(window).await;
```

Note that here we use the `await` keyword when calling our async `init_wgpu` function, which means that the application will suspend execution until the result of a `Future` is ready.

We then construct the `model`, `view`, and `projection` matrices by calling the `transforms::create_transforms` and `transforms::create_view_projection` methods.

Next, we create a uniform buffer to store our model-view-projection matrix `mvp_mat`. Note that we cannot use the `mvp_mat` matrix directly when creating the uniform buffer because `mvp_mat` is a `cgmath::Matrix4<f32>` type and the “`Pod`” trait is not implemented for this type. As a workaround, we convert `mvp_mat` into `mvp_ref`, which has a `&[f32; 16]` type which can be used in creating our uniform buffer:

```
let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Uniform Buffer"),
    contents: bytemuck::cast_slice(mvp_ref),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});
```

Now that we have a uniform buffer, we create a uniform-bind-group layout for it. Here, we only need the uniform information in the vertex shader to manipulate our vertices, so we set its visibility attribute to `wgpu::ShaderStages::VERTEX`. We then create the actual uniform bind group and register our uniform-bind-group layout with the render pipeline layout.

Next, we create the rendering pipeline, which is similar to what we used to create the basic primitives and 2D shapes in previous chapters. Please note that since we set the `primitive.topology` to “`LineStrip`”, we must specify the `strip_index_format` with `UInt32`.

In addition, the `State::new()` function also includes the `vertex_buffer` created using the function `device.create_buffer_init`. After initializing `wgpu` and configuring the surface and pipeline, we then add these new fields at the end of the `State::new()` function:

```
Self {
    init,
```

## 98 | Practical GPU Graphics with wgpu and Rust

```
    pipeline,
    vertex_buffer,
    uniform_buffer,
    uniform_bind_group,
    model_mat,
    view_mat,
    project_mat,
}
```

Note that inside the `resize` function, we update various fields associated with changes of the window size, including the projection matrix, because projection depends on the window's aspect ratio, which will change when the window is resized.

The rest of the code is similar to that used previously, except that we need to set the bind group to our uniform bind group inside the `render` function:

```
    render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
```

We also need to change the number of vertices to 300 in the `render_pass.draw` function:

```
    render_pass.draw(0..300, 0..1);
```

### 6.2.3 Shader Program

The shaders used in this example are different from those used in previous examples because now we need to incorporate uniform buffers for storing the transformation and projection matrices. Add a new `line3d.wgsl` file to the `examples/ch06/` folder and enter the following content into it:

```
[[block]] struct Uniforms {
    mvpMatrix : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

[[stage(vertex)]]
fn vs_main([[location(0)]] pos: vec4<f32>) -> [[builtin(position)]] vec4<f32> {
    return uniforms.mvpMatrix * pos;
}

[[stage(fragment)]]
fn fs_main() -> [[location(0)]] vec4<f32> {
    return vec4<f32>(1.0, 1.0, 0.0, 1.0);
}
```

You can see that we pass the model-view-projection matrix to the shader using a uniform buffer.

### 6.2.4 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch06_line3d"
path = "examples/ch06/line3d.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch06_line3d
```

This produces the results shown in Fig.6-1.

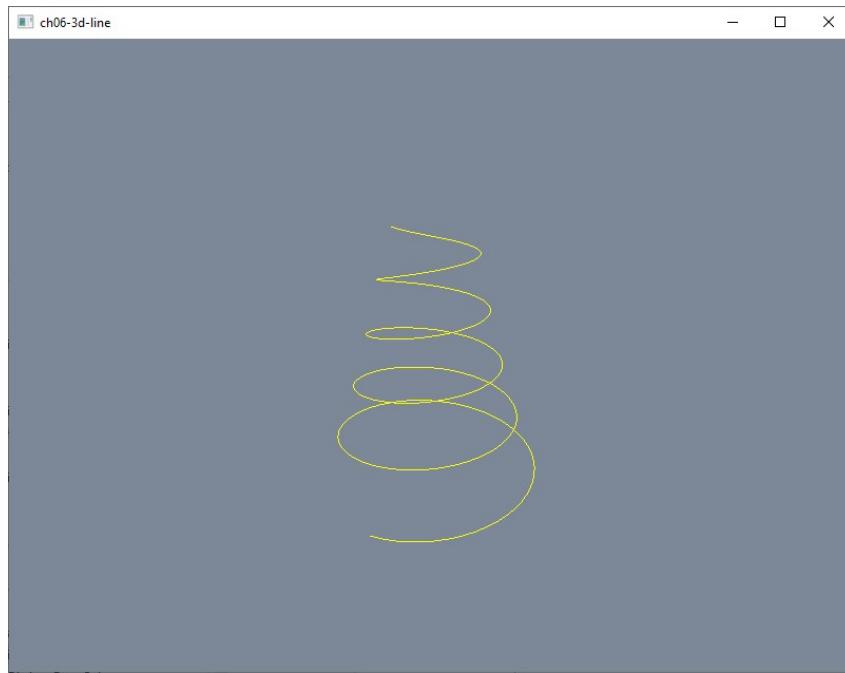


Fig.6-1. 3D line with perspective projection.

This 3D line is generated using perspective projection. We can also create a 3D line with orthographic projection, which can be done by simply setting the const IS\_PERSPECTIVE to false:

```
const IS_PERSPECTIVE:bool = false;
```

Rerunning the application produces a 3D line with orthographic projection as shown in Fig.6-2.

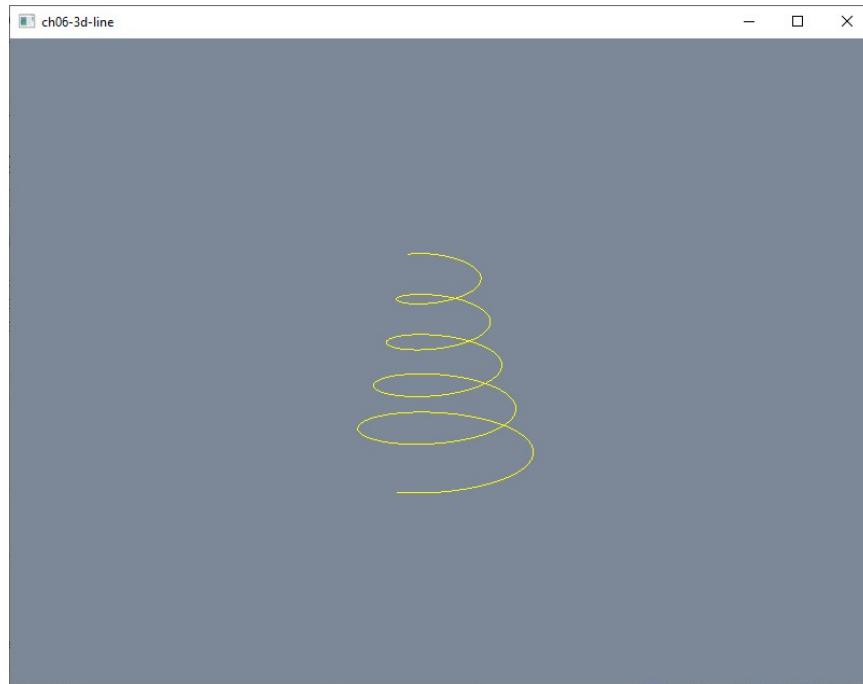


Fig.6-2. 3D line with orthographic projection.

## 6.3 Creating a Cube with Distinct Face Colors

In this section, we will create a 3D cube in *wgpu* with distinct face colors. The coordinates of a cube are shown in Fig.6-3.

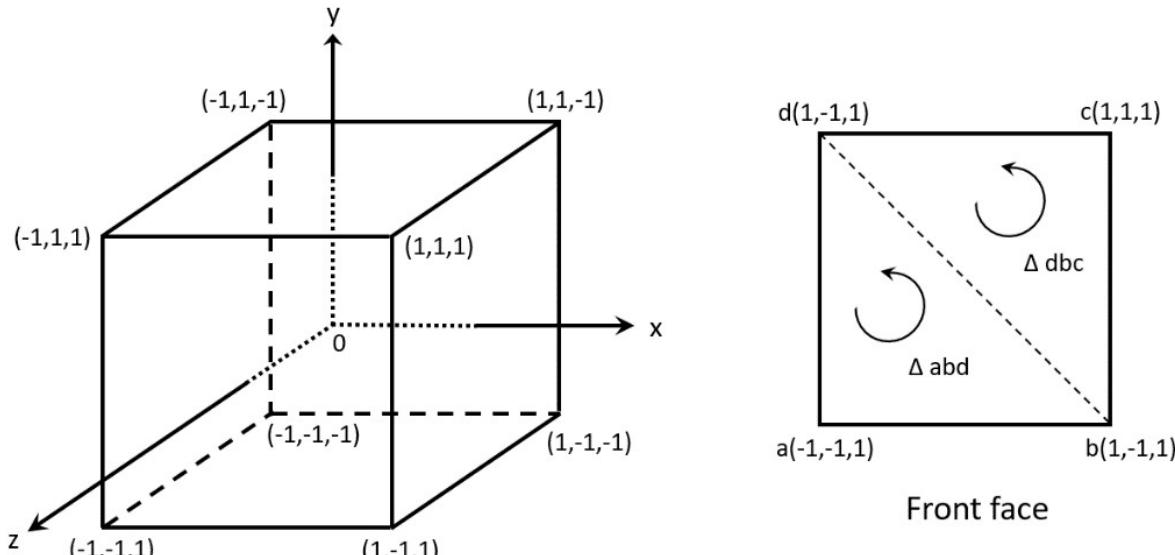


Fig.6-3. Coordinates of a cube. The right side shows the triangular meshes for the front face.

### 6.3.1 Create Vertex Data

Let us look at the front face of the cube, which has four vertices and needs two triangle primitives to render it. There are two approaches to rendering this face: using six vertices or using four vertices with indices, as demonstrated in Chapter 4. However, for our cube with distinct face colors, we cannot use the index method because each vertex will have a different color for different faces.

So that we can reuse the vertex data, we will place it in a common file. Add a new Rust file called *vertex\_data.rs* to the *examples/common/* folder and type the following content into it:

```
#![allow(dead_code)]  
  
pub fn cube_data() -> (<Vec<i8; 3>,<Vec<i8; 3>,<Vec<i8; 2>,<Vec<i8; 3>>) {  
    let positions = [  
        // front (0, 0, 1)  
        [-1, -1, 1], [1, -1, 1], [-1, 1, 1], [-1, 1, 1], [1, -1, 1], [1, 1, 1],  
        // right (1, 0, 0)  
        [1, -1, 1], [1, -1, -1], [1, 1, 1], [1, 1, -1], [1, -1, -1], [1, 1, -1],  
        // back (0, 0, -1)  
        [1, -1, -1], [-1, -1, -1], [1, 1, -1], [1, 1, -1], [-1, -1, -1], [-1, 1, -1],  
        // left (-1, 0, 0)  
        [-1, -1, -1], [-1, -1, 1], [-1, 1, -1], [-1, 1, 1], [-1, -1, 1], [-1, 1, 1],  
        // top (0, 1, 0)  
        [-1, 1, 1], [1, 1, 1], [-1, 1, -1], [-1, 1, -1], [1, 1, -1], [1, 1, 1], [1, 1, -1],
```

```

// bottom (0, -1, 0)
[-1, -1, -1], [ 1, -1, -1], [-1, -1,  1], [-1, -1,  1], [ 1, -1, -1], [ 1, -1,  1],
];

let colors = [
    // front - blue
    [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1],
    // right - red
    [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0],
    // back - yellow
    [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0], [1, 1, 0],
    // left - aqua
    [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1], [0, 1, 1],
    // top - green
    [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0],
    // bottom - fuchsia
    [1, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1], [1, 0, 1],
];

let uvs= [
    // front
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // right
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // back
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // left
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // top
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // bottom
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
];

let normals = [
    // front
    [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1],
    // right
    [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0], [1, 0, 0],
    // back
    [0, 0, -1], [0, 0, -1], [0, 0, -1], [0, 0, -1], [0, 0, -1], [0, 0, -1],
    // left
    [-1, 0, 0], [-1, 0, 0], [-1, 0, 0], [-1, 0, 0], [-1, 0, 0], [-1, 0, 0],
    // top
    [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0],
]

```

```

        // bottom
        [0, -1, 0], [0, -1, 0], [0, -1, 0], [0, -1, 0], [0, -1, 0], [0, -1, 0],
    ];
}

// return data
(positions.to_vec(), colors.to_vec(), uvs.to_vec(), normals.to_vec())
}

```

This code creates the position, color, UVs, and normal data using separate  $[i8; 3]$  vectors. The normal data represents the normal vector for each vertex, while the UV data represents texture coordinates. In the following chapters, we will use the normal data to compute lighting and the UV data to map textures instead of solid colors onto surfaces.

### 6.3.2 Rust Code

Add a new Rust file called `cube_face_color.rs` to the `examples/ch06/` folder with the following content:

```

use std:: {iter, mem};
use wgpu::util::DeviceExt;
use cgmath::*;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

const IS_PERSPECTIVE: bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 4],
    color: [f32; 4],
}

fn vertex(p:[i8;3], c:[i8; 3]) -> Vertex {
    Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
    }
}

fn create_vertices() -> Vec<Vertex> {
    let (pos, col, _uv, _normal) = vertex_data::cube_data();
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], col[i]));
    }
    data.to_vec()
}

```

```

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_buffer: wgpu::Buffer,
    uniform_bind_group: wgpu::BindGroup,
    model_mat: Matrix4<f32>,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
}
}

impl State {
    async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("cube_face_color.wgsl").into()),
        });

        // uniform data
        let camera_position = (3.0, 1.5, 3.0).into();
        let look_direction = (0.0, 0.0, 0.0).into();
        let up_direction = cgmath::Vector3::unit_y();

        let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0]);
        let (view_mat, project_mat, view_project_mat) =
            transforms::create_view_projection(camera_position, look_direction, up_direction,
                init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);
        let mvp_mat = view_project_mat * model_mat;

        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Uniform Buffer"),
            contents: bytemuck::cast_slice(mvp_ref),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });

        let uniform_bind_group_layout =
            init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
                entries: &[wgpu::BindGroupLayoutEntry {
                    binding: 0,
                    visibility: wgpu::ShaderStages::VERTEX,
                    ty: wgpu::BindingType::Buffer {
                        ty: wgpu::BufferBindingType::Uniform,
                        has_dynamic_offset: false,
                        min_binding_size: None,
                    },
                    count: None,
                }],
            });
    }
}

```

```

    }],
    label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        //cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
}

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&create_vertices()),
    usage: wgpu::BufferUsages::VERTEX,
}
);

```

```

});
```

```

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_buffer,
    uniform_bind_group,
    model_mat,
    view_mat,
    project_mat,
}
```

```

    }
}
```

```

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        self.project_mat = transforms::
            create_projection(new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
        let mvp_mat = self.project_mat * self.view_mat * self.model_mat;
        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
    }
}
```

```

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());
    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),

```

```

    });
}

{
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {
                    r: 0.2,
                    g: 0.247,
                    b: 0.314,
                    a: 1.0,
                }),
                store: true,
            },
        }],
        //depth_stencil_attachment: None,
        depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
            view: &depth_view,
            depth_ops: Some(wgpu::Operations {
                load: wgpu::LoadOp::Clear(1.0),
                store: false,
            }),
            stencil_ops: None,
        }),
    });
    render_pass.set_pipeline(&self.pipeline);
    render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
    render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
    render_pass.draw(0..36, 0..1);
}
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}","ch06-cube-face-color"));
    let mut state = pollster::block_on(State::new(&window));

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput { input:

```

```

    KeyboardInput {
        state: ElementState::Pressed,
        virtual_keycode: Some(VirtualKeyCode::Escape),
        ..
    },
    ..
} => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {:?}", e, e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

The code structure in this example is similar to that used in the preceding 3D line example. Here, we first define the *Vertex* struct with two [ $f32; 4$ ] fields that represent the position and color of each vertex. We then implement a helper function called *vertex*, which converts two input parameters of [ $i8; 3$ ] type into the *Vertex* struct, meaning we can simply enter integers as input without needing to type floating points for the vertex data.

Next, we implement another function called *create\_vertices*. Inside this function, we first call the *cube\_data()* function implemented in the *vertex\_data.rs* file to get the vertex data, and then use this data to populate a *Vertex* vector that will be used to create our 3D cube.

We then use the *wgpu* macro, *vertex\_attr\_array*, to write an implementation block that creates a static method called *desc<'a>()*, which returns *VertexBufferLayout*:

```

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

```

In this case, our *vertex\_attr\_array* contains two elements, one for position and the other for color.

In addition, we also add the *depth\_stencil* attribute to the render pipeline using the code:

```
depth_stencil: Some(wgpu::DepthStencilState {
    format: wgpu::TextureFormat::Depth24Plus,
    depth_write_enabled: true,
    depth_compare: wgpu::CompareFunction::LessEqual,
    stencil: wgpu::StencilState::default(),
    bias: wgpu::DepthBiasState::default(),
}),
```

If we do not set the *depth\_stencil* attribute or set it to *None*, we may get unexpected results: the sides of the cube will be drawn, but they will overlap each other in a strange way, as shown in Fig.6-4.

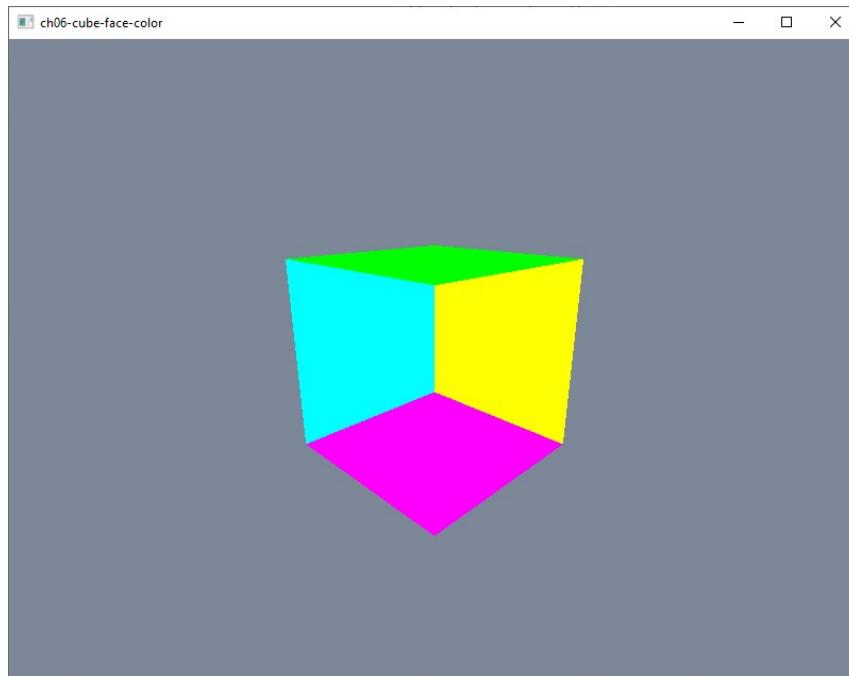


Fig.6-4. Cube created without the *depth\_stencil* setting.

The problem here is that when *wgpu* draws the cube triangle-by-triangle, it will simply write over pixels even though something else may have been drawn there before. In this case, *wgpu* will draw triangles in the back over triangles at the front. The *depth\_stencil* attribute tells *wgpu* when to draw over a pixel and when not to. To avoid such strange rendering issues in *wgpu*, I recommend that you always set this attribute for 3D objects.

In order for the *depth\_stencil* attribute to work, we also need to create a depth texture and add it to the render pass descriptor within the *render* function, as shown in the following code snippet:

```
let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
    size: wgpu::Extent3d {
        width: self.init.config.width,
        height: self.init.config.height,
        depth_or_array_layers: 1,
    },
    mip_level_count: 1,
    sample_count: 1,
    dimension: wgpu::TextureDimension::D2,
    format: wgpu::TextureFormat::Depth24Plus,
```

```

usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
label: None,
});
let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default()));

let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
    label: Some("Render Pass"),
    color_attachments: &[wgpu::RenderPassColorAttachment {
        view: &view,
        resolve_target: None,
        ops: wgpu::Operations {
            load: wgpu::LoadOp::Clear(wgpu::Color { r: 0.2, g: 0.247, b: 0.314, a: 1.0 }),
            store: true,
        },
    }],
    //depth_stencil_attachment: None,
    depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
        view: &depth_view,
        depth_ops: Some(wgpu::Operations {
            load: wgpu::LoadOp::Clear(1.0),
            store: false,
        }),
        stencil_ops: None,
    }),
});

```

Finally, we also need to change the `primitive.topology` to `TriangleList` and set the `stripe_index_format` to `None` when creating the render pipeline:

```

primitive: wgpu::PrimitiveState{
    topology: wgpu::PrimitiveTopology::TriangleList,
    strip_index_format: None,
    cull_mode: Some(wgpu::Face::Back),
    ..Default::default()
},

```

Note that we set the `cull_mode` attribute to `wgpu::Face::Back`, meaning that the back-facing polygons are discarded. For our cube, we only need to render the faces that are facing the viewer, so we can discard the faces that are facing backwards, which will save more than 50% of our cube's total fragment shader runs.

The rest of the code is the same as that used in our previous 3D line example.

### 6.3.3 Shader Program

The shaders are also different from what we used in the preceding example because in our current example, we need to take into account the color buffer. Add a `cube_face_color.wgsl` file to the `examples/ch06/` folder with the following content:

```

[[block]] struct Uniforms {
    mvpMatrix : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Output {
    [[builtin(position)]] Position : vec4<f32>;
    [[location(0)]] vColor : vec4<f32>;
};

```

```
[[stage(vertex)]]
fn vs_main([[location(0)]] pos: vec4<f32>, [[location(1)]] color: vec4<f32>) -> Output {
    var output: Output;
    output.Position = uniforms.mvpMatrix * pos;
    output.vColor = color;
    return output;
}

[[stage(fragment)]]
fn fs_main([[location(0)]] vColor: vec4<f32>) -> [[location(0)]] vec4<f32> {
    return vColor;
}
```

This shader is very similar to that used in the previous example, except that here, the color is processed using the data stored in the buffer.

### 6.3.4 Run Application

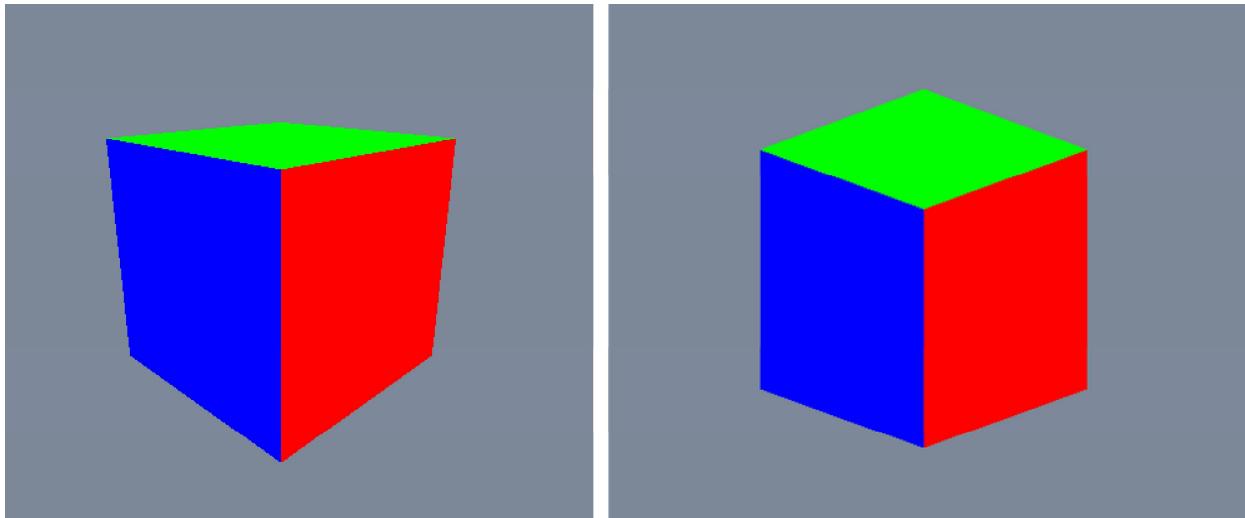
Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch06_cube_face_color"
path = "examples/ch06/cube_face_color.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch06_cube_face_color
```

This produces the results shown in Fig.6-5.



*Fig. 6-5. 3D cube with different projections: perspective (left) and orthographic (right)*

Here, we created the cube with two different projections, perspective and orthographic, by simply setting the constant *IS\_PERSPECTIVE* to *true* or *false* when creating the view-projection matrix.

## 6.4 Creating a Cube with Distinct Vertex Colors

In the preceding example, we created a cube with distinct face colors. In this section, we will create the same cube, but with distinct vertex colors instead. Since each vertex will have unique attributes in our current example, unlike the previous example, we can use an index buffer to avoid vertex data duplication.

### 6.4.1 Create Vertex Data

Add a new method called `cube_data_index` to the `vertex_data.rs` file in the `examples/common/` folder with the following code:

```
pub fn cube_data_index() -> (Vec<[i8; 3]>, Vec<[i8; 3]>, Vec<u16>) {
    let positions = [
        [-1, -1, 1], // vertex a
        [1, -1, 1], // vertex b
        [1, 1, 1], // vertex c
        [-1, 1, 1], // vertex d
        [-1, -1, -1], // vertex e
        [1, -1, -1], // vertex f
        [1, 1, -1], // vertex g
        [-1, 1, -1], // vertex h
    ];

    let colors = [
        [0, 0, 1], // vertex a
        [1, 0, 1], // vertex b
        [1, 1, 1], // vertex c
        [0, 1, 1], // vertex d
        [0, 0, 0], // vertex e
        [1, 0, 0], // vertex f
        [1, 1, 0], // vertex g
        [0, 1, 0], // vertex h
    ];

    let indices = [
        0, 1, 2, 2, 3, 0, // front
        1, 5, 6, 6, 2, 1, // right
        4, 7, 6, 6, 5, 4, // back
        0, 3, 7, 7, 4, 0, // left
        3, 2, 6, 6, 7, 3, // top
        0, 4, 5, 5, 1, 0, // bottom
    ];

    (positions.to_vec(), colors.to_vec(), indices.to_vec())
}
```

Here, we only need to create eight actual vertices. This way, we only store the unique vertices, and we can create another `u16` vector to store the indices, which will be used to create our cube.

### 6.4.2 Rust Code

Add a new Rust file called `cube_vertex_color.rs` to the `examples/ch06/` folder with the following code:

```
use std::{iter, mem};
use wgpu::util::DeviceExt;
```

## 112 | Practical GPU Graphics with wgpu and Rust

```
use cgmath::*;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

const IS_PERSPECTIVE: bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 4],
    color: [f32; 4],
}

fn vertex(p:[i8;3], c:[i8; 3]) -> Vertex {
    Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
    }
}

fn create_vertices() -> (Vec<Vertex>, Vec<u16>) {
    let (pos, col, ind) = vertex_data::cube_data_index();
    let mut data: Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], col[i]));
    }
    (data.to_vec(), ind)
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_buffer: wgpu::Buffer,
    uniform_bind_group: wgpu::BindGroup,
    index_buffer: wgpu::Buffer,
    indices_len: u32,
    model_mat: Matrix4<f32>,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
```

```

}

impl State {
    async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;
        let (vertex_data, index_data) = create_vertices();

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("cube_face_color.wgsl").into()),
        });

        // uniform data
        let camera_position = (3.0, 1.5, 3.0).into();
        let look_direction = (0.0, 0.0, 0.0).into();
        let up_direction = cgmath::Vector3::unit_y();

        let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [1.0, 1.0, 1.0]);
        let (view_mat, project_mat, view_project_mat) =
            transforms::create_view_projection(camera_position, look_direction, up_direction,
                init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);
        let mvp_mat = view_project_mat * model_mat;

        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Uniform Buffer"),
            contents: bytemuck::cast_slice(mvp_ref),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });

        let uniform_bind_group_layout =
            init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
                entries: &[wgpu::BindGroupLayoutEntry {
                    binding: 0,
                    visibility: wgpu::ShaderStages::VERTEX,
                    ty: wgpu::BindingType::Buffer {
                        ty: wgpu::BufferBindingType::Uniform,
                        has_dynamic_offset: false,
                        min_binding_size: None,
                    },
                    count: None,
                }],
                label: Some("Uniform Bind Group Layout"),
            });
        let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
            layout: &uniform_bind_group_layout,
            entries: &[wgpu::BindGroupEntry {
                binding: 0,
                resource: uniform_buffer.as_entire_binding(),
            }],
            label: Some("Uniform Bind Group"),
        });

        let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
            label: Some("Render Pipeline Layout"),
            bind_group_layouts: &[&uniform_bind_group_layout],
            push_constant_ranges: &[],
        });
    }
}

```

```

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
};

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});
;

let index_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Index Buffer"),
    contents: bytemuck::cast_slice(&index_data),
    usage: wgpu::BufferUsages::INDEX,
});
;
let indices_len = index_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_buffer,
    uniform_bind_group,
    index_buffer,
    indices_len,
    model_mat,
    view_mat,
}
;
```

```

        project_mat,
    }
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        self.project_mat = transforms::
            create_projection(new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
        let mvp_mat = self.project_mat * self.view_mat * self.model_mat;
        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {

```

```

        load: wgpu::LoadOp::Clear(wgpu::Color {
            r: 0.2,
            g: 0.247,
            b: 0.314,
            a: 1.0,
        }),
        store: true,
    },
},
],
//depth_stencil_attachment: None,
depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
    view: &depth_view,
    depth_ops: Some(wgpu::Operations {
        load: wgpu::LoadOp::Clear(1.0),
        store: false,
    }),
    stencil_ops: None,
}),
),
);
}

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_index_buffer(self.index_buffer.slice(..), wgpu::IndexFormat::Uint16);
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw_indexed(0..self.indices_len, 0, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}", "ch06-cube-vertex-color"));
    let mut state = pollster::block_on(State::new(&window));

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                            ..
                        },
                        ..
                    } => *control_flow = ControlFlow::Exit,
                    WindowEvent::Resized(physical_size) => {
```

```

```

        state.resize(*physical_size);
    }
    WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
        state.resize(**new_inner_size);
    }
    _ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!(":{}?", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

This code is very similar to that used in the preceding example, except that we use an index buffer to create our cube with distinct vertex colors. As with before, we first define the *Vertex* struct with two [ $f32; 4$ ] fields that represent position and color for each vertex. We then implement a helper function called *vertex* that converts two input parameters of [ $i8; 3$ ] type into the *Vertex* struct, meaning we can simply enter integers as input without needing to type floating points for the vertex data.

Now, we implement another function called *create\_vertices*. Inside this function, we first call the *cube\_data\_index()* function implemented in the *vertex\_data.rs* file to get the vertex and index data, and then use the vertex data to populate a *Vertex* vector, which will be used to create our 3D cube.

We need to create a buffer to store the index data:

```

let index_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Index Buffer"),
    contents: bytemuck::cast_slice(&index_data),
    usage: wgpu::BufferUsages::INDEX,
});
let indices_len = index_data.len() as u32;

```

Here, we do not need to implement *Pod* and *Zeroable* for our indices, because *bytemuck* has already implemented them for basic types like *u16*. This means we can simply add the *index\_buffer* and *indices\_len* to the State struct:

```

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    index_buffer: wgpu::Buffer,
    indices_len: u32,
    model_mat,
    view_mat,
    project_mat,
}

```

And then populate these fields in the constructor:

```
Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    index_buffer,
    indices_len,
    model_mat,
    view_mat,
    project_mat,
}
```

We also need to update the `render()` method to use the `index_buffer`:

```
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_index_buffer(self.index_buffer.slice(..), wgpu::IndexFormat::Uint16);
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw_indexed(0..self.indices_len, 0, 0..1);
```

Note that when using the index buffer, we need to call `draw_indexed` function to create our cube.

The rest of the code and the shader program are the same as those used in our preceding example.

### 6.4.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch06_cube_vertex_color"
path = "examples/ch06/cube_vertex_color.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch06_cube_vertex_color
```

This produces the results shown in Fig.6-6.

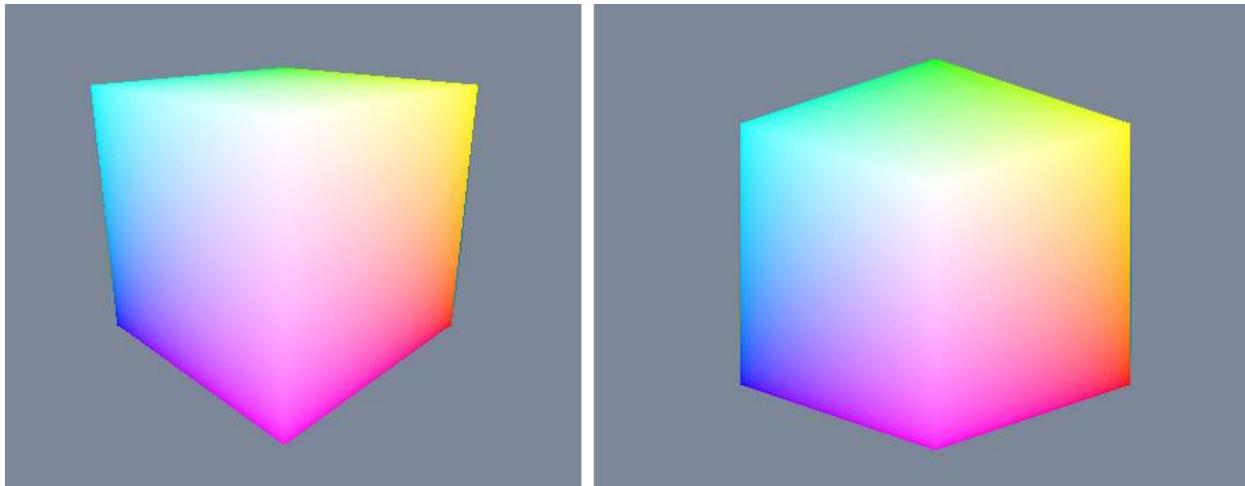


Fig.6-6. Cube with different projections: perspective (left) and orthographic (right).

## 6.5 Rotating Objects

The 3D objects we have created so far are static. In this section, we will take our cube with distinct face colors shown in the previous example and make it to continuously rotate by updating the model matrix. This means that we fix the camera (i.e., view and projection matrices) and only move the object itself (i.e., the model matrix). At the same time, we can also fix the object and move the camera around to achieve the same animation effect.

### 6.5.1 Rust Code

Add a new Rust file called `rotate_cube.rs` to the `examples/ch06/` folder. Most of the code for this file is similar to that used in our `cube_face_color` example, except that we need to replace the `update` function with the following code:

```
const ANIMATION_SPEED:f32 = 1.0;

fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0,0.0,0.0],
        [dt.sin(), dt.cos(), 0.0], [1.0, 1.0, 1.0]);
    let mvp_mat = self.project_mat * self.view_mat * model_mat;
    let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
}
```

This function accepts the time duration `dt` as its input argument, which will be used as our animation parameter. Inside this function, we rescale `dt` with the `ANIMATION_SPEED` constant. We then recreate the model matrix with a modified rotation vector `[dt.sin(), dt.cos(), 0.0]`; that is, we will constantly change the rotation angles around the `x` and `y` axes. Next, we recreate the model-view-projection matrix using this updated model matrix. Finally, we update the `uniform_buffer` with this new model-view-projection matrix by calling the `write_buffer` method.

Another place we need to make changes is within the `main` function:

```
fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}","ch06-rotate_cube"));
    let mut state = pollster::block_on(State::new(&window));

    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
```

```

        state: ElementState::Pressed,
        virtual_keycode: Some(VirtualKeyCode::Escape),
        ..
    },
    ..
} => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {:?}", e, e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

The highlighted code snippet is the new code, which defines the time duration and calls the *update* function to rotate our cube.

## 6.5.2 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

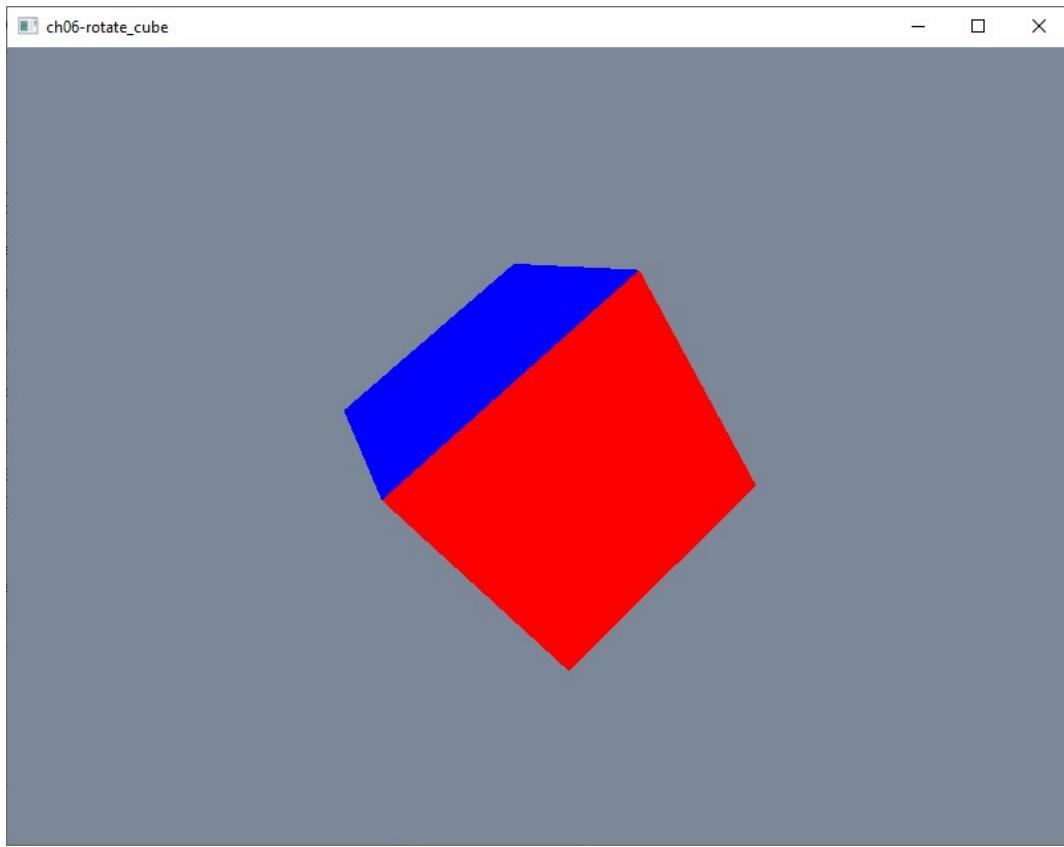
[[example]]
name = "ch06_rotate_cube"
path = "examples/ch06/rotate_cube.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch06_rotate_cube
```

This produces the results shown in Fig.6-7.



*Fig.6-7. Rotating cube.*

## 6.6 Camera Controls

In the preceding 3D examples, we actually created a very simple 3D camera with both perspective and orthographic projections. This is a fixed camera without any user interactions. Even though implementing an interactive camera is not specific to *wgpu* applications, it is still very useful to have camera with some user control functionalities.

Here, I will show you how to rotate a camera using mouse movement. Following the procedure presented here, you can easily add more user controls to the camera like zooming using the mouse wheel or changing camera location using the keyboard inputs.

### 6.6.1 Camera Code

We will place the camera-related code in a separate file. Add a new *camera.rs* file to the *examples/ch06/* folder. Here is the code listing for the *camera.rs* file:

```
use cgmath::*;
use std::f32::consts::PI;

pub struct Camera {
    pub position: Point3<f32>,
    yaw: Rad<f32>,
    pitch: Rad<f32>,
```

```

    }

impl Camera {
    pub fn new<Pt: Into<Point3<f32>>, Yaw: Into<Rad<f32>>, Pitch: Into<Rad<f32>>>(
        position: Pt,
        yaw: Yaw,
        pitch: Pitch,
    ) -> Self {
        Self {
            position: position.into(),
            yaw: yaw.into(),
            pitch: pitch.into(),
        }
    }

    pub fn view_mat(&self) -> Matrix4<f32> {
        Matrix4::look_to_rh(
            self.position,
            Vector3::new(self.pitch.0.cos()*self.yaw.0.cos(), self.pitch.0.sin(),
            self.pitch.0.cos()*self.yaw.0.sin()).normalize(),
            Vector3::unit_y()
        )
    }
}

pub struct CameraController {
    rotatex: f32,
    rotatey: f32,
    speed: f32,
}

impl CameraController {
    pub fn new(speed: f32) -> Self {
        Self {
            rotatex: 0.0,
            rotatey: 0.0,
            speed,
        }
    }

    pub fn mouse_move(&mut self, mousex: f64, mousey: f64) {
        self.rotatex = mousex as f32;
        self.rotatey = mousey as f32;
    }

    pub fn update_camera(&mut self, camera: &mut Camera) {
        camera.yaw += Rad(self.rotatex) * self.speed;
        camera.pitch += Rad(self.rotatey) * self.speed;

        self.rotatex = 0.0;
        self.rotatey = 0.0;
        if camera.pitch < -Rad(89.0*PI/180.0) {
            camera.pitch = -Rad(89.0*PI/180.0);
        } else if camera.pitch > Rad(89.0*PI/180.0) {
            camera.pitch = Rad(89.0*PI/180.0);
        }
    }
}

```

Here, we first create a *Camera* struct that stores the *position*, *yaw* (horizontal rotation), and *pitch* (vertical rotation). We implement a *view\_mat* method, which calls the *Matrix4::look\_to\_rh* function from the *cgmath* library to create the view matrix for our camera.

Next, we create a *CameraController* struct that takes into account inputs from the user's mouse. The *mouse\_move* method relates the mouse movement directly to the camera rotation, which allows you to rotate the camera with your mouse. We use the *update\_camera* method to convert the mouse inputs into camera's, yaw, and pitch parameters, and then use these parameters to update the camera.

## 6.6.2 Rust Code

Add a new Rust file called *camera\_controller.rs* to the *examples/ch06/* folder. Most of the code for this file is similar to that used in the preceding example, but for your reference, I will still provide the full code listing here:

```
use std::iter, mem;
use cgmath::*;
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;
mod camera;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

#[repr(C)]
#[derive(Copy, Clone, bytemuck::Pod, bytemuck::Zeroable)]
struct CameraUniform {
    view_mat: [[f32; 4]; 4],
}

impl CameraUniform {
    fn new() -> Self {
        Self {
            view_mat: cgmath::Matrix4::identity().into(),
        }
    }

    fn update_view_project(&mut self, camera: &camera::Camera, project_mat: Matrix4<f32>) {
        self.view_mat = (project_mat * camera.view_mat()).into()
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 4],
    color: [f32; 4],
}

fn vertex(p:[i8;3], c:[i8; 3]) -> Vertex {
```

```

Vertex {
    position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
    color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
}
}

fn create_vertices() -> Vec<Vertex> {
    let (pos, col, _uv, _normal) = vertex_data::cube_data();
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], col[i]));
    }
    data.to_vec()
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,

    camera: camera::Camera,
    projection: Matrix4<f32>,
    camera_controller: camera::CameraController,
    camera_uniform: CameraUniform,
    camera_buffer: wgpu::Buffer,
    camera_bind_group: wgpu::BindGroup,
    mouse_pressed: bool,
}

impl State {
    async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("cube_face_color.wgsl").into()),
        });

        // uniform data
        let camera = camera::Camera::new((2.0, 3.0, 5.0), cgmath::Deg(-80.0), cgmath::Deg(-30.0));
        let projection = transforms::create_projection(
            init.config.width as f32/init.config.height as f32, true);
        let camera_controller = camera::CameraController::new(0.005);

        let mut camera_uniform = CameraUniform::new();
        camera_uniform.update_view_proj(&camera, projection);

        let camera_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Camera Buffer"),

```

```

contents: bytemuck::cast_slice(&[camera_uniform]),
usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let camera_bind_group_layout = init.device.create_bind_group_layout(
&wgpu::BindGroupLayoutDescriptor{
    entries: &[wgpu::BindGroupLayoutEntry {
        binding: 0,
        visibility: wgpu::ShaderStages::VERTEX | wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    }],
    label: Some("Uniform Bind Group Layout"),
});

let camera_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &camera_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: camera_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&camera_bind_group_layout],
    push_constant_ranges: &[],
});
};

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
},

```

```

//depth_stencil: None,
depth_stencil: Some(wgpu::DepthStencilState {
    format: wgpu::TextureFormat::Depth24Plus,
    depth_write_enabled: true,
    depth_compare: wgpu::CompareFunction::LessEqual,
    stencil: wgpu::StencilState::default(),
    bias: wgpu::DepthBiasState::default(),
}),
multisample: wgpu::MultisampleState::default(),
});

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&create_vertices()),
    usage: wgpu::BufferUsages::VERTEX,
});

Self {
    init,
    pipeline,
    vertex_buffer,
    camera,
    projection,
    camera_controller,
    camera_buffer,
    camera_bind_group,
    camera_uniform,
    mouse_pressed: false
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.projection = transforms::create_projection(
            new_size.width as f32/new_size.height as f32, true);
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
    }
}

fn input(&mut self, event: &DeviceEvent) -> bool {
    match event {
        DeviceEvent::Button {
            button: 1, // Left Mouse Button
            state,
        } => {
            self.mouse_pressed = *state == ElementState::Pressed;
            true
        }
        DeviceEvent::MouseMotion { delta } => {
            if self.mouse_pressed {
                self.camera_controller.mouse_move(delta.0, delta.1);
            }
            true
        }
        _ => false,
    }
}

```

```

fn update(&mut self) {
    // UPDATED!
    self.camera_controller.update_camera(&mut self.camera);
    self.camera_uniform
        .update_view_projection(&self.camera, self.projection);
    self.init.queue.write_buffer(
        &self.camera_buffer,
        0,
        bytemuck::cast_slice(&[self.camera_uniform]),
    );
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format: wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
                view: &depth_view,
                depth_ops: Some(wgpu::Operations {
                    load: wgpu::LoadOp::Clear(1.0),

```

```
        store: false,
    }),
    stencil_ops: None,
),
}),
});

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.camera_bind_group, &[]);
render_pass.draw(0..36, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
env_logger::init();
let event_loop = EventLoop::new();
let window = WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&format!("{}", "ch06_camera_control"));
let mut state = pollster::block_on(State::new(&window));
event_loop.run(move |event, _, control_flow| {
    *control_flow = ControlFlow::Poll;
    match event {
        Event::MainEventsCleared => window.request_redraw(),
        Event::DeviceEvent {
            ref event,
            .. // We're not using device_id currently
        } => {
            state.input(event);
        }
        Event::WindowEvent {
            ref event,
            window_id,
        } if window_id == window.id() => {
            //if !state.input(event) {
            match event {
                WindowEvent::CloseRequested
                | WindowEvent::KeyboardInput {
                    input:
                        KeyboardInput {
                            state: ElementState::Pressed,
                            virtual_keycode: Some(VirtualKeyCode::Escape),
                            ..
                        },
                    ..
                } => *control_flow = ControlFlow::Exit,
                WindowEvent::Resized(physical_size) => {
                    state.resize(*physical_size);
                }
                WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                    state.resize(**new_inner_size);
                }
                _ => {}
            }
        }
    //}
})
```

```

        }
        Event::RedrawRequested(_) => {
            state.update();

            match state.render() {
                Ok(_) => {}
                Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
                Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
                Err(e) => eprintln!(":{}?", e),
            }
        }
    } => {}
});
```

Here, we first create a *CameraUniform* struct and implement an *update\_view\_project* method to obtain the view-projection matrix from the *camera* module.

We then add the following camera-related fields to the *State* struct:

```
struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,

    camera: camera::Camera,
    projection: Matrix4<f32>,
    camera_controller: camera::CameraController,
    camera_uniform: CameraUniform,
    camera_buffer: wgpu::Buffer,
    camera_bind_group: wgpu::BindGroup,
    mouse_pressed: bool,
}
```

In the uniform data section, we create *camera* (view matrix), *projection*, and *camera\_controller* with the code snippet:

```
let camera = camera::Camera::new((2.0, 3.0, 5.0), cgmath::Deg(-80.0), cgmath::Deg(-30.0));
let projection = transforms::create_projection(
    init.config.width as f32/init.config.height as f32, true);
let camera_controller = camera::CameraController::new(0.005);
```

We then construct the camera uniform matrix by calling the *update\_view\_project* method we just implemented. The code for creating *camera\_buffer*, *camera\_bind\_group\_layout*, *camera\_bind\_group* and render pipeline are the same as those used in the preceding example.

To use the camera controller, we need to change the code for the *input* methods:

```
fn input(&mut self, event: &DeviceEvent) -> bool {
    match event {
        DeviceEvent::Button {
            button: 1,
            state,
        } => {
            self.mouse_pressed = *state == ElementState::Pressed;
            true
        }
        DeviceEvent::MouseMotion { delta } => {
            if self.mouse_pressed {
```

```

        self.camera_controller.mouse_move(delta.0, delta.1);
    }
    true
}
- => false,
}
}

```

### 6.6.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

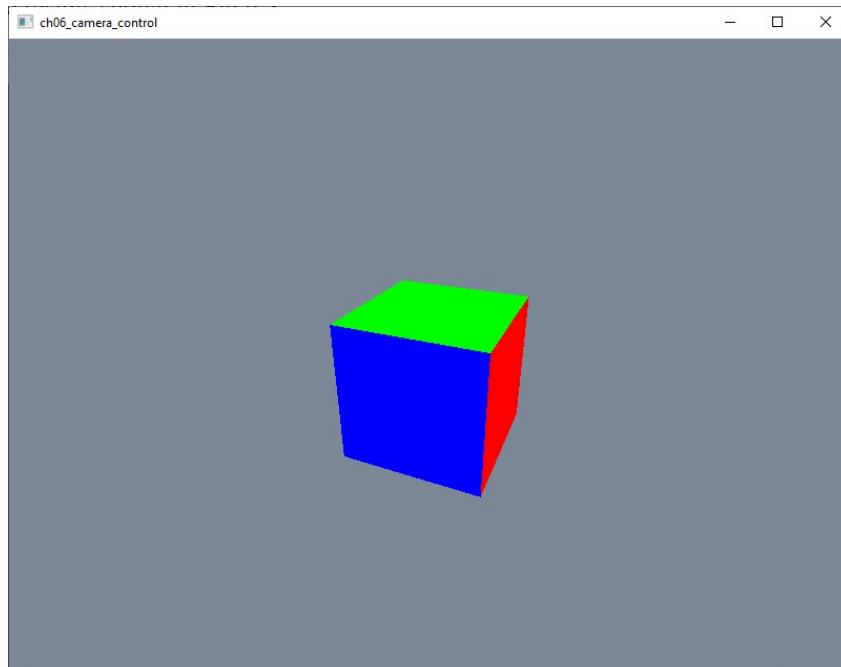
[[example]]
name = "ch06_camera_control"
path = "examples/ch06/camera_control.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch06_camera_control
```

This produces the results shown in Fig.6-8.



*Fg.6-8. Interact with the cube using your mouse or keyboard.*

You can use the mouse to move the camera around.

# 7 3D Wireframe Shapes

A wireframe model is a visual representation of a 3D object used in computer graphics. It is created by drawing just the outlines of the polygons that make up the object. The term “wire frame” comes from designers using metal wire to represent the 3D shape of solid objects.

Using a wireframe model allows for visualization of the underlying structure of a 3D shape. Since wireframe renderings do not require lighting, depth-stencil states, material attributes, or textures, they are relatively simple to construct and fast to calculate. Therefore, wireframes are often used in cases where a high screen frame rate is needed.

In this chapter, I will explain how to create wireframe models in `wgpu` for various 3D shapes, including a cube, sphere, cylinder, cone, and torus. The key to creating 3D wireframe shapes is to specify the correct coordinates for their vertices.

## 7.1 Common Code

You might have noticed in the preceding chapter that there was still a lot of duplicated code when we were creating different 3D objects. To avoid such code duplication, we will now implement a common Rust file that can be reused for creating different 3D wireframes.

Add a new sub-folder called `ch07` to the `examples/` folder, and then add a common Rust file named `common.rs` to this newly created folder. Enter the following content into this file:

```
use std:: {iter, mem};
use cgmath::Matrix4;
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
```

## 132 | Practical GPU Graphics with wgpu and Rust

```
pub struct Vertex {
    position: [f32; 4],
}

pub fn vertex(p:[f32;3]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
    }
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
    num_vertices: u32,
}

impl State {
    async fn new(window: &Window, mesh_data: &Vec<Vertex>) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!(concat!(env!("CARGO_MANIFEST_DIR"),
                "/examples/ch06/line3d.wgsl")).into()),
        });

        // uniform data
        let camera_position = (3.0, 1.5, 3.0).into();
        let look_direction = (0.0,0.0,0.0).into();
        let up_direction = cgmath::Vector3::unit_y();

        let model_mat = transforms::create_transforms([0.0,0.0,0.0], [0.0,0.0,0.0], [1.0,1.0,1.0]);
        let (view_mat, project_mat, view_project_mat) =
            transforms::create_view_projection(camera_position, look_direction, up_direction,
                init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);
        let mvp_mat = view_project_mat * model_mat;

        let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
        let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Uniform Buffer"),
            contents: bytemuck::cast_slice(mvp_ref),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });
    }
}
```

```

let uniform_bind_group_layout =
    init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
        entries: &[wgpu::BindGroupLayoutEntry {
            binding: 0,
            visibility: wgpu::ShaderStages::VERTEX,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Uniform,
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        }],
        label: Some("Uniform Bind Group Layout"),
    });

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::LineList,
        strip_index_format: None,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});

```

});

```

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {

```

```

        label: Some("Vertex Buffer"),
        contents: cast_slice(mesh_data),
        usage: wgpu::BufferUsages::VERTEX,
    });
let num_vertices = mesh_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    uniform_buffer,
    view_mat,
    project_mat,
    num_vertices,
}
}

fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::
            create_projection(new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0],
  [dt.sin(), dt.cos(), 0.0], [1.0, 1.0, 1.0]);
    let mvp_mat = self.project_mat * self.view_mat * model_mat;
    let mvp_ref:&[f32; 16] = mvp_mat.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {

```

```

        view: &view,
        resolve_target: None,
        ops: wgpu::Operations {
            load: wgpu::LoadOp::Clear(wgpu::Color {
                r: 0.2,
                g: 0.247,
                b: 0.314,
                a: 1.0,
            }),
            store: true,
        },
    ],
    depth_stencil_attachment: None,
});

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

pub fn run(mesh_data: &Vec<Vertex>, title: &str) {
env_logger::init();
let event_loop = EventLoop::new();
let window = winit::window::WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&format!("ch07_{}", title));

let mut state = pollster::block_on(State::new(&window, &mesh_data));
let render_start_time = std::time::Instant::now();

event_loop.run(move |event, _, control_flow| {
match event {
    Event::WindowEvent {
        ref event,
        window_id,
    } if window_id == window.id() => {
        if !state.input(event) {
            match event {
                WindowEvent::CloseRequested
                | WindowEvent::KeyboardInput {
                    input:
                        KeyboardInput {
                            state: ElementState::Pressed,
                            virtual_keycode: Some(VirtualKeyCode::Escape),
                            ..
                        },
                    ..
                } => *control_flow = ControlFlow::Exit,
                WindowEvent::Resized(physical_size) => {
                    state.resize(*physical_size);
                }
                WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                    state.resize(**new_inner_size);
                }
            }
        }
    }
})
}

```

```

        }
    }
}

Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {:?}", e, e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
```

}

This code listing is similar to what we used in the *ch06\_line3d* example presented in the previous chapter, but now we are adding a rotation animation as we did in our *ch06\_rotate\_cube* example. In order to use this file to create a variety of 3D wireframes, we make the *Vertex* and *State* structs, as well as the functions inside the *impl State*, public. In particular, the *async new* function now accepts *mesh\_data* as its input argument, which will be different for different wireframes.

Here, to avoid code duplication, we also convert the content of the *main* function in previous examples into a new *run* function. In addition, we use the shader code from the *ch06\_line3d* example. We also change the *primitive.topology* attribute from *LineStrip* to *LineList* within the render pipeline.

## 7.2 Cube Wireframe

In this section, we will create a simple cube wireframe using the *LineList* primitive. In this case, we only need to draw four line segments for the top face, four line segments for the bottom face, and four line segments for the side faces.

### 7.2.1 Rust File

Add a new Rust file called *cube.rs* file to the *examples/ch07/* folder and enter the following code into it:

```
mod common;

fn create_vertices() -> Vec<common::Vertex> {
    // vertex positions
    let p:[[f32; 3]; 8] = [
        [-1.0, 1.0, 1.0],
        [-1.0, 1.0, -1.0],
        [1.0, 1.0, -1.0],
        [1.0, 1.0, 1.0],
        [-1.0, -1.0, 1.0],
```

```

[ -1.0, -1.0, -1.0],
[ 1.0, -1.0, -1.0],
[ 1.0, -1.0,  1.0],
];

// line segments
let lines:[[f32; 3]; 24] = [
    // 4 lines on top face
    p[0], p[1], p[1], p[2], p[2], p[3], p[3], p[0],  

    // 4 lines on bottom face
    p[4], p[5], p[5], p[6], p[6], p[7], p[7], p[4],  

    // 4 lines on sides
    p[0], p[4], p[1], p[5], p[2], p[6], p[3], p[7],  

];
let mut data: Vec<common::Vertex> = Vec::with_capacity(lines.len());
for i in 0..lines.len() {
    data.push(common::vertex(lines[i]));
}
data.to_vec()
}

fn main(){
    let title = "cube";
    let mesh_data = create_vertices();
    common::run(&mesh_data, title);
}

```

Here, we first introduce the mod *common* from the *common.rs* file in the same folder. We then implement a *create\_vertices* method in which we define a point array object that contains eight vertices. These vertices define the coordinates of a cube in 3D space. We then rearrange that vertex array to create a new [[f32; 3]; 24] array that includes 24 vertex positions. This method returns a vector of our *Vertex* struct that is ready for creating our cube wireframe.

Next, we implement the *main* function where we create the *mesh\_data* for our cube wireframe by calling the *create\_vertices* method. We then call the *run* function to create our cube wireframe.

## 7.2.2 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

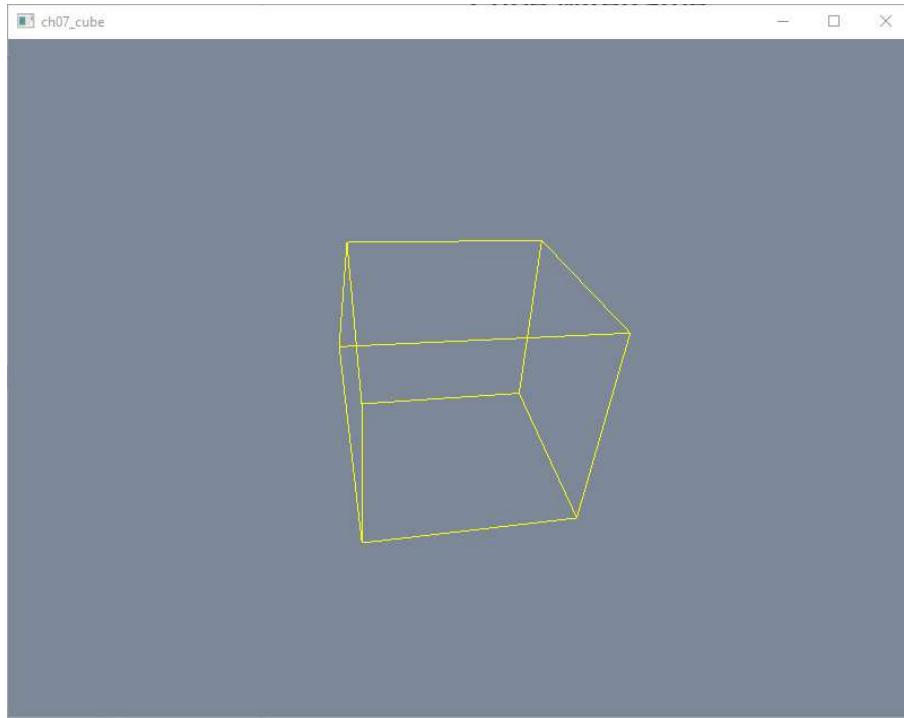
[[example]]
name = "ch07_cube"
path = "examples/ch07/cube.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch07_cube
```

This produces the results shown in Fig.7-1.



*Fig.7-1. Cube wireframe.*

## 7.3 Sphere Wireframe

In this section, we will create a sphere wireframe shape in *wgpu*.

### 7.3.1 Spherical Coordinate System

To do this, we need to be familiar with the spherical coordinate system. A point in the spherical coordinate system is specified by  $r$ ,  $\theta$ , and  $\phi$ . Here,  $r$  is the distance from the point to the origin,  $\theta$  is the polar angle, and  $\phi$  is the azimuthal angle in the  $x$ - $z$  plane from the  $x$ -axis. In our notation, we will alternate the conventional  $y$ - and  $z$ -axes so that the coordinate system is consistent with the one used in *wgpu*. Fig.7-2 shows a point in this coordinate system.

From this figure, we can easily obtain the following relationships:

$$x = r \sin \theta \cos \phi$$

$$y = r \cos \theta$$

$$z = -r \sin \theta \sin \phi$$

In order to create a sphere wireframe in *wgpu* using these relationships, we can start with the familiar concept of longitude and latitude, sometimes called the UV-sphere method. The standard UV-sphere model is made out of  $u$  segments and  $v$  rings, as shown in Fig.7-3. It can be seen that the  $u$  and  $v$  lines form grids on the surface of the sphere. To begin creating a wireframe for this surface, it is enough to consider just one unit grid, as shown on the right side of Fig.7-3.

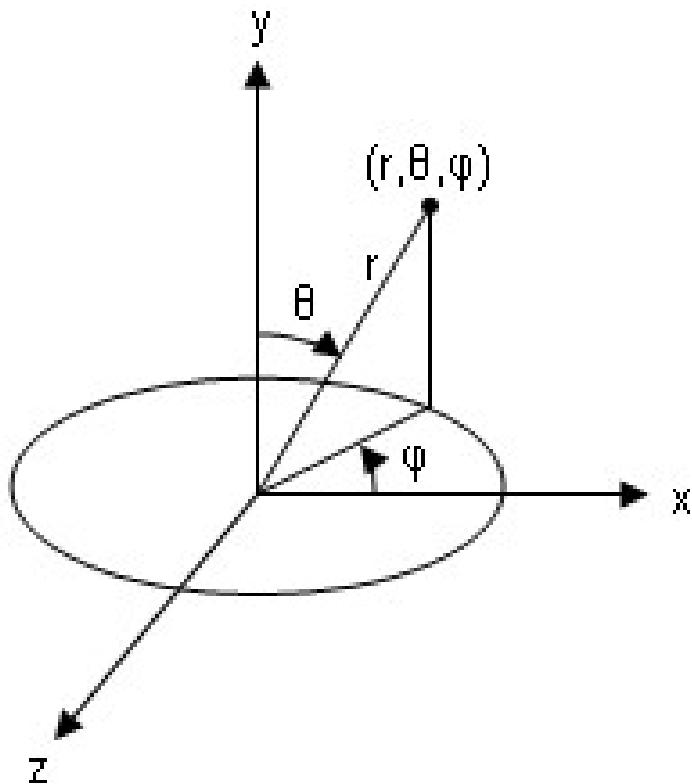


Fig. 7-2. Spherical coordinate system.

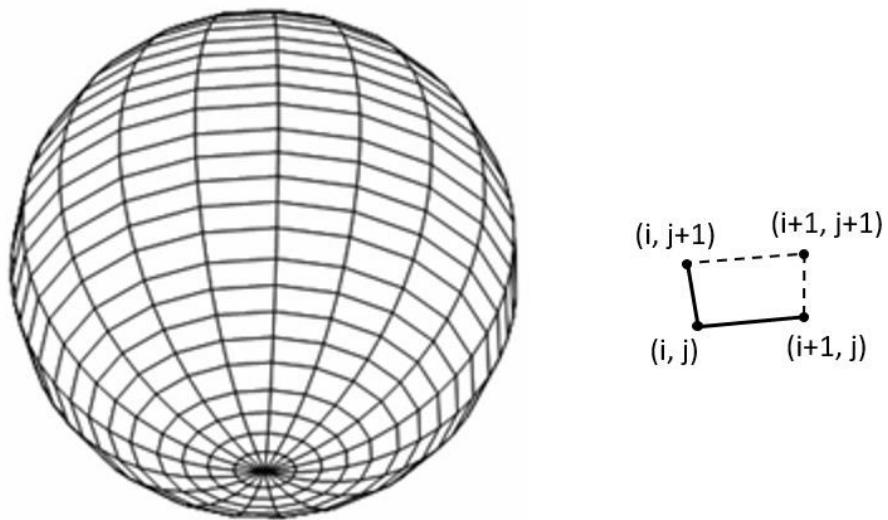


Fig. 7-3. UV-sphere model and a unit grid used to create line segments for a wireframe.

This unit grid can be divided into four line segments with two solid lines and two dashed lines. We only need to draw the two solid line segments because the two dashed lines will be drawn by the other unit grids. This lets us avoid drawing the same line segment multiple times. By putting together all of the grids, we can create a wireframe shape for the entire surface of the sphere.

## 7.3.2 Rust Code

Following what we learned in the preceding section, we can easily create the vertex data for a sphere wireframe. First, add a new Rust file named *math-func.rs* to the *examples/common/* folder, and add a new method, *sphere\_position*, to this file, using the following code:

```
pub fn sphere_position(r:f32, theta:Deg<f32>, phi:Deg<f32>) ->[f32; 3]{
    let snt = theta.sin();
    let cnt = theta.cos();
    let.snp = phi.sin();
    let.cnp = phi.cos();
    [r*snt*cnp, r*cnt, -r*snt*snp]
}
```

This method returns a point on a sphere with the specified  $(r, \theta, \phi)$  coordinates.

Next, add a new Rust file called *sphere.rs* to the *examples/ch07/* folder and enter the following code into it:

```
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{WindowBuilder},
};

use cgmath::*;

mod common;
#[path="../common/math_func.rs"]
mod math_func;

fn create_vertices(r:f32, u: usize, v:usize) -> Vec<common::Vertex> {
    let mut pts: Vec<common::Vertex> = Vec::with_capacity((4* (u - 1)*(v - 1)) as usize);
    for i in 0..u - 1 {
        for j in 0..v - 1 {
            let theta = i as f32 *180.0/(u as f32 - 1.0);
            let phi = j as f32 * 360.0/(v as f32 - 1.0);
            let theta1 = (i as f32 + 1.0) *180.0/(u as f32 - 1.0);
            let phi1 = (j as f32 + 1.0) * 360.0/(v as f32 - 1.0);
            let p0 = math_func::sphere_position(r, Deg(theta), Deg(phi));
            let p1 = math_func::sphere_position(r, Deg(theta1), Deg(phi));
            let p3 = math_func::sphere_position(r, Deg(theta), Deg(phi1));
            pts.push(common::vertex(p0));
            pts.push(common::vertex(p1));
            pts.push(common::vertex(p0));
            pts.push(common::vertex(p3));
        }
    }
    pts.to_vec()
}

fn main(){
    let title = "sphere";
    let mesh_data = create_vertices(17, 15, 20);
    common::run(&mesh_data, title);
}
```

Within the *create\_vertices* method, we construct the wireframe by dividing the sphere surface into segments and rings. We can then specify the number of segments and rings using two integers, *u* and *v*. This method also allows us to specify the radius of the sphere.

Inside the `create_vertices` method, we first obtain vertex positions of the three vertices within a unit cell by calling the `sphere_position` function we just implemented in the `math_func.rs` file. We then create the line segments for the unit cell, where we only draw two line segments:  $p0(i, j)$  to  $p1(i+1, j)$  and  $p0(i, j)$  to  $p3(i, j+1)$ .

Finally, we implement the `main` function where we create the `mesh_data` for our sphere wireframe by calling the `create_vertices` method. We then call the `run` function to create our sphere wireframe.

### 7.3.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch07_sphere"
path = "examples/ch07/sphere.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch07_sphere
```

This produces the results shown in Fig.7-4.

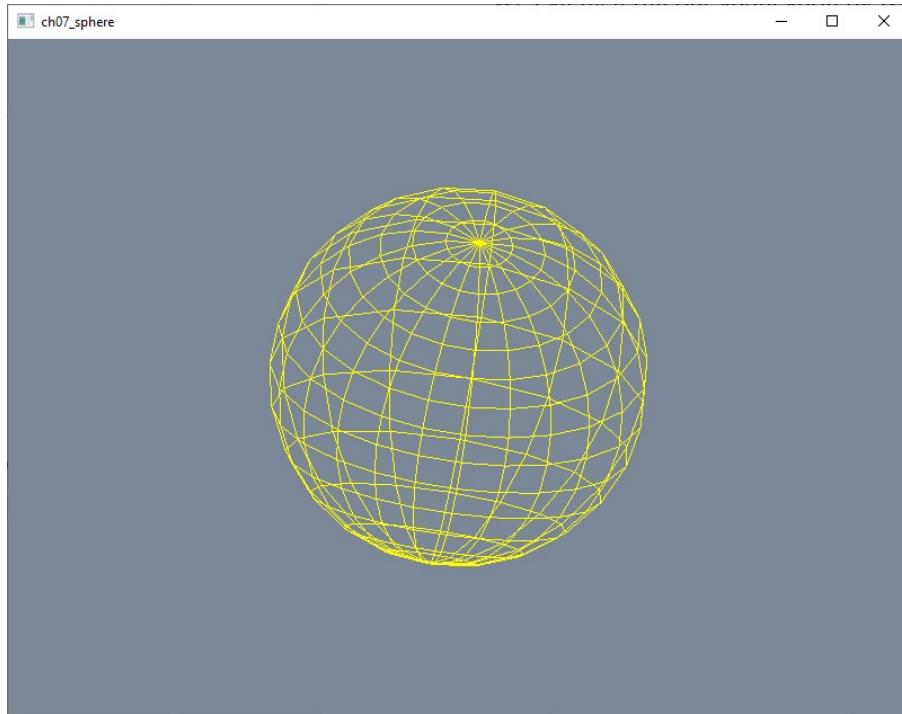


Fig.7-4. Sphere wireframe shape.

## 7.4 Cylinder Wireframe

In this section, I will show you how to create a cylinder wireframe in `wgpu`. We will create a general cylinder shape that allows you to specify its inner and outer radii. By setting a non-zero inner radius, you can create a cylindrical tube shape.

### 7.4.1 Cylindrical Coordinate System

In a cylindrical coordinate system, a point is specified by three parameters,  $r$ ,  $\theta$ , and  $y$ . This is a bit different from the conventional definition of cylindrical coordinates that uses  $r$ ,  $\theta$ , and  $z$ . The notation we use here is only for convenience, since a computer screen can always be described using the  $x$ - $y$  plane. Here  $r$  is the distance of a projected point on the  $x$ - $z$  plane from the origin, and  $\theta$  is the azimuthal angle.

Fig.7-5 shows a point in the cylindrical coordinate system.

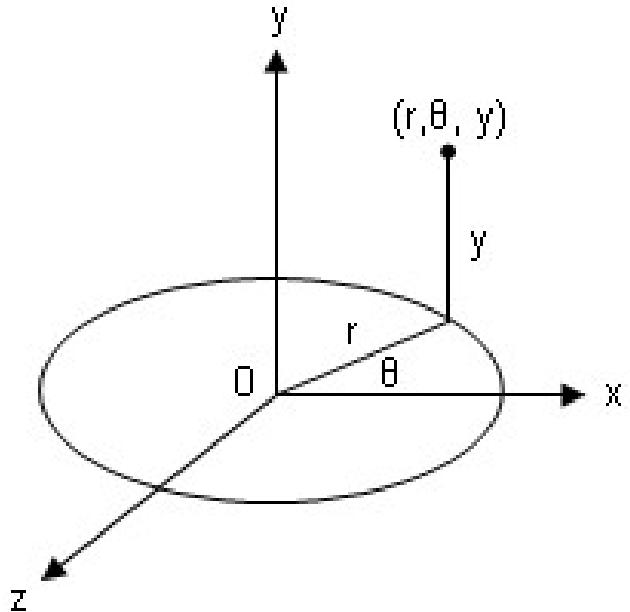


Fig.7-5. Cylindrical coordinate system.

From this figure, we have:

$$x = r \cos \theta$$

$$z = -r \sin \theta$$

$$y = y$$

By using the cylindrical coordinate system, we can easily create cylindrical objects in *wgpu*. First, we need to make slices on the surface of the cylinder. As shown in Fig.7-6, the cylinder surface can be divided into  $n$  slices, and a unit cell is formed by the  $i$ -th and  $i+1$ -th slice lines. You can see that each unit contains eight vertices and four surfaces. There are eight line segments that need to be drawn for each unit cell.

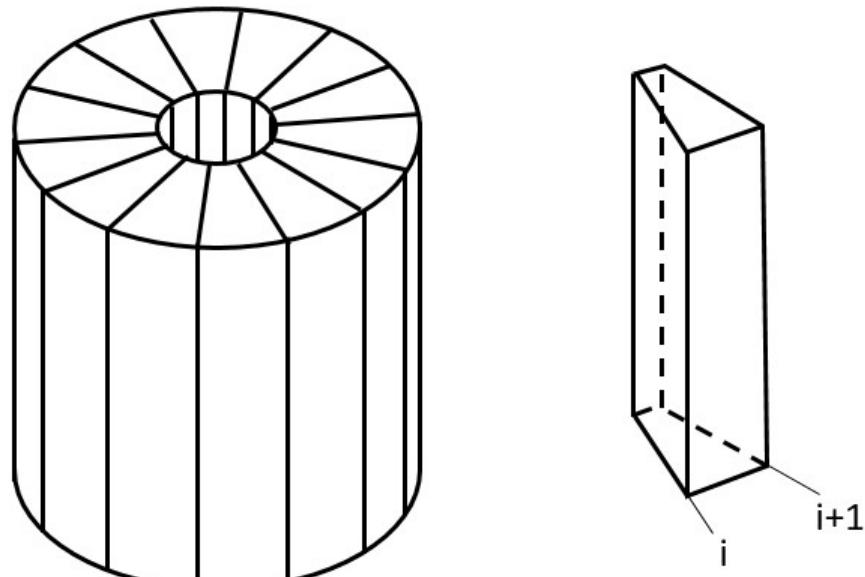


Fig. 7-6. A cylinder and a unit cell.

#### 7.4.2 Rust Code

With the background information described in the preceding section, we can create the vertex data for a cylinder wireframe. Add a new function called *cylinder\_position* to the *math\_func.rs* file with the following code:

```
pub fn cylinder_position(r:f32, y:f32, theta:Deg<f32>) -> [f32; 3] {
    [r*theta.cos(), y, -r*theta.sin()]
}
```

This method returns the position on the surface of the cylinder specified by the given parameters  $r$ ,  $\theta$ , and  $y$  in cylindrical coordinates.

Add a new Rust file called *cylinder.rs* to the *examples/ch07/* folder and type the following code into it:

```
use cgmath::*;

mod common;
#[path="../common/math_func.rs"]
mod math_func;

fn create_vertices(rin:f32, rout:f32, height:f32, n:usize) -> Vec<common::Vertex> {
    let h = height / 2.0;
    let mut pts:Vec<common::Vertex> = Vec::with_capacity(16*(n-1));

    for i in 0..n-1 {
        let theta = i as f32 * 360.0/(n as f32 - 1.0);
        let theta1 = (i as f32 + 1.0) * 360.0/(n as f32 - 1.0);
        let p0 = math_func::cylinder_position(rout, h, Deg(theta));
        let p1 = math_func::cylinder_position(rout, -h, Deg(theta));
        let p2 = math_func::cylinder_position(rin, -h, Deg(theta));
        let p3 = math_func::cylinder_position(rin, h, Deg(theta));
        let p4 = math_func::cylinder_position(rout, h, Deg(theta1));
        let p5 = math_func::cylinder_position(rout, -h, Deg(theta1));
```

```

let p6 = math_func::cylinder_position(rin, -h, Deg(theta1));
let p7 = math_func::cylinder_position(rin, h, Deg(theta1));

// top face 3 lines
pts.push(common::vertex(p0));
pts.push(common::vertex(p3));
pts.push(common::vertex(p3));
pts.push(common::vertex(p7));
pts.push(common::vertex(p4));
pts.push(common::vertex(p0));

// bottom face 3 lines
pts.push(common::vertex(p1));
pts.push(common::vertex(p2));
pts.push(common::vertex(p2));
pts.push(common::vertex(p6));
pts.push(common::vertex(p5));
pts.push(common::vertex(p1));

// side 2 lines
pts.push(common::vertex(p0));
pts.push(common::vertex(p1));
pts.push(common::vertex(p3));
pts.push(common::vertex(p2));
}

pts.to_vec()
}

fn main(){
    let title = "cylinder";
    let mesh_data = create_vertices(0.0, 1.0, 2.5, 20);
    common::run(&mesh_data, title);
}

```

The `create_vertices` method is used to generate the vertex data for our cylinder wireframe by specifying the inner and outer radii, the height, and the position of our cylinder. Inside this method, we first get the eight vertex positions within a unit grid by calling the `cylinder_position` function implemented in the `math_func.rs` file. We need to draw eight line segments for each unit cell, as highlighted by the thick-solid lines in Fig.7-7, but we do not need to draw the four dashed line segments on the  $i+1$  surface in order to avoid drawing them multiple times.

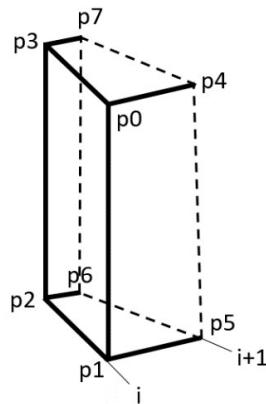


Fig.7-7. Unit cell for a cylinder wireframe. Only eight thick line segments need to be drawn.

If we run this method over all of the unit cells, we can create a wireframe shape for the entire surface of the cylinder.

Finally, we implement the *main* function where we create the *mesh\_data* for our cylindrical wireframe by calling the *create\_vertices* method. We then call the *run* function to create our cylindrical wireframe.

### 7.4.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch07_cylinder"
path = "examples/ch07/cylinder.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch07_cylinder
```

This produces the results shown in Fig.7-8.

If we set the *rin* parameter to zero when calling the *create\_vertices* method:

```
let mesh_data = create_vertices(0.0, 1.0, 1.2, 20);
```

we will get a solid regular cylinder as shown in Fig.7-9.

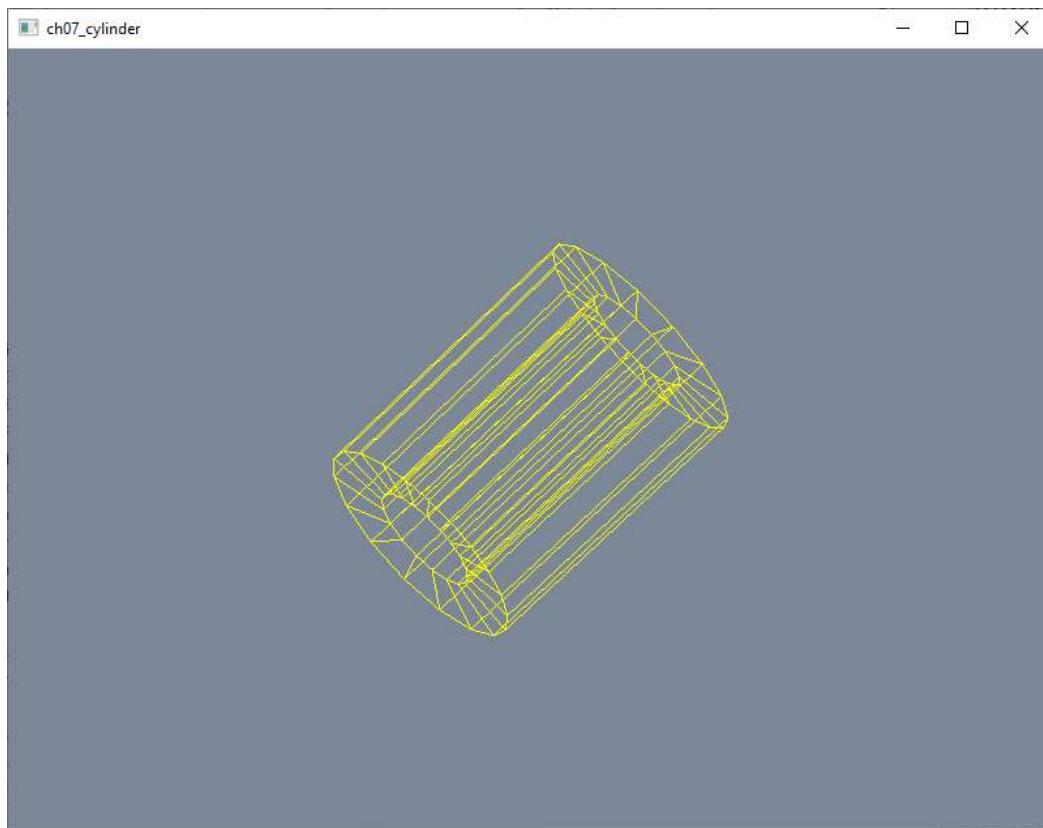
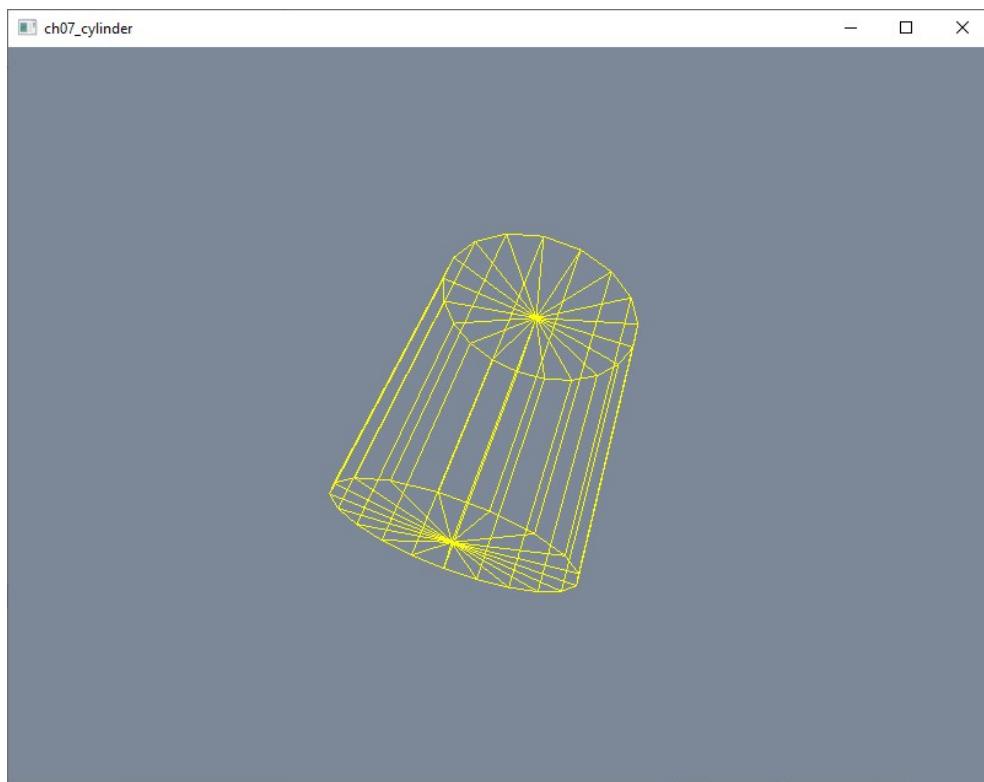


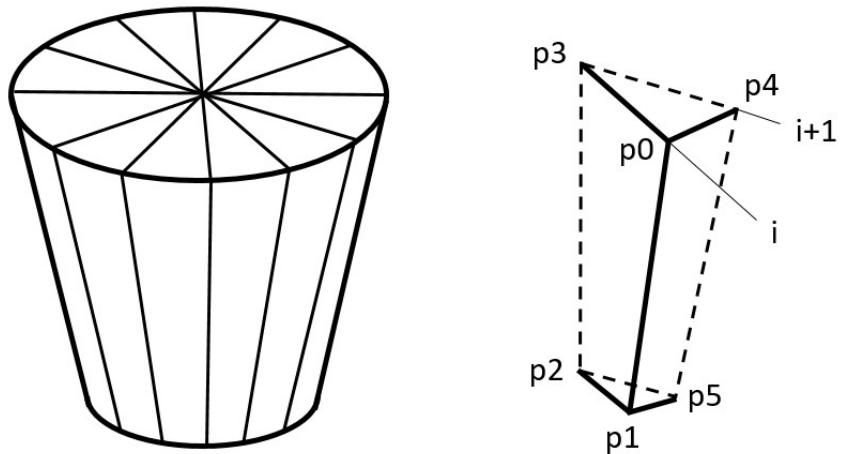
Fig.7-8. Cylinder wireframe shape.



*Fig. 7-9. Regular cylinder wireframe with  $rin = 0$ .*

## 7.5 Cone Wireframe

We can also create a cone wireframe shape using the cylindrical coordinate system. Here, we want to create a general cone shape whose top radius, bottom radius, and height can all be specified. Fig.7-10 illustrates how slices are made on the surface of the cone and how a unit cell is defined.



*Fig. 7-10. A cone shape and a unit cell.*

This unit cell contains six vertices and only five line segments need to be drawn for each unit cell, as highlighted by the solid lines in the figure.

## 7.5.1 Rust Code

With the information presented in Fig.7-10, we can now create the vertex data for our cone wireframe. Here, we can use the same math function we used for the cylinder wireframe because the cone shape can be described using the cylindrical coordinate system.

Add a new Rust file called *cone.ts* to the *examples/sh07/* folder and type the following code into it:

```
use cgmath::*;

mod common;
#[path="../common/math_func.rs"]
mod math_func;

fn create_vertices(rtop:f32, rbottom:f32, height:f32, n:usize) -> Vec<common::Vertex> {
    let h = height / 2.0;
    let mut pts:Vec<common::Vertex> = Vec::with_capacity(10*(n-1));

    for i in 0..n-1 {
        let theta = i as f32 * 360.0/(n as f32 - 1.0);
        let theta1 = (i as f32 + 1.0) *360.0/(n as f32 - 1.0);
        let p0 = math_func::cylinder_position(rtop, h, Deg(theta));
        let p1 = math_func::cylinder_position(rbottom, -h, Deg(theta));
        let p2 = math_func::cylinder_position(0.0, -h, Deg(theta));
        let p3 = math_func::cylinder_position(0.0, h, Deg(theta));
        let p4 = math_func::cylinder_position(rtop, h, Deg(theta1));
        let p5 = math_func::cylinder_position(rbottom, -h, Deg(theta1));

        // top face 2 lines
        pts.push(common::vertex(p0));
        pts.push(common::vertex(p3));
        pts.push(common::vertex(p4));
        pts.push(common::vertex(p0));

        // bottom face 2 lines
        pts.push(common::vertex(p1));
        pts.push(common::vertex(p2));
        pts.push(common::vertex(p5));
        pts.push(common::vertex(p1));

        // side 1 line
        pts.push(common::vertex(p0));
        pts.push(common::vertex(p1));
    }

    pts.to_vec()
}

fn main(){
    let title = "cone";
    let mesh_data = create_vertices(0.0, 1.5, 2.0, 20);
    common::run(&mesh_data, title);
}
```

The `create_vertices` method allows us to generate the vertex data for our cone wireframe by specifying the top and bottom radii and height of the cone. Note that we set the radius to zero when we call the `cylinder_position` function to generate the vertex positions on the central axis:

```
let p2 = math_func::cylinder_position(0.0, -h, Deg(theta));
let p3 = math_func::cylinder_position(0.0, h, Deg(theta));
```

The rest of the code is similar to that used in our previous cylinder wireframe example.

## 7.5.2 Run Application

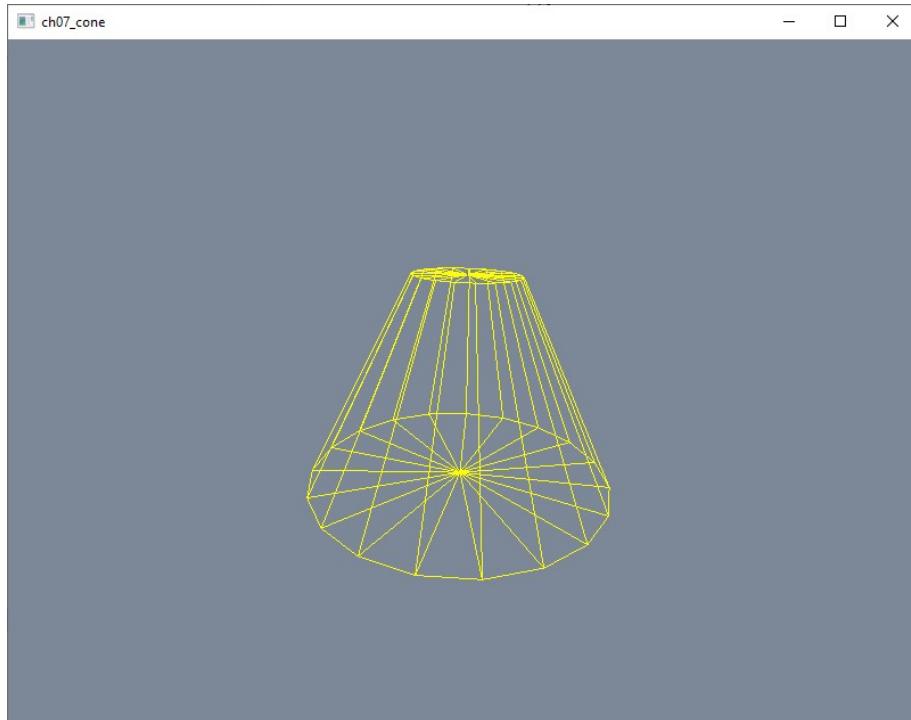
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch07_cone"
path = "examples/ch07/cone.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch07_cone
```

This produces the results shown in Fig.7-11.



*Fig.7-11. Cone wireframe with default parameters.*

You can create various cone wireframe shapes by specifying different sets of parameters. For example, you can create a cylinder by setting the top and bottom radii to be equal; a regular cone by setting one of the radii to zero; a truncated cone by setting both radii to be finite; or even a pyramid by setting the number of slices to a small integer such as four. Fig.7-12 shows a regular cone and a pyramid.

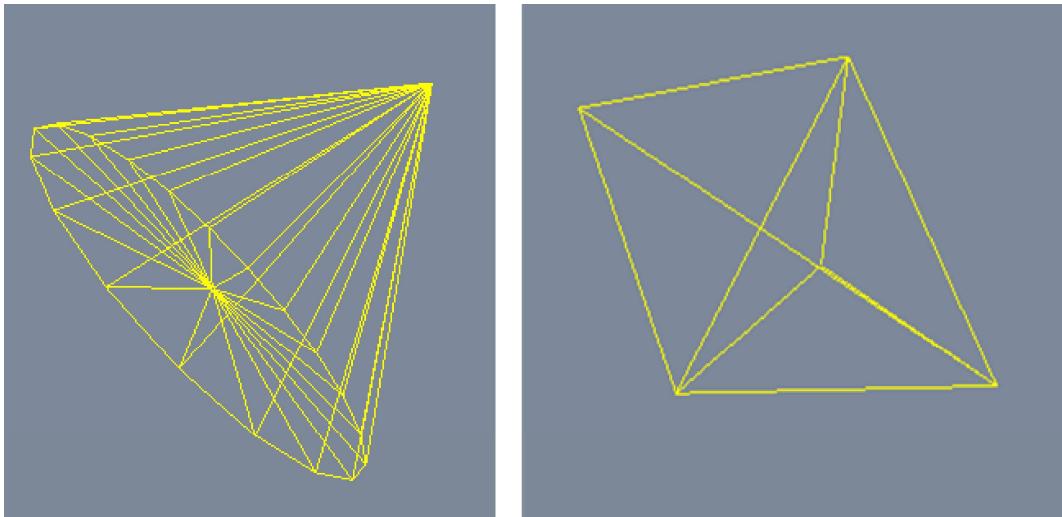


Fig.7-12. Regular cone and pyramid wireframe shapes created by changing cone parameters.

## 7.6 Torus Wireframe

Another popular 3D shape is the torus. A torus is a surface revolution generated by revolving a circle in 3D space about a specified axis. It can be defined using the following parametrized equations:

$$\begin{aligned}x &= (R + r \cos v) \cos u \\y &= r \sin v \\z &= -(R + r \cos v) \sin u\end{aligned}$$

where  $u$  and  $v$  are angles defined in the range  $[0, 2\pi]$ ,  $R$  is the torus radius representing the distance from the center of the tube to the center of the torus, and  $r$  is the radius of the tube.

In order to construct a torus wireframe, we need to divide its surface using tube rings and torus rings, as shown in Fig.7-13.

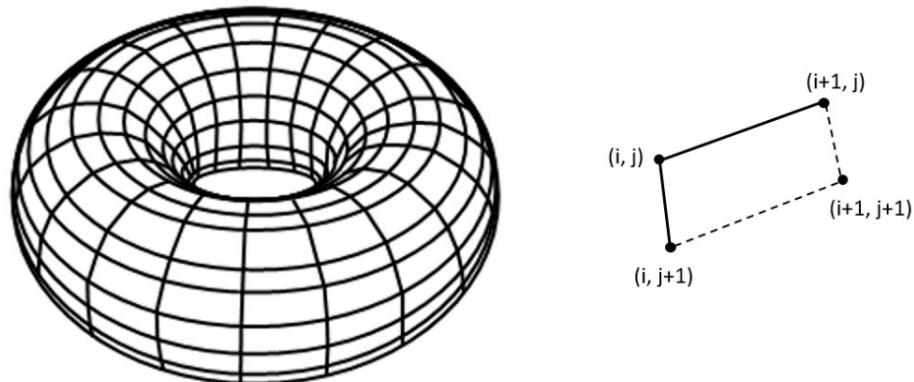


Fig.7-13. A torus and a unit cell.

## 150 | Practical GPU Graphics with wgpu and Rust

This unit cell can be divided into four line segments with two solid lines and two dashed lines. We only need to draw the two solid line segments because the dashed lines will be drawn by the other unit cells. This way, we avoid drawing the same line segment multiple times. By iterating over all of the unit cells, you can create a wireframe shape for the entire surface of the torus.

### 7.6.1 Rust Code

With the background information presented in Fig.7-13, we can now create the vertex data for our torus wireframe. Add a new method called *torus\_position* to the *math\_func.rs* file in the *examples/common/* folder with the following code:

```
pub fn torus_position(r_torus:f32, r_tube:f32, u:Deg<f32>, v: Deg<f32>) -> [f32; 3] {
    let x = (r_torus + r_tube * v.cos())*u.cos();
    let y = r_tube*v.sin();
    let z = -(r_torus + r_tube * v.cos())*u.sin();
    [x, y, z]
}
```

This method returns a point on the surface of the torus specified by the *r\_torus*, *r\_tube*, *u*, and *v* parameters.

Add a new Rust file called *torus.rs* to the *examples/ch07/* folder and enter the following code into it:

```
use cgmath::*;

mod common;
#[path = "../common/math_func.rs"]
mod math_func;

fn create_vertices(r_torus:f32, r_tube:f32, n_torus: usize, n_tube:usize) -> Vec<common::Vertex> {
    let mut pts: Vec<common::Vertex> = Vec::with_capacity((4*(n_torus - 1)*(n_tube - 1)) as usize);
    for i in 0..n_torus - 1 {
        for j in 0..n_tube - 1 {
            let u = i as f32 * 360.0/(n_torus as f32 - 1.0);
            let v = j as f32 * 360.0/(n_tube as f32 - 1.0);
            let u1 = (i as f32 + 1.0) * 360.0/(n_torus as f32 - 1.0);
            let v1 = (j as f32 + 1.0) * 360.0/(n_tube as f32 - 1.0);
            let p0 = math_func::torus_position(r_torus, r_tube, Deg(u), Deg(v));
            let p1 = math_func::torus_position(r_torus, r_tube, Deg(u1), Deg(v));
            let p3 = math_func::torus_position(r_torus, r_tube, Deg(u), Deg(v1));

            pts.push(common::vertex(p0));
            pts.push(common::vertex(p1));
            pts.push(common::vertex(p0));
            pts.push(common::vertex(p3));
        }
    }
    pts.to_vec()
}

fn main(){
    let title = "torus";
    let mesh_data = create_vertices(1.5, 0.3, 40, 13);
    common::run(&mesh_data, title);
}
```

Inside the `create_vertices` method, we construct the wireframe by dividing the torus surface into large and small rings. We then specify the number of segments and rings using two integers, `n_torus` and `n_tube`. This function also allows you to specify the radii (`r_torus` and `r_tube`) of the torus.

Notice that when we create the line segments for the unit grid inside the `create_vertices` method, we only need to draw two line segments:  $p0(i, j)$  to  $p1(i+1, j)$  and  $p0(i, j)$  to  $p3(i, j+1)$ .

## 7.6.2 Run Application

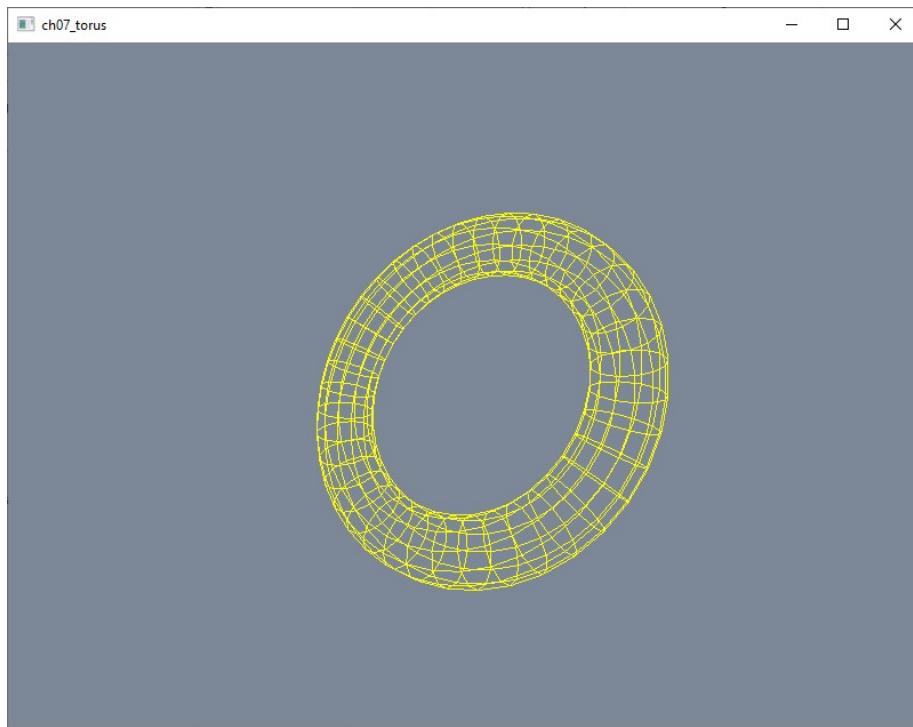
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch07_torus"
path = "examples/ch07/torus.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch07_torus
```

This produces the results shown in Fig.7-14.



*Fig. 7-14. Torus created using default parameters.*

Following the procedure presented in this chapter, you can easily create different 3D wireframe shapes simply by specifying your own mathematical functions to generate the mesh data.

In this chapter, you learned how to create various 3D wireframe shapes in `wgpu`. In the next chapter, I will explain the lighting and materials that will allow you to create real world 3D shapes.



# 8 Lighting in WGPU

In the preceding chapters, we created some simple primitives and various 3D wireframe shapes. However, so far we have only used solid colors or simple gradients when rendering our objects, resulting in images that often look flat rather than truly three-dimensional. This is because up until now, we have neglected the interaction between light and surfaces in our objects – yet this aspect is critical if we want to create a scene that is more recognizably 3D and realistic.

Despite lighting being one of the most important factors in creating real-world shaded 3D graphics objects, *wgpu*, like WebGL, does not have many built-in lighting features. In fact, it really only runs two functions – a vertex shader and a fragment shader. This means that if you want lighting effects in your 3D scene, you will have to write your own functions and create your own lighting model.

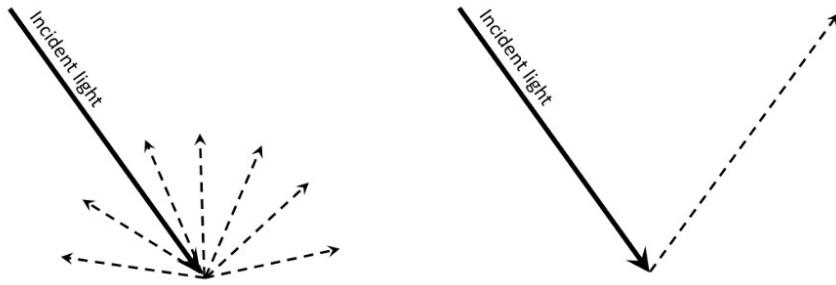
There are several lighting models used in 3D computer graphics. The most modern 3D lighting systems use physically based rendering (PBR) to mimic the real-life behavior of light. Ray tracing is a technique used to model light transport by simulating light as particles that travel in straight lines and bounce off objects. These lighting models take advantage of today's real-time photorealistic rendering technology, and are becoming more widespread as such technology advances. Most recently, hardware acceleration for real-time ray tracing has become standard on new commercial graphics cards, allowing developers to use hybrid ray tracing and rasterization based rendering in games and other real-time applications with a lesser hit to frame render times.

Since PBR and ray-tracing models are often too computationally expensive for most real-time applications, in this book, we will use a less accurate but more efficient model based on the Phong or Blinn-Phong reflection model. In this chapter, you will learn how to build a simple Blinn-Phong lighting model in *wgpu* and how to use it to simulate light sources and the way light interacts with the objects in your scene. I will start by discussing three light sources: ambient light, diffuse light, and specular light.

## 8.1 Light Components

Ambient, diffuse, and specular lights are all different components of lighting. Ambient light is light that illuminates all objects uniformly regardless of their location or orientation. It is the global illumination in an environment.

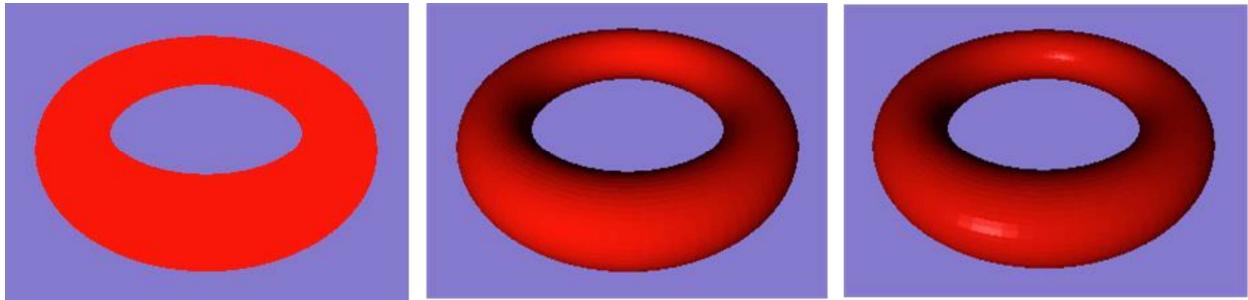
Diffuse lighting and specular lighting reflections depend on the angle of the light hitting a surface. The difference between diffuse and specular reflection is that once light hits a surface, diffuse reflection occurs in all directions, while specular reflection occurs in a single direction, as shown in Fig.8-1.



*Fig.8-1. Diffuse reflection (left) and specular reflection (right).*

Diffuse lighting is the major reflective component in reflection. Matte paint and many natural materials are diffuse reflectors. Perfect diffuse surfaces scatter light equally in all directions. On the other hand, specular reflection (also known as specular highlight) produces a shiny glossiness because most of the light is reflected or scattered within a narrow range of the angle of reflection. The reflected light from a given angle emerges at a single angle, following the rule that the angle of reflection is equal to the angle of incidence.

Fig.8-2 shows the effects of lighting from different light sources on a torus surface.



*Fig.8-2. Light reflection on a torus surface from different light sources: ambient (left), diffuse (center), and specular (right).*

You can see from Fig.8-2 that the reflected light depends on the properties of both surface and the light. But it also depends on the angle at which the light hits the surface. The angle of incidence is essential to specular reflection and affects diffuse reflection too. This angle is always associated with the surface normal that is perpendicular to the face at that vertex.

## 8.2 Normal Vectors

To build a lighting model, we need to calculate the surface normal based on the direction the surface is facing. When used in a lighting calculation, the normal must be normalized to a unit vector and it must be specified for each vertex.

Since the surface of a 3D object can be curved, it can face different directions at different vertices. So, the normal vector is different for different vertices, that is, a normal vector is always associated with a particular point on a surface. In *wgpu*, we can assign normal vectors to the vertices of a surface and then use them to perform lighting calculations for the entire surface.

In the following subsections, I will show you how to calculate the surface normal for several simple 3D shapes, including a cube, sphere, cylinder, and a general polyhedral surface.

### 8.2.1 Surface Normal of a Cube

If you have a unit cube, then each normal simply points outwards from each respective face of the cube. For example, the front face oriented toward the positive  $z$  direction has four vertices:  $(-1, 1, 1)$ ,  $(-1, -1, 1)$ ,  $(1, -1, 1)$ , and  $(1, 1, 1)$ . All of these vertices have the same surface normal, which is equal to the unit vector along the positive  $z$ -direction:  $(0, 0, 1)$ . In the same way, you can easily calculate the surface normal for the other five surfaces of the cube.

### 8.2.2 Surface Normal of a Sphere

Computing the surface normal of a sphere is straightforward. For a unit sphere, the surface normal of any point  $(x, y, z)$  on the unit sphere is simply  $(x, y, z)$ . That is, the normal data for a unit sphere is identical to its vertex data. In the spherical coordinate system, the normal vector at point  $(x, y, z)$  can be expressed in the form:

$$\begin{aligned} x &= \sin \theta \\ y &= \cos \theta \\ z &= -\sin \theta \sin \varphi \end{aligned}$$

### 8.2.3 Surface Normal of a Cylinder

A cylinder surface can be divided into three parts: the top face, the bottom face, and the round tube. The surface normals for the round tube all point outward around the  $x$ - $z$  plane. If we slice the tube into  $n$  equal parts, there are  $n$  such vertices with  $y = 1$  and  $y = -1$ . For any one of these vertices at point  $P(x, y, z)$ , its normal is  $(x, 0, z)$ . In the cylindrical coordinate system, it can be written in the form  $(\cos \theta, 0, -\sin \theta)$ .

For the vertices on the top face, the surface normals are all equal to  $(0, 1, 0)$ , while for the bottom face, the surface normals are all equal to  $(0, -1, 0)$ .

### 8.2.4 Surface Normal of a Polyhedral Surface

For a general polyhedral surface consisting of faces and vertices, we can compute the surface normals in two steps: first, we compute the weighted face normals, and then we compute the surface normals from the weighted face normals.

To compute the weighted face normals, we calculate a normal vector for each face that is weighted by the area of that face. Faces with larger areas get more weight. We do this by looping around the face and taking every three vertices in succession. We actually do the cross product for these three vertices using the following pseudo code:

```
vec3 compute_triangle_normal(vec3 a, vec3 b, vec3 c) {
    return normalize(cross_product(b-c, c-a));
}
```

Note that the length of the product is proportional to the area inside of the triangle  $abc$ . Therefore, surface normal at a vertex shared by several triangles can be computed by summing up all of these cross products

and normalizing the result. The following pseudo code shows how to calculate the surface normal for a general polyhedral surface:

```
vec3 compute_surface_normal(vextex v) {
    vec3 sum = vec3(0,0,0);
    list<vertex> adjacent_vertices = get_adjacent_vertice(a);
    for(var i = 1; i < adjacent_vertices.length; ++i){
        vec3 b = adjacent_vertices[i-1];
        vec3 c = adjacent_vertices[i];
        sum += cross_product(b-a, c-a);
    }
    if(normalize(sum == 0)) return sum;
    return normalize(sum);
}
```

If the polyhedral surface is a quadrilateral (or quad), the surface normal calculation becomes much easier. For a quad  $abcd$ , its surface normal can be calculated using the following simple formula:

```
normalize(cross_product(c-a, d-b));
```

Note that the four vertices of the quad,  $abcd$ , must be arranged in counterclockwise order.

## 8.3 Lighting Calculation

Once we have calculated the normal vector at a vertex, we can use the normal and the direction of the light source to compute the lighting intensity. The lighting intensity then contributes to the final color of the pixel using the formula:

```
vec4 final_color = vec4(0.0, 0.0, 0.0, 1.0);
vec3 color = vec3(r, g, b);
final_color = vec4(color*intensity, 1.0);
```

The *wgpu* API clamps colors written to the output image to the range [0.0, 1.0]. If the lighting intensity, by itself or in combination with other lights, is greater than 1.0, it may result in undesired visual effects.

### 8.3.1 Diffuse Light

The diffuse light intensity equals the cosine between the vector pointing towards the light ( $L$ ) and the vector normal ( $N$ ) to the surface:

$$\cos(\alpha) = \text{normalize}(L \cdot N)$$

We can compute the diffuse intensity  $I_d$  by adding the diffuse component of the object's material  $K_d$ :

$$I_d = K_d * \max[\cos(\alpha), 0]$$

Here, the max function is used to avoid negative values of  $\cos(\alpha)$ .

We can also easily add ambient light of intensity  $I_a$  to the above, given that ambient light is a uniform light that does not depend on the direction:

$$I_d = \max\{K_d * \max[\cos(\alpha), 0], I_a\}$$

### 8.3.2 Specular Light

The simplest model of specular light is the Phong model. In this model, the light reflected in the direction of the viewer varies based on the angle between the view direction and the direction of perfect reflection. Mathematically, the Phong model can be expressed in the form:

$$\text{Phong term} \sim (V \cdot R)^s$$

Where  $V$  and  $R$  represent the view direction and light reflection direction respectively, and  $s$  is the specular exponent that represents the material property known as shininess or roughness. A rougher-looking surface will have a small  $s$ , while a shinier surface will have a large  $s$ .

Note that the Phong model is not based on anything in the real world. All it does is create a bright circular area on the surface that gets dimmer the farther away the viewer is from the direction of perfect reflection. The formula is only valid if the dot product is greater than zero; otherwise, the specular contribution is zero.

The problem with the Phong model is that it requires the angle between the view direction and the reflection direction to be less than 90 degrees in order for the specular Phong term to contribute. To address this issue, we can use the so-called Blinn-Phong model to compute the specular light. The Blinn-Phong model uses a different set of vectors that are based on the half-angle vector. The half-angle vector is the direction halfway between the view direction and the light vector, as shown in Fig.8-3.

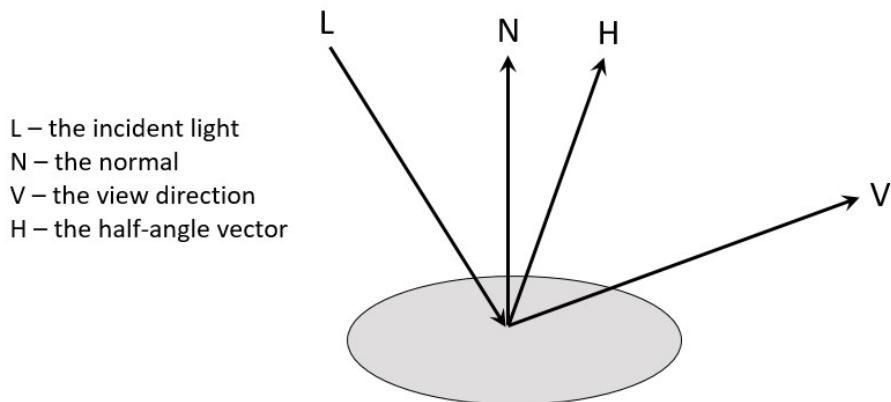


Fig.8-3. Blinn-Phong model based on the half-angle vector.

The half-angle vector  $H$  can be expressed by

$$H = \frac{L - V}{|L - V|}$$

When the view direction and reflection direction are perfectly aligned, the half-angle vector perfectly aligns with the surface normal. Therefore, instead of comparing the reflection vector to the view direction, the Blinn-Phong model compares the half-angle vector to the surface normal. This is then raised to the power of specular exponent  $s$ :

$$\text{Blinn-Phong term} \sim (N \cdot H)^s$$

Since the angle between the half-angle vector and the normal is always less than 90 degrees, the Blinn-Phong model produces results similar to the Phong model, but without the problems associated with the Phong model.

## 8.4 Lighting in Shaders

In this section, I will show you how to implement a simple lighting model in a shader program. This model takes into account the ambient light, the diffuse light, and the specular light based on the Blinn-Phong model.

### 8.4.1 Transform Normals

As mentioned previously, normal vectors are essential for lighting calculations. When we apply a transform to a surface, we need to derive normal vectors for the resulting surface from the original normal vectors. As we know, translation does not change the direction of normals. However, if we perform a non-uniform scaling on our object, the transformation of its normal vectors needs special consideration.

In order to transform normal vectors correctly, we do not simply multiply them by the same matrix used to transform our object, but multiply them by the transpose of the inverse of that matrix:

$$N' = (M^{-1})^T * N$$

Where  $N$  is the original normal vector and  $N'$  is the normal vector after transformation, and  $M$  is the transform matrix for our object.

Mathematically, we can consider  $M$  to be a  $3 \times 3$  transformation matrix because translation does not affect normal vectors. Suppose  $N$  is the normal vector and  $T$  is the tangent vector:

$$N \cdot T = 0$$

The dot product can be transformed into a product of vectors:

$$N^T * T = 0$$

Suppose  $M$  can be any  $3 \times 3$  invertible transformation:

$$M^{-1}M = I$$

And suppose  $N'$  and  $T'$  are the vectors after being transformed by  $M$ . For the tangent vector  $T$ , we can use the same transformation matrix we used on our object ( $M$ ) to obtain  $T'$ . But with  $N$ , the situation is different:

$$N' \cdot T' = (N')^T * T' = 0$$

Suppose the matrix  $M_1$  is the correct transform matrix for transforming the normal vector. Then we have:

$$(M_1N) \cdot (MT) = (M_1N)^T * (MT)$$

We also know that the transpose of a multiplication is the multiplication of the transpose, hence:

$$(M_1N)^T * (MT) = N^T M_1^T MT$$

We started by stating that the dot product between  $N$  and  $T$  was zero, so if

$$M_1^T M = I \quad \text{or} \quad M_1 = (M^{-1})^T$$

Then we have

$$N' \cdot T' = N \cdot T = 0$$

Which is exactly the conclusion we want; that is, to transform normal vectors correctly, we must perform the transpose of the inverse of the transform matrix used in transforming our object.

From the above proof, we know that it is possible to get the correct transformation matrix for normal vectors from an object transform matrix by taking the transpose-inverse of the transform matrix. In OpenGL or WebGL, we would usually implement this transpose-inverse operation in the shader to speed up computation, using the following GLSL code:

```
vNormal = mat3(transpose(inverse(modelMatrix))) * normal;
```

However, WGSL has not yet implemented a matrix *inverse* function, so we currently cannot perform this calculation easily in the WGSL shader. Instead, we can use GLSL shader, or we can calculate the transpose and inverse transform matrix for normal vectors using Rust code in our application in CPU and then pass the matrix to the WGSL shader to calculate lighting in GPU. Here, we will use the latter approach with the WGSL shader.

The *cgmath* library has the *invert* and *transpose* functions, which allow us to calculate the transformed normal matrix from the *model\_mat* we used to transform our object:

```
let normal_mat = (model_mat.invert().unwrap()).transpose();
```

This code computes the inverse of *model\_mat*, then performs the transpose on it, and finally places the result in *normal\_mat*. After this inverse-transpose transformation, *normal\_mat* can be sent to the shader program, where we use *normal\_mat \* normal\_data* to compute the lighting.

## 8.4.2 Shader with Lighting

With all the background information presented above, we are now ready to implement our shaders. In our shader program, we will consider three types of light sources: ambient light, diffuse light, and specular light.

Add a new sub-folder called *ch08* to the *examples/* folder and add a new *shader.wgsl* file to this newly created folder with the following code:

```
// vertex shader

[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
    normal_mat : mat4x4<f32>;
};
[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_position : vec4<f32>;
    [[location(1)]] v_normal : vec4<f32>;
};

[[stage(vertex)]]
```

## 160 | Practical GPU Graphics with wgpu and Rust

```

fn vs_main([[location(0)]] pos: vec4<f32>, [[location(1)]] normal: vec4<f32>) -> Output {
    var output: Output;
    let m_position:vec4<f32> = uniforms.model_mat * pos;
    output.v_position = m_position;
    output.v_normal = uniforms.normal_mat * normal;
    output.position = uniforms.view_project_mat * m_position;
    return output;
}

// fragment shader

[[block]] struct Uniforms {
    light_position : vec4<f32>;
    eye_position : vec4<f32>;
};

[[binding(1), group(0)]] var<uniform> frag_uniforms : Uniforms;

[[block]] struct Uniforms {
    color : vec4<f32>;
    specular_color : vec4<f32>;
    ambient_intensity: f32;
    diffuse_intensity :f32;
    specular_intensity: f32;
    specular_shininess: f32;
};
[[binding(2), group(0)]] var<uniform> light_uniforms : Uniforms;

[[stage(fragment)]]
fn fs_main([[location(0)]] v_position: vec4<f32>, [[location(1)]] v_normal: vec4<f32>) -> [[location(0)]] vec4<f32> {
    let N:vec3<f32> = normalize(v_normal.xyz);
    let L:vec3<f32> = normalize(frag_uniforms.light_position.xyz - v_position.xyz);
    let V:vec3<f32> = normalize(frag_uniforms.eye_position.xyz - v_position.xyz);
    let H:vec3<f32> = normalize(L + V);
    let diffuse:f32 = light_uniforms.diffuse_intensity * max(dot(N, L), 0.0);
    let specular: f32 = light_uniforms.specular_intensity *
        pow(max(dot(N, H),0.0), light_uniforms.specular_shininess);
    let ambient:f32 = light_uniforms.ambient_intensity;
    return light_uniforms.color.xyz*(ambient + diffuse) + light_uniforms.specular_color.xyz * specular;
}

```

Here, we first pass three uniform matrices to the vertex shader: *model\_mat*, *view\_project\_mat*, and *normal\_mat*. The input variables *position* and *normal* represent the original vertex data and normal vector data, while the output variables *v\_position* and *v\_normal* defined inside the *Output* struct represent the vertex data and normal data after transformation, which will be passed to the fragment shader to calculate lighting.

In the fragment shader, we have two uniform structs: the *frag\_uniforms* struct is used to pass the *light\_position* and *eye\_position* data, while the *light\_uniforms* struct is used to pass the parameters used to compute lighting. The lighting calculations inside the fragment shader are straightforward, starting with defining the various vectors *N*, *L*, *V*, and *H* which were explained previously. Note that the ambient light is included by simply adding a constant *ambient\_intensity* to the diffuse light.

The *light\_uniforms* struct includes several fields that allow you to specify contributions from different light sources. For example:

- Ambient light only: set *ambient\_intensity* = 1.0, *diffuse\_intensity* = 0.0, and *specular\_intensity* = 0.0.

- Diffuse light only: set *ambient\_intensity* to a small number such as 0.1, *diffuse\_intensity* = 1.0, and *specular\_intensity* = 0.0.
- Specular light only: set *ambient\_intensity* to a small number such as 0.1, *diffuse\_intensity* = 0.0, and *specular\_intensity* = 1.0. You can also specify *specular\_shininess*, where a smaller *specular\_shininess* parameter such as 5.0 represents a rough surface, while a larger *shininess* such as 300.0 represents a metallic shiny surface.
- Mixed light: set *ambient\_intensity* to a small number such as 0.1, and set *diffuse\_intensity* and *specular\_intensity* to values in the middle of the [0.0, 1.0] range.
- The *color* field in *light\_uniforms* represents the color of the ambient and diffuse lights, while the *specular\_color* field is the color of the specular light. The diffuse light and specular light can have different colors if these two fields are different.

Please note that all the fields inside the *light\_uniforms* struct are either *vec4<f32>* or *f32* type, so their values need to include a decimal point followed by zero or a digit. For example, *specular\_shininess* = 100 will generate an error when running the application; you must use *specular\_shininess* = 100.0 instead.

## 8.5 Common Code

As we did in the preceding chapter, to avoid code duplication, we will now implement a common file called *common.rs*, which can be reused to create many different 3D objects with lighting.

Add a new Rust file called *common.rs* to the *examples/ch08/* folder and enter the following content into it:

```
use std:: {iter, mem };
use cgmath::{ Matrix, Matrix4, SquareMatrix };
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ ControlFlow, EventLoop }
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    color: [f32; 4],
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
}

pub fn light(c:[f32; 3], sc:[f32;3], ai: f32, di: f32, si: f32, ss: f32) -> Light {
    Light {
```

```

        color:[c[0], c[1], c[2], 1.0],
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ai,
        diffuse_intensity: di,
        specular_intensity: si,
        specular_shininess: ss,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
}

#[allow(dead_code)]
pub fn vertex(p:[f32;3], n:[f32; 3]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
    }
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
    num_vertices: u32,
}

impl State {
    async fn new(window: &Window, vertex_data: &Vec<Vertex>, light_data: Light) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("shader.wgsl").into()),
        });

        // uniform data
        let camera_position = (3.0, 1.5, 3.0).into();
        let look_direction = (0.0, 0.0, 0.0).into();
        let up_direction = cgmath::Vector3::unit_y();
    }
}

```

```

let (view_mat, project_mat, _view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
        init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

// create vertex uniform buffer
// model_mat and view_projection_mat will be stored in
// vertex_uniform_buffer inside the update function
let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 192,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// create fragment uniform buffer. here we set eye_position = camera_position and
// light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 48,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout =
    init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
            },
        ],
    });

```

```

        },
        count: None,
    },
    wgpu::BindGroupLayoutEntry {
        binding: 2,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    }
],
label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
}

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    })
});
}

```

```

        },
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
}

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = vertex_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    view_mat,
    project_mat,
    num_vertices,
}
}

fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0],
        [dt.sin(), dt.cos(), 0.0], [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;
}

```

```

let normal_mat = (model_mat.invert().unwrap()).transpose();

let model_ref:&[f32; 16] = model_mat.as_ref();
let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
let normal_ref:&[f32; 16] = normal_mat.as_ref();

self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
    bytemuck::cast_slice(view_projection_ref));
self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            //depth_stencil_attachment: None,
            depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
                view: &depth_view,
                depth_ops: Some(wgpu::Operations {

```

```

        load: wgpu::LoadOp::Clear(1.0),
        store: false,
    })),
    stencil_ops: None,
),
}),
});

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

pub fn run(vertex_data: &Vec<Vertex>, light_data: Light, title: &str) {
env_logger::init();
let event_loop = EventLoop::new();
let window = winit::window::WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&format!("ch08_{}", title));

let mut state = pollster::block_on(State::new(&window, &vertex_data, light_data));
let render_start_time = std::time::Instant::now();

event_loop.run(move |event, _, control_flow| {
match event {
Event::WindowEvent {
ref event,
window_id,
} if window_id == window.id() => {
if !state.input(event) {
match event {
WindowEvent::CloseRequested
| WindowEvent::KeyboardInput {
input:
KeyboardInput {
state: ElementState::Pressed,
virtual_keycode: Some(VirtualKeyCode::Escape),
..
},
..
} => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
state.resize(**new_inner_size);
}
_ => {}
}
}
}
}

Event::RedrawRequested(_) => {
let now = std::time::Instant::now();
let dt = now - render_start_time;
}
}
}
}
}
```

```

        state.update(dt);

        match state.render() {
            Ok(_) => {}
            Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
            Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
            Err(e) => eprintln!(":{}?", e),
        }
    }

    Event::MainEventsCleared => {
        window.request_redraw();
    }
    _ => {}
}
);
}
}

```

Here, we first define a *Light* struct which includes the parameters used to calculate lighting, and then implement a function called *light* that accepts the *color* and *specular\_color* fields as  $[f32; 3]$  types and converts them into  $[f32; 4]$  types, which can be used for lighting computations in the shader.

Next, we define the *Vertex* struct, which consists of two fields, *position* and *normal*, representing the vertex position and normal vector data respectively. We then use the *vertex\_attr\_array* macro to write an implementation block for the *Vertex* struct that creates a static method called *desc<'a>()*, which returns *VertexBufferLayout*. Note that the *ATTRIBUTES* constant specifies a list that contains two elements: the first element represents the vertex position with *shader\_location* = 0; and the second element represents the normal-vector data with *shader\_location* = 1. Both elements have the same format of *Float32x4*

We then pack all the fields required for drawing our graphics shape into another struct called *State*, and create the following methods on it:

```

pub struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
    num_vertices: u32,
}

```

Within the *State* implementation, the async *new* function now accepts *vertex\_data* and *light\_data* as its input arguments that will be provided by the user.

In the uniform data section, we create the view and projection matrices, *view\_mat* and *project\_mat*, by calling the *transforms::create\_view\_projection* method. We then create a *vertex\_uniform\_buffer* using the *device.create\_buffer* function, which is different from the *device.create\_buffer\_init* function that we previously used to create the *uniform\_buffer*. This is because we want to set the buffer *size* and *offset* attributes explicitly when creating our vertex uniform buffer:

```

let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 192,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

```

In our current example, the `vertex_uniform_buffer` will store three 4-by-4 matrices, including `model_mat`, `view_projection_mat`, and `normal_mat`. Each matrix needs 64 bytes of memory to store its data, so the total buffer size is equal to  $3 \times 64 = 192$ . Since these matrices will change through either a rotation animation (model and normal matrices) or a window resize (projection matrix), we will write these data to the `vertex_uniform_buffer` inside the `update` function.

The `vertex_uniform_buffer` also contains a `mapped_at_creation` attribute that is set to `false`, which means that our buffer is unmapped. Note that the GPU can only access an unmapped buffer. However, in situations where you may want to use the CPU to manipulate data stored in a buffer, you need to set the `mapped_at_creation` attribute to `true`.

Next, we create a `fragment_uniform_buffer` that will store the eye position (camera position or view position) and the light position. For simplicity's sake, here, we set `light_position = eye_position`, but you are free to set them to different values. This buffer has a size of 32 because it will be used to store two positions of  $[f32; 4]$  type. The `light_position` and `eye_position` will not change with rotation animations or window resizing, so we can store them immediately after creating the buffer:

```
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));
```

Note that we set the `offset` to 16 when writing the `eye_position` to the buffer.

Next, we create a `light_uniform_buffer` with the following code snippet:

```
// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 48,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));
```

Here, we set the buffer `size` to 48 because the `Light` struct contains two  $[f32; 4]$  fields (32 bytes) and four `f32` fields (16 bytes). We also write the lighting parameters directly to the buffer.

I should point out here that you can use a uniform buffer to pass any type of parameters to the shader in the same way we did here with lighting parameters.

We then create a `uniform_bind_group_layout` and a `uniform_bind_group` whose `entries` attribute contains three elements with a `binding` of 0, 1, and 2, which correspond to the `vertex_uniform_buffer`, `fragment_uniform_buffer`, and `light_uniform_buffer`, respectively.

In fact, we can also create three separate bind groups (and three bind group layouts), one for each specific buffer. In such a case, each bind group would only have a single binding of 0. Remember that in our shader code, we use the bind group and binding location to access data stored in different buffers.

Inside the *update* function, we set a rotation animation with a time duration, which is passed to the model matrix. We then create the view-projection matrix and the normal transform matrix by inverting and transposing the model matrix. Finally, we write the model, view-projection, and normal matrices to the *vertex\_uniform\_buffer* with an *offset* of 0, 64, and 128, respectively.

To avoid code duplication, we also convert the content usually found in the *main* function into a new *run* function.

The rest of the code is the same as that used in the examples presented in the preceding chapter.

## 8.6 Cube with Lighting

In this section, we will add lighting to our simple 3D cube. We previously created the normal data for our cube in the *cube\_data* method in the *vertex\_data.rs* file located in the *examples/common/* folder. Here, we will use this data to compute lighting effects for the cube.

### 8.6.1 Rust Code

Add a new Rust file called *cube.rs* to the *examples/ch08/* folder and type the following code into it:

```
mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn vertex(p:[i8; 3], n: [i8; 3]) -> common::Vertex {
    common::Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        normal: [n[0] as f32, n[1] as f32, n[2] as f32, 1.0],
    }
}

fn create_vertices() -> Vec<common::Vertex> {
    let(pos, _col, _uv, normal) = vertex_data::cube_data();
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i]));
    }
    data.to_vec()
}

fn main(){
    let vertex_data = create_vertices();
    let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
    common::run(&vertex_data, light_data, "cube");
}
```

This code is very simple. It first implements a new *vertex* method that converts position and normal data of *[i8; 3]* type into corresponding data of *[f32; 4]* type. We cannot use the *common::vertex* method because that method requires *[f32; 3]*-type data as inputs, while our data from the *vertex\_data::cube\_data* method is *[i8; 3]* type.

Next, we use the *create\_vertices* method to generate the vertex position and normal-vector data with the correct data type of *Vec<common::Vertex>*, which can be used to create the vertex buffer.

Inside the `main` function, we call the `create_vertices` and `common::light` methods to generate the vertex and light data that we use as inputs for the `common::run` function:

```
let vertex_data = create_vertices();
let light_data = common::light([1.0, 0.0, 0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
```

## 8.6.2 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch08_cube"
path = "examples/ch08/cube.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch08_cube
```

This produces the results shown in Fig.8-4.

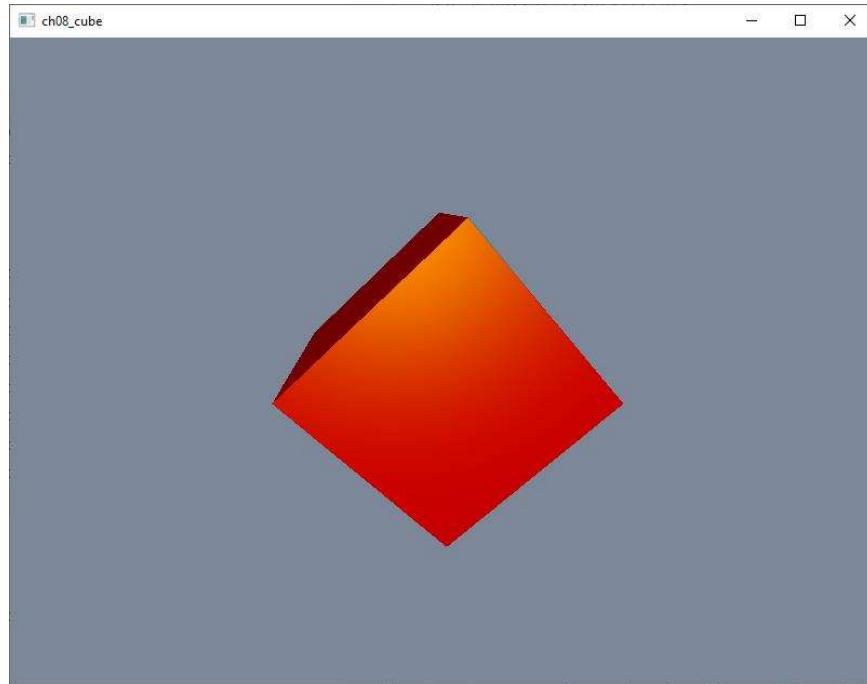


Fig.8-4. Cube with lighting.

You can obtain different lighting effects on your cube by changing the lighting parameters when you generate light input data using the `common::light` method.

## 8.7 Sphere with Lighting

The vertex and normal data for a sphere can be easily generated using the spherical coordinate system, as shown in Fig.7-2. We will also use the standard UV-sphere method made out of  $u$  segments and  $v$  rings, as shown in Fig.7-3. We can see that the  $u$  and  $v$  lines form grids, or cells, on the surface of the

sphere. In order to create vertex and normal data for this surface, it is enough to consider just one unit cell, as shown in Fig.8-5.

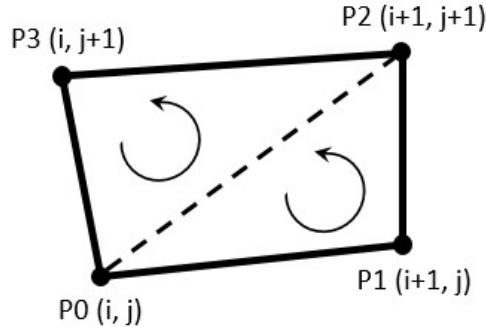


Fig.8-5. Unit cell of a UV-spherical surface.

### 8.7.1 Vertex and Normal Data

To create vertex data for the unit cell shown in Fig.8-5, we need two triangles,  $\Delta(P0, P1, P2)$  and  $\Delta(P2, P3, P0)$ , whose points are arranged in counterclockwise order.

The normal for a sphere is very simple – it is just the normalized vertex data at point  $(x, y, z)$ :

```
normal = normalize(vertex_data(x,y,z));
```

or in other words, the vertex data divided by the radius.

Add a new method called *sphere\_data* to the *vertex\_data.rs* file in the *examples/common/* folder with the following code:

```
use cgmath::*;

pub fn sphere_data(r: f32, u: usize, v: usize) -> (Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 3]>) {
    let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4 * (u - 1) * (v - 1)) as usize);
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4 * (u - 1) * (v - 1)) as usize);
    let uvs: Vec<[f32; 3]> = Vec::with_capacity((4 * (u - 1) * (v - 1)) as usize);

    for i in 0..u - 1 {
        for j in 0..v - 1 {
            let theta = i as f32 * 180.0 / (u as f32 - 1.0);
            let phi = j as f32 * 360.0 / (v as f32 - 1.0);
            let theta1 = (i as f32 + 1.0) * 180.0 / (u as f32 - 1.0);
            let phi1 = (j as f32 + 1.0) * 360.0 / (v as f32 - 1.0);
            let p0 = math_func::sphere_position(r, Deg(theta), Deg(phi));
            let p1 = math_func::sphere_position(r, Deg(theta1), Deg(phi));
            let p2 = math_func::sphere_position(r, Deg(theta1), Deg(phi1));
            let p3 = math_func::sphere_position(r, Deg(theta), Deg(phi1));

            // positions
            positions.push(p0);
            positions.push(p1);
            positions.push(p3);
            positions.push(p1);
            positions.push(p2);
            positions.push(p3);
        }
    }
}
```

```

    // normals
    normals.push([p0[0]/r, p0[1]/r, p0[2]/r]);
    normals.push([p1[0]/r, p1[1]/r, p1[2]/r]);
    normals.push([p3[0]/r, p3[1]/r, p3[2]/r]);
    normals.push([p1[0]/r, p1[1]/r, p1[2]/r]);
    normals.push([p2[0]/r, p2[1]/r, p2[2]/r]);
    normals.push([p3[0]/r, p3[1]/r, p3[2]/r]);
}
}

(positions, normals, uvs)
}

```

This method first generates position coordinates for four vertices within a unit cell by calling the `math_func::sphere_position` function. We then create the position vertices and normal vector data for a unit cell that contains two triangles. Here, we also create a placeholder for UV coordinate data, which will be defined in later chapters when we discuss image textures in `wgpu` applications.

## 8.7.2 Rust Code

The Rust file used in this example is similar to the one used in the preceding example, except that the vertex data and normal vector data of the cube are replaced by those of a sphere.

Add a new Rust file called `sphere.rs` to the `examples/ch08/` folder with the following code:

```

mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices(r: f32, u:usize, v: usize) -> Vec<common::Vertex> {
    let(pos, normal, _uvs) = vertex_data::sphere_data(r, u, v);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i]));
    }
    data.to_vec()
}

fn main(){
    let vertex_data = create_vertices(1.5, 15, 20);
    let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
    common::run(&vertex_data, light_data, "sphere");
}

```

Here, we first define a `create_vertices` method that converts the position and normal vector data from `vertex_data::sphere_data()` into the correct format to be used to create the vertex buffer.

Inside the `main` function, we call the `create_vertices` and `common::light` methods to generate the vertex and light data used as inputs for the `common::run` function:

```

let vertex_data = create_vertices(1.5, 15, 20);
let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);

```

Here, we use the parameters  $r = 1.5$ ,  $u = 15$ , and  $v = 20$  to generate the vertex data for our sphere.

### 8.7.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch08_sphere"
path = "examples/ch08/sphere.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch08_sphere
```

This produces the results shown in Fig.8-6.

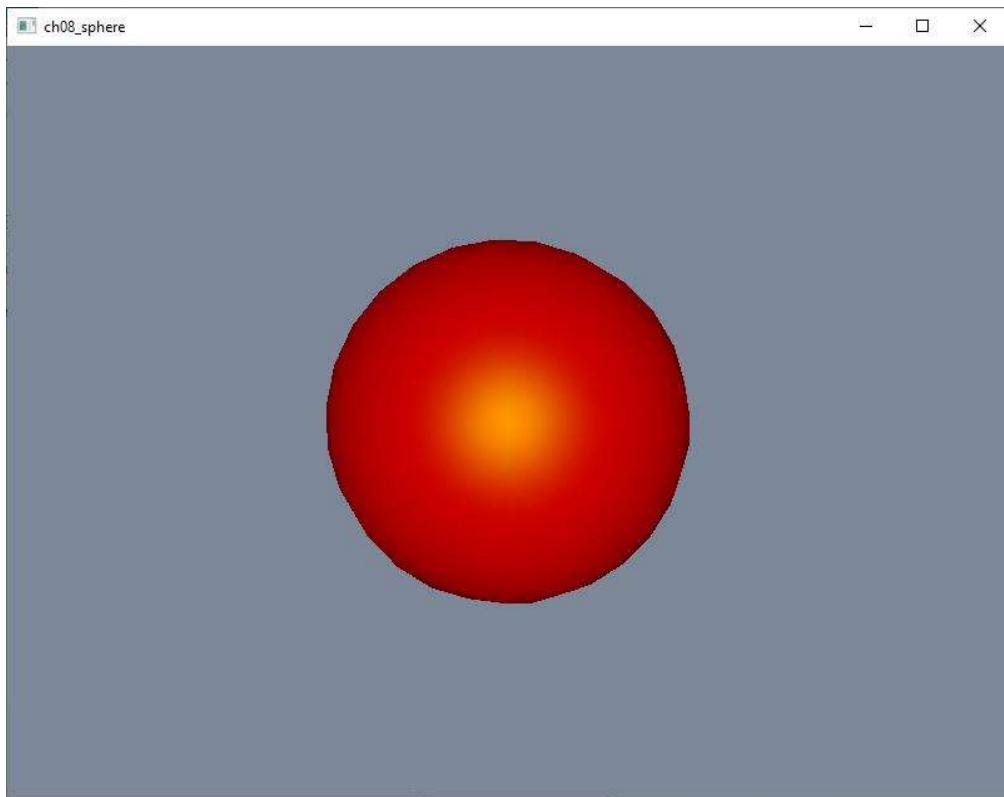


Fig.8-6. Sphere with lighting.

### 8.8 Cylinder with Lighting

The vertex and normal data for a cylinder can be easily generated using the cylindrical coordinate system, as shown in Fig.7-5. Like we did before, we need to make slices on the surface of the cylinder. As shown in Fig.7-6, the cylinder surface can be divided into  $n$  slices, and a unit cell is formed by the  $i$ -th and  $i+1$ -th slice lines. In order to create vertex and normal data, we need to perform surface triangulation for this unit cell, as shown in Fig.8-7.

You can see that each unit cell contains eight vertices and four surfaces. There are eight triangles that need to be drawn for each unit cell.

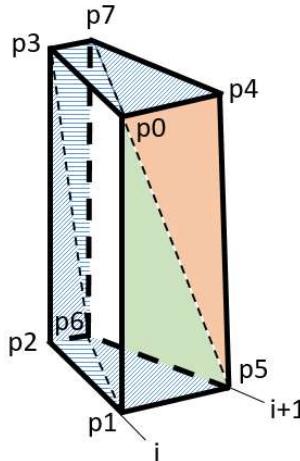


Fig.8-7. Surface triangulation for a cylindrical unit cell.

### 8.8.1 Vertex and Normal Data

A cylindrical unit cell has four surfaces: a top face, bottom face, inner face, and outer face. The normal vectors for the outer face all point outward while those for the inner face all point inward. For vertices on the top face, the surface normals are all equal to  $(0, 1, 0)$ , while for vertices on the bottom face, the surface normals are all equal to  $(0, -1, 0)$ .

Add a new method called *cylinder\_data* to the *vertex\_data* file in the *examples/common/* folder with the following code:

```
pub fn cylinder_data(rin: f32, rout: f32, height: f32, n: usize) ->
    (Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 3]>) {
    let h = height / 2.0;
    let mut positions: Vec<[f32; 3]> = Vec::with_capacity(24 * (n - 1));
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity(24 * (n - 1));
    let uvs: Vec<[f32; 3]> = Vec::with_capacity(24 * (n - 1));

    for i in 0..n - 1 {
        let theta = i as f32 * 360.0 / (n as f32 - 1.0);
        let theta1 = (i as f32 + 1.0) * 360.0 / (n as f32 - 1.0);
        let p0 = math_func::cylinder_position(rout, h, Deg(theta));
        let p1 = math_func::cylinder_position(rout, -h, Deg(theta));
        let p2 = math_func::cylinder_position(rin, -h, Deg(theta));
        let p3 = math_func::cylinder_position(rin, h, Deg(theta));
        let p4 = math_func::cylinder_position(rout, h, Deg(theta1));
        let p5 = math_func::cylinder_position(rout, -h, Deg(theta1));
        let p6 = math_func::cylinder_position(rin, -h, Deg(theta1));
        let p7 = math_func::cylinder_position(rin, h, Deg(theta1));

        // positions
        positions.push(p0);
        positions.push(p4);
        positions.push(p7);
        positions.push(p7);
        positions.push(p3);
        positions.push(p0);
    }
}
```

```

// bottom face
positions.push(p1);
positions.push(p2);
positions.push(p6);
positions.push(p6);
positions.push(p5);
positions.push(p1);

// outer face
positions.push(p0);
positions.push(p1);
positions.push(p5);
positions.push(p5);
positions.push(p4);
positions.push(p0);

// inner face
positions.push(p2);
positions.push(p3);
positions.push(p7);
positions.push(p7);
positions.push(p6);
positions.push(p2);

// normals

// top face
normals.push([0.0, 1.0, 0.0]);

// bottom face
normals.push([0.0, -1.0, 0.0]);

// outer face
normals.push([p0[0]/rout, 0.0, p0[2]/rout]);
normals.push([p1[0]/rout, 0.0, p1[2]/rout]);
normals.push([p5[0]/rout, 0.0, p5[2]/rout]);
normals.push([p5[0]/rout, 0.0, p5[2]/rout]);
normals.push([p4[0]/rout, 0.0, p4[2]/rout]);
normals.push([p0[0]/rout, 0.0, p0[2]/rout]);

// inner face
normals.push([p3[0]/rin, 0.0, p3[2]/rin]);
normals.push([p7[0]/rin, 0.0, p7[2]/rin]);
normals.push([p6[0]/rin, 0.0, p6[2]/rin]);
normals.push([p6[0]/rin, 0.0, p6[2]/rin]);
normals.push([p2[0]/rin, 0.0, p2[2]/rin]);
normals.push([p3[0]/rin, 0.0, p3[2]/rin]);
}
(positions, normals, uvs)
}

```

Note that the normal vectors for the inner and outer faces are simply the vertex data divided by their respective radius. Here, we also create a placeholder for the UV coordinate data, which will be defined later when we discuss image textures.

## 8.8.2 Rust Code

The Rust file used in this example is similar to the one used in the preceding example, except that the vertex data and normal data are replaced by those of a cylinder.

Add a new Rust file called *cylinder.rs* to the *examples/ch08/* folder and type the following code into it:

```
mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices(rin: f32, rout: f32, h: f32, n: usize) -> Vec<common::Vertex> {
    let(pos, normal, _uvs) = vertex_data::cylinder_data(rin, rout, h, n);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i]));
    }
    data.to_vec()
}

fn main(){
    let vertex_data = create_vertices(0.5, 1.5, 3.0, 30);
    let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
    common::run(&vertex_data, light_data, "cylinder");
}
```

Here, we first define a *create\_vertices* method that converts the position and normal vector data from *vertex\_data::cylinder\_data()* into the correct format to be used to create the vertex buffer.

Inside the *main* function, we call the *create\_vertices* and *common::light* methods to generate the vertex and light data as inputs for the *common::run* function:

```
let vtx_data = create_vertices(0.5, 1.5, 1.5, 30);
let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
```

Here, we use the parameters *rin* = 0.5, *rout* = 1.5, *h* = 1.5 and *n* = 30 to generate the vertex data for our cylinder.

## 8.8.3 Run Application

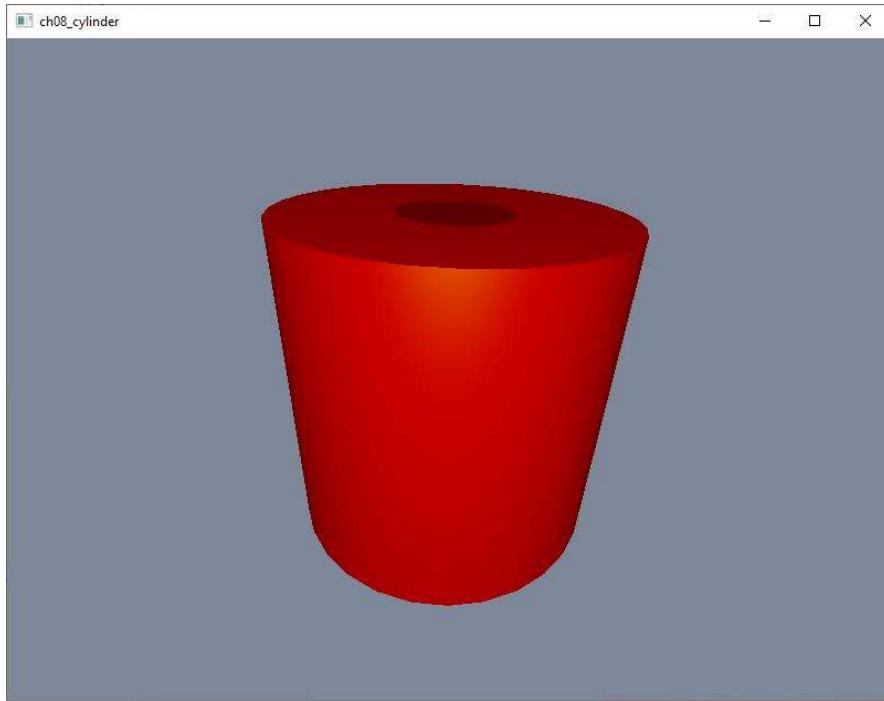
Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch08_cylinder"
path = "examples/ch08/cylinder.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch08_cylinder
```

This produces the results shown in Fig.8-8.



*Fig.8-8. Cylinder with lighting.*

You can see from the figure that there is no lighting on the inner surface of the cylinder. This is because the normal vectors for the inner surface are in the opposite direction of the normal vectors for the outer surface. We need a two-sided lighting model to resolve this issue, which will be the topic of the next chapter when we discuss 3D surfaces with colormaps.

## 8.9 Cone with Lighting

We can also use the cylindrical coordinate system to create a cone shape with lighting. Here we want to generate a general cone shape whose top radius, bottom radius, and height can all be specified. Fig.7-10 illustrates how slices are made on the surface of the cone and how a unit cell is defined. The unit cell contains three faces: a top face, bottom face, and outer face. The top face and bottom face are each a single triangle, with normal vectors of  $[0, 1, 0]$  for the top face and  $[0, -1, 0]$  for the bottom face.

Note that the outer face is a general quad with four vertices and two triangles, as shown in Fig.8-9.

As discussed previously, the normal vector for this quad can be easily calculated using the cross product like in the following pseudo code:

```
normal = normalize[cross-product(P5 - P0, P4 - p1)];
```

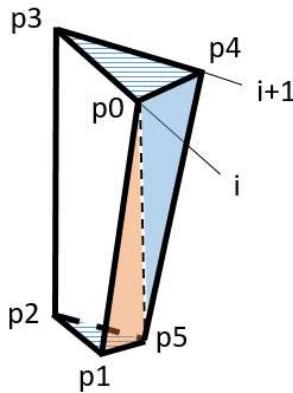


Fig. 8-9. Cone unit cell with four triangles.

### 8.9.1 Vertex and Normal Data

Add a new method called `cone_data` to the `vertex_data.rs` file and enter the following code into it:

```
pub fn cone_data(rtop: f32, rbottom: f32, height: f32, n: usize) ->
    (Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 3]>) {

    let h = height / 2.0;
    let mut positions: Vec<[f32; 3]> = Vec::with_capacity(12 * (n-1));
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity(12 * (n-1));
    let uvs: Vec<[f32; 3]> = Vec::with_capacity(12 * (n-1));

    for i in 0..n - 1 {
        let theta = i as f32 * 360.0 / (n as f32 - 1.0);
        let theta1 = (i as f32 + 1.0) * 360.0 / (n as f32 - 1.0);
        let p0 = math_func::cylinder_position(rtop, h, Deg(theta));
        let p1 = math_func::cylinder_position(rbottom, -h, Deg(theta));
        let p2 = math_func::cylinder_position(0.0, -h, Deg(theta));
        let p3 = math_func::cylinder_position(0.0, h, Deg(theta));
        let p4 = math_func::cylinder_position(rtop, h, Deg(theta1));
        let p5 = math_func::cylinder_position(rbottom, -h, Deg(theta1));

        // positions
        // top face
        positions.push(p0);
        positions.push(p4);
        positions.push(p3);

        // bottom face
        positions.push(p1);
        positions.push(p2);
        positions.push(p5);

        // outer face
        positions.push(p0);
        positions.push(p1);
        positions.push(p5);
        positions.push(p5);
        positions.push(p4);
        positions.push(p0);
    }
}
```

```

// normals
let ca = Vector3::new(p5[0]-p0[0], p5[1]-p0[1], p5[2]-p0[2]);
let db = Vector3::new(p4[0]-p1[0], p4[1]-p1[1], p4[2]-p1[2]);
let cp = (ca.cross(db)).normalize();

// top face
normals.push([0.0, 1.0, 0.0]);
normals.push([0.0, 1.0, 0.0]);
normals.push([0.0, 1.0, 0.0]);

// bottom face
normals.push([0.0, -1.0, 0.0]);
normals.push([0.0, -1.0, 0.0]);
normals.push([0.0, -1.0, 0.0]);

// outer face
normals.push([cp[0], cp[1], cp[2]]);
}
(positions, normals, uvs)
}

```

The vertex data for each unit cell is generated using four triangles and the normal vectors for the outer face are calculated using the cross product. Note that the four vertices on the outer face have the same normal vector:  $[cp[0], cp[1], cp[2]]$ .

## 8.9.2 Rust Code

The Rust file used in this example is similar to the one used in the preceding example, except that the vertex and normal data are replaced by those of a cone.

Add a new Rust file called *cone.ts* to the *examples/ch08/* folder and type the following code into it:

```

mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices(rtop: f32, rbottom: f32, height: f32, n: usize) -> Vec<common::Vertex> {
    let(pos, normal, _uvs) = vertex_data::cone_data(rtop, rbottom, height, n);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i]));
    }
    data.to_vec()
}

fn main(){
    let vertex_data = create_vertices(0.5, 1.5, 2.0, 30);
    let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
    common::run(&vertex_data, light_data, "cone");
}

```

Here, we first define a *create\_vertices* method that converts the position and normal vector data from *vertex\_data::cone\_data()* into the correct format to be used to create the vertex buffer.

Inside the `main` function, we call the `create_vertices` and `common::light` methods to generate the vertex and light data used as inputs for the `common::run` function:

```
let vtx_data = create_vertices(0.5, 1.5, 2.0, 30);
let light_data = common::light([1.0, 0.0, 0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
```

Here, we use the parameters  $r_{top} = 0.5$ ,  $r_{bottom} = 1.5$ ,  $h = 2.0$ , and  $n = 30$  to generate the vertex data for our cone.

### 8.9.3 Run Application

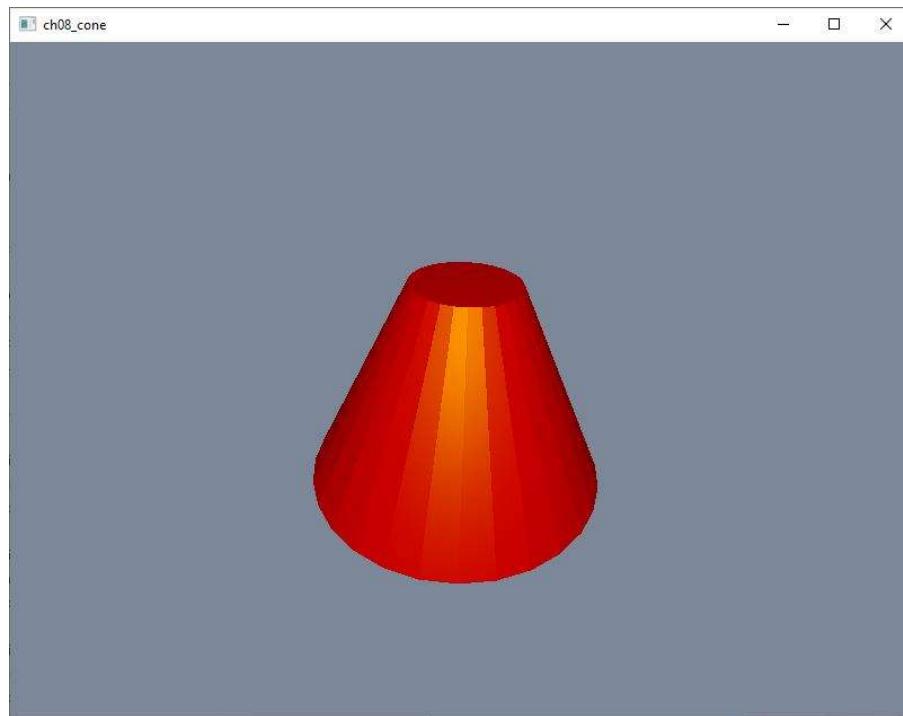
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch08_cone"
path = "examples/ch08/cone.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch08_cone
```

This produces the results shown in Fig.8-10.



*Fig.8-10. Cone with lighting.*

## 8.10 Torus with Lighting

In order to create a torus shape, we need to divide its surface using tube rings and torus rings, as shown in Fig.7-12.

The unit cell for the torus is a quad with four vertices as shown in Fig.8-11. It consists of two triangles. The normal vector for this unit cell can be calculated using the following formula:

```
normal = normalize[cross-product(P2 - P0, P3 - p1)];
```

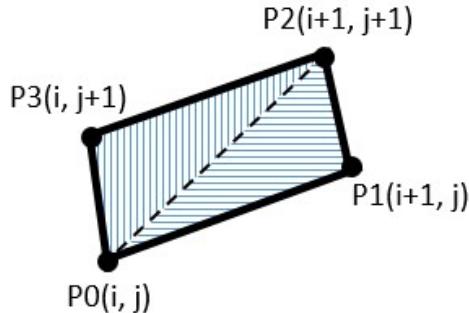


Fig.8-11. Unit cell of a torus.

### 8.10.1 Vertex and Normal Data

Add a new method called `torus_data` to the `vertex_data.rs` file and enter the following content into it:

```
pub fn torus_data(r_torus:f32, r_tube:f32, n_torus:usize, n_tube:usize) ->
    (Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 3]>) {
    let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4* (n_torus - 1)*(n_tube -1)) as usize);
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4* (n_torus - 1)*(n_tube -1)) as usize);
    let uvs: Vec<[f32; 3]> = Vec::with_capacity((4* (n_torus - 1)*(n_tube -1)) as usize);

    for i in 0..n_torus - 1 {
        for j in 0..n_tube - 1 {
            let u = i as f32 * 360.0/(n_torus as f32 - 1.0);
            let v = j as f32 * 360.0/(n_tube as f32 - 1.0);
            let u1 = (i as f32 + 1.0) * 360.0/(n_torus as f32 - 1.0);
            let v1 = (j as f32 + 1.0) * 360.0/(n_tube as f32 - 1.0);
            let p0 = math_func::torus_position(r_torus, r_tube, Deg(u), Deg(v));
            let p1 = math_func::torus_position(r_torus, r_tube, Deg(u1), Deg(v));
            let p2 = math_func::torus_position(r_torus, r_tube, Deg(u1), Deg(v1));
            let p3 = math_func::torus_position(r_torus, r_tube, Deg(u), Deg(v1));

            // positions
            positions.push(p0);
            positions.push(p1);
            positions.push(p2);
            positions.push(p2);
            positions.push(p3);
            positions.push(p0);

            // normals
            let ca = Vector3::new(p2[0]-p0[0], p2[1]-p0[1], p2[2]-p0[2]);
            let db = Vector3::new(p3[0]-p1[0], p3[1]-p1[1], p3[2]-p1[2]);
            let cp = (ca.cross(db)).normalize();

            normals.push([cp[0], cp[1], cp[2]]);
        }
    }
}
```

```

        normals.push([cp[0], cp[1], cp[2]]));
        normals.push([cp[0], cp[1], cp[2]]));
        normals.push([cp[0], cp[1], cp[2]]));
        normals.push([cp[0], cp[1], cp[2]]));
        normals.push([cp[0], cp[1], cp[2]]));
    }
}
(positions, normals, uvs)
}

```

The vertex data for each unit cell is generated using two triangles and the normal vectors are calculated using the cross product. Note that the vertices within a unit cell have the same normal vector:  $[cp[0], cp[1], cp[2]]$ .

## 8.10.2 Rust Code

The Rust file used in this example is similar to the one used in the preceding example, except that the vertex and normal data are replaced by those of a torus.

Add a new Rust file called *torus.rs* to the *examples/ch08/* folder with the following code:

```

mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices(r_torus: f32, r_tube: f32, n_torus: usize, n_tube:usize) -> Vec<common::Vertex> {
    let(pos, normal, _uvs) = vertex_data::torus_data(r_torus, r_tube, n_torus, n_tube);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i]));
    }
    data.to_vec()
}

fn main(){
    let vertex_data = create_vertices(1.5, 0.4, 100, 50);
    let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);
    common::run(&vertex_data, light_data, "torus");
}

```

Here, we first define a *create\_vertices* method that converts the position and normal vector data from *vertex\_data::torus\_data()* into the correct format to be used to create the vertex buffer.

Inside the *main* function, we call the *create\_vertices* and *common::light* methods to generate the vertex and light data used as inputs for the *common::run* function:

```

let vtx_data = create_vertices(1.5, 0.4, 100, 50);
let light_data = common::light([1.0,0.0,0.0], [1.0, 1.0, 0.0], 0.1, 0.6, 0.3, 30.0);

```

Here, we use the parameters  $r\_torus = 1.5$ ,  $r\_tube = 0.4$ ,  $n\_torus = 100$ , and  $n\_tube = 50$  to generate the vertex data for our torus.

## 8.10.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch08_torus"

```

```
path = "examples/ch08/torus.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch08_torus
```

This produces the results shown in Fig.8-12.

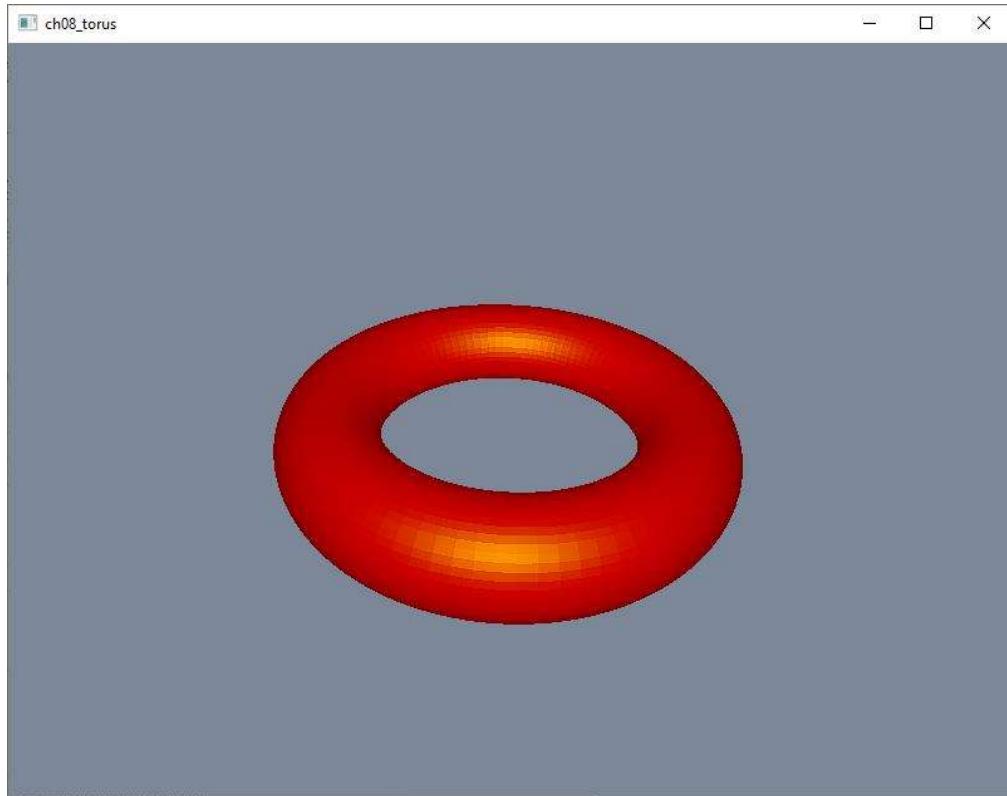


Fig.8-12. Torus with lighting.

# 9 Colormaps and 3D Surfaces

For most of the examples presented in this book so far, we have applied fixed colors to our 3D objects by assigning our desired color values to the *frag\_color* variable in the fragment shader. The way *wgpu* builds objects is by using sets of vertices, each of which has a position and a color. By default, the colors and other attributes of all other pixels are computed using interpolation, automatically creating smooth gradients. In Chapter 6, we demonstrated this color interpolation when we created a cube with distinct vertex colors.

In this chapter, I will explain how to use color models and colormaps to render simple and parametric 3D surfaces by specifying various mathematical functions. Surfaces play an important role in various applications, including computer graphics, virtual reality, computer games, and 3D data visualization.

## 9.1 Color Models

Colors in *wgpu* are created by a combination of red (R), green (G), and blue (B) light. We can generate different colors by varying the intensity of each type of light. A color can be specified by three components, one for each of R, G, and B, in the form of a number in the range [0, 1] representing the intensity of the color, with zero being the minimum intensity and one being the maximum. This approach toward specifying color is called the RGB color model. For example, the number (1.0, 1.0, 0.0) represents a yellow color obtained by setting red and green to full intensity, while blue is set to minimum intensity.

RGB colors are often represented using 8 bits per color component, for a total of 24 bits per color. This RGB color system is often called 24-bit color model. Since an 8-bit number can represent  $2^8$ , or 256, a color can then be specified as a triple of integers  $(r, g, b)$  with each component being an integer in the range [0, 255].

The human eye can really only distinguish between approximately 256 shades of R, G, and B, so this is an ideal model for representing colors. However, in computer graphics, you will often see applications use 16 bit integers or even 32-bit floating-point values for each component in order to avoid rounding errors during color calculations.

Additionally, we sometimes add a fourth component to the color model, *alpha* (A), which represents transparency. In the four-component RGBA color model, zero *alpha* means a color is completely transparent, or invisible, and maximum alpha of one means it is completely opaque, with intermediate levels of translucency or transparency in between.

In this book, we will use a *float32* value to represent each color component.

## 9.2 Colormaps

One approach we can use for creating certain visual effects is to associate our graphics objects with colormaps. A colormap is simply a table or list of colors organized in a specific way. We can easily create a custom colormap using an  $m \times 3$  colormap matrix, with each row representing an RGB value. These RGB color values can then be mapped onto an object using the row indices of the matrix.

For example, we can linearly scale the  $y$  data of a 2D chart to the row indices, so that each  $y$  data value is associated with the RGB row (i.e., color) in the colormap matrix identified by the index. All we need to do is use  $Y_{\min}$  and  $Y_{\max}$  to perform a linear transformation on the  $y$  data to convert it into color indices. If our matrix has  $m$  rows, then for each individual value of  $y$ , we can calculate a corresponding color index using the following formula:

$$\text{Color Index} = \begin{cases} 1 & y < Y_{\min} \\ (\text{int}) \left( \frac{(y - Y_{\min})m}{Y_{\max} - Y_{\min}} \right) & Y_{\min} \leq y < Y_{\max} \\ m & y \geq Y_{\max} \end{cases}$$

This means we can use the entire range of colors in the colormap over the plotted data. We can represent the height of 3D graphics objects and 3D surface charts in the same way if we replace the  $y$  data with the height ( $z$  or  $y$  depending on your coordinate system) data.

### 9.2.1 Colormap Data

You can use simple mathematical formulas to easily generate your own custom colormaps. Here, I will simply provide several commonly used colormaps in the form of various  $m \times 3$  colormap arrays. Add a new Rust file called `colormap.rs` to the `examples/common/` folder and type the following code into it:

```
pub fn colormap_data(colormap_name: &str) -> [[f32; 3]; 11] {
    let colors = match colormap_name {
        "hsv" => [[1.0, 0.0, 0.0], [1.0, 0.5, 0.0], [0.97, 1.0, 0.01], [0.0, 0.99, 0.04], [0.0, 0.98, 0.52],
                    [0.0, 0.98, 1.0], [0.01, 0.49, 1.0], [0.03, 0.0, 0.99], [1.0, 0.0, 0.96], [1.0, 0.0, 0.49], [1.0, 0.0, 0.02]],
        "hot" => [[0.0, 0.0, 0.0], [0.3, 0.0, 0.0], [0.6, 0.0, 0.0], [0.9, 0.0, 0.0], [0.93, 0.27, 0.0],
                    [0.97, 0.55, 0.0], [1.0, 0.82, 0.0], [1.0, 0.87, 0.25], [1.0, 0.91, 0.5], [1.0, 0.96, 0.75], [1.0, 1.0, 1.0]],
        "cool" => [[0.49, 0.0, 0.7], [0.45, 0.0, 0.85], [0.42, 0.15, 0.89], [0.38, 0.29, 0.93], [0.27, 0.57, 0.91],
                    [0.0, 0.8, 0.77], [0.0, 0.97, 0.57], [0.0, 0.98, 0.46], [0.0, 1.0, 0.35], [0.16, 1.0, 0.03], [0.58, 1.0, 0.0]],
        "spring" => [[1.0, 0.0, 1.0], [1.0, 0.1, 0.9], [1.0, 0.2, 0.8], [1.0, 0.3, 0.7], [1.0, 0.4, 0.6],
                      [1.0, 0.5, 0.5], [1.0, 0.6, 0.4], [1.0, 0.7, 0.3], [1.0, 0.8, 0.2], [1.0, 0.9, 0.1], [1.0, 1.0, 0.0]],
        "summer" => [[0.0, 0.5, 0.4], [0.1, 0.55, 0.4], [0.2, 0.6, 0.4], [0.3, 0.65, 0.4], [0.4, 0.7, 0.4],
                      [0.5, 0.75, 0.4], [0.6, 0.8, 0.4], [0.7, 0.85, 0.4], [0.8, 0.9, 0.4], [0.9, 0.95, 0.4], [1.0, 1.0, 0.4]],
        "autumn" => [[1.0, 0.0, 0.0], [1.0, 0.1, 0.0], [1.0, 0.2, 0.0], [1.0, 0.3, 0.0], [1.0, 0.4, 0.0], [1.0, 0.5, 0.0],
                      [1.0, 0.6, 0.0], [1.0, 0.7, 0.0], [1.0, 0.8, 0.0], [1.0, 0.9, 0.0], [1.0, 1.0, 0.0]],
        "winter" => [[0.0, 0.0, 1.0], [0.0, 0.1, 0.95], [0.0, 0.2, 0.9], [0.0, 0.3, 0.85], [0.0, 0.4, 0.8],
                      [0.0, 0.5, 0.75], [0.0, 0.6, 0.7], [0.0, 0.7, 0.65], [0.0, 0.8, 0.6], [0.0, 0.9, 0.55], [0.0, 1.0, 0.5]],
        "bone" => [[0.0, 0.0, 0.0], [0.08, 0.08, 0.11], [0.16, 0.16, 0.23], [0.25, 0.25, 0.34], [0.33, 0.33, 0.45],
                      [0.41, 0.44, 0.54], [0.5, 0.56, 0.62], [0.58, 0.67, 0.7], [0.66, 0.78, 0.78], [0.83, 0.89, 0.89],
                      [1.0, 1.0, 1.0]],
```

```

"cooper" => [[0.0,0.0,0.0],[0.13,0.08,0.05],[0.25,0.16,0.1],[0.38,0.24,0.15],[0.5,0.31,0.2],
[0.62,0.39,0.25],[0.75,0.47,0.3],[0.87,0.55,0.35],[1.0,0.63,0.4],[1.0,0.71,0.45],
[1.0,0.78,0.5]],

"greys" => [[0.0,0.0,0.0],[0.1,0.1,0.1],[0.2,0.2,0.2],[0.3,0.3,0.3],[0.4,0.4,0.4],[0.5,0.5,0.5],
[0.6,0.6,0.6],[0.7,0.7,0.7],[0.8,0.8,0.8],[0.9,0.9,0.9],[1.0,1.0,1.0]],

// "jet" as default
_ => [[0.0,0.0,0.51],[0.0,0.24,0.67],[0.01,0.49,0.78],[0.01,0.75,0.89],[0.02,1.0,1.0],
[0.51,1.0,0.5],[1.0,1.0,0.0],[0.99,0.67,0.0],[0.99,0.33,0.0],[0.98,0.0,0.0],[0.5,0.0,0.0]],

};

colors
}

```

Inside the `colormap_data` method, each colormap contains 11 RGB color arrays and has a standard colormap name. Fig.9-1 shows the color strips generated using these colormap arrays.

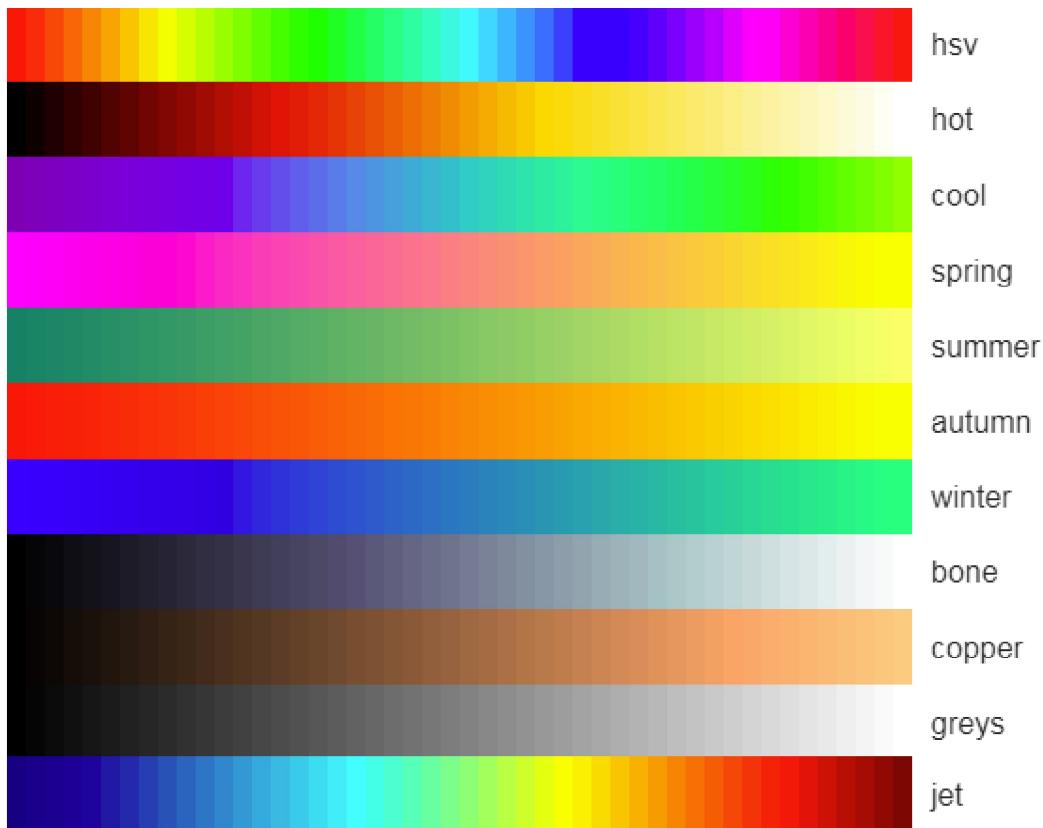


Fig.9-1. Colormaps with standard names.

## 9.2.2 Color Interpolation

In the `colormap_data` method presented in the preceding section, we assume that the 11 color arrays for each colormap are uniformly distributed in the range [0, 1]; namely, `colors[0]` represents a color at 0, while `colors[5]` represents a color at 0.5, etc. However, we have to use interpolation to get a color at 0.55 or any other arbitrary location.

For our specifically arranged colormap arrays, we can easily implement a linear color interpolation function. Add a new method called `color_lerp` to the `colormap.rs` file with the following code:

```
pub fn color_lerp(colormap_name: &str, min:f32, max:f32, mut t:f32) -> [f32; 3]{
    if t < min {
        t = min;
    }
    if t > max {
        t = max;
    }
    let tn = (t-min)/(max - min);
    let colors = colormap_data(colormap_name);
    let indx = (10.0 * tn).floor() as usize;

    if indx as f32 == 10.0 * tn {
        colors[indx]
    } else {
        let tn1 = (tn - 0.1 * indx as f32) * 10.0; // rescale
        let a = colors[indx];
        let b = colors[indx+1];
        let color_r = a[0] + (b[0] - a[0]) * tn1;
        let color_g = a[1] + (b[1] - a[1]) * tn1;
        let color_b = a[2] + (b[2] - a[2]) * tn1;
        [color_r, color_g, color_b]
    }
}
```

This method accepts an argument  $t$  whose value is in the range  $[min, max]$ , allowing us to interpolate a color for any arbitrary  $t$  value. Within the method, we first get the color arrays by calling the `colormap_data` method with the colormap name, and then make sure that the input parameter  $t$  is in the  $[min, max]$  range. Next, we normalize the  $t$  variable to the range  $[0, 1]$ , and finally we interpolate a color for the  $t$  variable. This method returns a  $[r, g, b]$  array with each color component in the range  $[0, 1]$ .

Note that here, we interpolate the color's R, G, and B components. This type of linear interpolation works for 3D vectors, but it may not work well when it comes for color. There exists a difference between the 3D coordinate space and the RGB color space: the way the human eye perceives colors. While it makes sense to connect two points in 3D space with a straight line, the same does not always apply for points in RGB color space. Interpolating the R, G, and B components independently offers no guarantee that the hue of the resulting color will look “right” to our eyes. Even so, we will use this method in this book because linearly interpolating the individual RGB components is still the easiest and most common approach.

## 9.3 Shaders with Lighting and Vertex Color

Our shader program is expected to become more complicated because we now want to incorporate lighting and vertex color into our shaders. In addition, we need to apply lighting to both the front face and back face of a surface because most surfaces are not closed 3D objects.

Add a new sub-folder called `ch09` to the `examples/` folder, then add a new `shader.wgsl` file to this newly created folder with the following code:

```
// vertex shader

[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
```

```

    normal_mat : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Input {
    [[location(0)]] pos : vec4<f32>;
    [[location(1)]] normal : vec4<f32>;
    [[location(2)]] color : vec4<f32>;
};

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_position : vec4<f32>;
    [[location(1)]] v_normal : vec4<f32>;
    [[location(2)]] v_color : vec4<f32>;
};

[[stage(vertex)]]
fn vs_main(in: Input) -> Output {
    var output: Output;
    let m_position:vec4<f32> = uniforms.model_mat * in.pos;
    output.v_position = m_position;
    output.v_normal = uniforms.normal_mat * in.normal;
    output.v_color = in.color;
    output.position = uniforms.view_project_mat * m_position;
    return output;
}

// fragment shader

[[block]] struct Uniforms {
    light_position : vec4<f32>;
    eye_position : vec4<f32>;
};
[[binding(1), group(0)]] var<uniform> frag_uniforms : Uniforms;

[[block]] struct Uniforms {
    specular_color : vec4<f32>;
    ambient_intensity: f32;
    diffuse_intensity :f32;
    specular_intensity: f32;
    specular_shininess: f32;
    is_two_side: i32;
};
[[binding(2), group(0)]] var<uniform> light_uniforms : Uniforms;

[[stage(fragment)]]
fn fs_main(in:Output) -> [[location(0)]] vec4<f32> {
    let N:vec3<f32> = normalize(in.v_normal.xyz);
    let L:vec3<f32> = normalize(frag_uniforms.light_position.xyz - in.v_position.xyz);
    let V:vec3<f32> = normalize(frag_uniforms.eye_position.xyz - in.v_position.xyz);
    let H:vec3<f32> = normalize(L + V);

    // front side
    var diffuse:f32 = light_uniforms.diffuse_intensity * max(dot(N, L), 0.0);
    var specular: f32 = light_uniforms.specular_intensity *
        pow(max(dot(N, H),0.0), light_uniforms.specular_shininess);

    // back side
    if(light_uniforms.is_two_side == 1) {

```

```

diffuse = diffuse + light_uniforms.diffuse_intensity * max(dot(-N, L), 0.0);
specular = specular + light_uniforms.specular_intensity *
    pow(max(dot(-N, H), 0.0), light_uniforms.specular_shininess);
}

let ambient:f32 = light_uniforms.ambient_intensity;
let final_color:vec3<f32> = in.v_color.xyz*(ambient + diffuse) +
    light_uniforms.specular_color.xyz * specular;
return vec4<f32>(final_color, 1.0);
}

```

This shader is very similar to the lighting shader used in the preceding chapter, except that in addition to the vertex and normal data, we also pass the color data at *location* = 2 to our vertex shader. Within the *vs\_main* method, we process this color data and store the output in the *v\_color* variable. In the fragment shader, we use the processed color data (*v\_color*) as our object's primary color to obtain the final color:

```

let final_color:vec3<f32> = in.v_color.xyz*(ambient + diffuse) +
    light_uniforms.specular_color.xyz * specular;

```

The *light\_uniforms* struct contains several fields; most of them are similar to those used in our lighting shader, except for the *is\_two\_side* field. This field controls whether the lighting is applied to one side or two sides of the surface: setting it to 1 means two-sided lighting while setting it to 0 means one-sided lighting. For two-sided lighting, we apply the lighting to the back-facing side by adding light with reversed normal vectors to both the diffuse and specular lights:

```

// front side
var diffuse:f32 = light_uniforms.diffuse_intensity * max(dot(N, L), 0.0);
var specular: f32 = light_uniforms.specular_intensity *
    pow(max(dot(N, H), 0.0), light_uniforms.specular_shininess);

// back side
if(light_uniforms.is_two_side == 1) {
    diffuse = diffuse + light_uniforms.diffuse_intensity * max(dot(-N, L), 0.0);
    specular = specular + light_uniforms.specular_intensity *
        pow(max(dot(-N, H), 0.0), light_uniforms.specular_shininess);
}

```

This way, we apply the lighting to both sides of a 3D open surface. Here, we only consider a simple case – the color of each vertex is the same on both sides of a surface. If you want to apply different colors to each side of a surface, you can let the fragment shader determine whether the front- or back-facing color should be applied by using the built-in *front\_facing* variable.

## 9.4 Simple 3D Surfaces

Mathematically, a surface draws a *z* function for each *x* and *y* coordinate in a region of interest. For each *x* and *y* value, a simple surface can have at most one *z* value.

The coordinate system in *wgpu* is oriented so that the *y*-axis indicates the “up” viewing direction. To be consistent with this notation, we will consider surfaces to be defined by functions that return a *y* value (instead of a *z* value) for each *x* and *z* coordinate in a region of interest. In order to translate a function from a system that gives *z* as a function of *x* and *y*, we simply transform a *z*-up point (*x*, *y*, *z*) into a *y*-up point (*x'*, *y'*, *z'*) using the following transformation:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} y \\ z \\ x \end{pmatrix} \quad (9.1)$$

A simple surface is defined by the  $y$ -coordinates of points above a rectangular grid in the  $x$ - $z$  plane. The surface is formed by joining adjacent points using straight lines. Simple surfaces are useful for visualizing 2D data arrays (i.e., matrices) that are too large to display in numerical form, and for graphing functions of two variables.

Typically, a surface is formed using rectangular meshes. However, *wgpu* only provides triangles as basic units for representing any surface in 3D. In order to represent a surface using traditional rectangles, we need to write our own code.

#### 9.4.1 Position, Normal, and Color Data

In this section, you will learn how to generate vertex and normal data for a 3D simple surface described by a math function  $y = f(x, z)$ . We will use a rectangular mesh or quad mesh to approximate the 3D surface. Add a new Rust file called `surface_data.rs` to the `examples/common/` folder and then add two private methods called `normalize_point` and `create_quad` to it with the following code:

## 192 | Practical GPU Graphics with wgpu and Rust

```
// color
let c0 = colormap::color_lerp(colormap_name, ymin, ymax, p0[1]);
let c1 = colormap::color_lerp(colormap_name, ymin, ymax, p1[1]);
let c2 = colormap::color_lerp(colormap_name, ymin, ymax, p2[1]);
let c3 = colormap::color_lerp(colormap_name, ymin, ymax, p3[1]);
let mut color:Vec<[f32;3]> = Vec::with_capacity(6);
color.push(c0);
color.push(c1);
color.push(c2);
color.push(c2);
color.push(c3);
color.push(c0);

// uv
let uv:Vec<[f32;3]> = Vec::with_capacity(6);

(position, normal, color, uv)
}
```

The `normalize_point` method maps the region of a surface into a region of  $scale * [-1, 1]$ , which gives a better view of the scene. We can control the size of the surface using the `scale` parameter.

The `create_quad` method takes four vertex points as its inputs. The remaining three input arguments, `ymin`, `ymax`, and `colormap_name`, are used to create the color data for each vertex.

Inside this method, we generate the vertex data using two triangles and the normal data is produced using the cross product of two diagonal vectors on the quad face. The color data is created by calling the `color_lerp` method with the `y`-component of each vertex position. We also create a placeholder for UV data that will be used later.

Now, we can create the position, normal, and color data for our simple 3D surface. Add a new public method called `simple_surface_data` to the `surface_data.rs` file and enter the following code into it:

```
pub fn simple_surface_data(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, xmin:f32, xmax:f32,
zmin:f32, zmax:f32, nx:usize, nz: usize, scale: f32, scaley: f32)
-> (Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>) {

    let dx = (xmax-xmin)/(nx as f32-1.0);
    let dz = (zmax-zmin)/(nz as f32-1.0);
    let mut ymin1: f32 = 0.0;
    let mut ymax1: f32 = 0.0;

    let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nz]; nx];
    for i in 0..nx {
        let x = xmin + i as f32 * dx;
        let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nz);
        for j in 0..nz {
            let z = zmin + j as f32 * dz;
            let pt = f(x, z);
            pt1.push(pt);
            ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
            ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
        }
        pts[i] = pt1;
    }

    let ymin = ymin1 - scaley * (ymax1 - ymin1);
    let ymax = ymax1 + scaley * (ymax1 - ymin1);

    for i in 0..nx {
```

```

for j in 0..nz {
    pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
}
}

let cmin = normalize_point([0.0, ymin1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];
let cmax = normalize_point([0.0, ymax1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];

let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);
let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);
let mut colors: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);
let uvs: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);
let uv1: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);

for i in 0..nx - 1 {
    for j in 0.. nz - 1 {
        let p0 = pts[i][j];
        let p1 = pts[i][j+1];
        let p2 = pts[i+1][j+1];
        let p3 = pts[i+1][j];

        let ( mut pos, mut norm, mut col, _uv) =
            create_quad(p0, p1, p2, p3, cmin, cmax, colormap_name);

        // positions
        positions.append(&mut pos);

        // normals
        normals.append(&mut norm);

        // colors
        colors.append(&mut col);
    }
}
(positions, normals, colors, uvs, uv1)
}

```

This method takes the function  $f$  as an input variable, which describes a 3D simple surface using a mathematical formula. It allows you to import your own function or data and use it to draw surfaces. The `colormap_name` parameter is used to specify the colormap for the vertex color. The `xmin`, `xmax`, `zmin`, and `zmax` parameters specify the data region, and the `nx` and `nz` parameters define a 2D grid in the  $x$ - $z$  plane.

This method also accepts two scaling parameters, `scale` and `scaley`. The former, used in the `normalize_point` function, allows you to easily specify the size of the 3D surface on your screen, while the `scaley` parameter is used to control the aspect ratio of the surface plot.

Inside this method, we first populate a 2D array with the function  $f(x, z)$  on the 2D data grid and then normalize the data into the region  $[-1, 1]$  by calling the `normalize_point` method. Next, we define a quad face using four adjacent vertex points and then call the `create_quad` method to generate vertex position, normal and color data for this unit cell. The entire surface is created when we iterate over all the grid points.

## 9.4.2 Common Code

As we did previously, in order to avoid code duplication, we will implement a common file called `common.rs` that can be reused to create many different 3D surfaces.

## 194 | Practical GPU Graphics with wgpu and Rust

Add a new Rust file called `common.rs` to the `examples/ch09/` folder with the following code:

```
#![allow(dead_code)]
use std:: {iter, mem };
use cgmath::{ Matrix, Matrix4, SquareMatrix };
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
};
use bytemuck:: {Pod, Zeroable, cast_slice};
#[path="../common/transforms.rs"]
mod transforms;
#[path="../common/surface_data.rs"]
mod surface;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
    is_two_side: i32,
}

pub fn light(sc:[f32;3], ambient: f32, diffuse: f32, specular: f32, shininess: f32, two_side: i32) -> Light {
    Light {
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ambient,
        diffuse_intensity: diffuse,
        specular_intensity: specular,
        specular_shininess: shininess,
        is_two_side: two_side,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
    pub color: [f32; 4],
}

pub fn vertex(p:[f32;3], n:[f32; 3], c:[f32; 3]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
        color: [c[0], c[1], c[2], 1.0],
    }
}
```

```

pub fn create_vertices(f: &dyn Fn(f32, f32) ->[f32;3], colormap_name: &str, xmin:f32, xmax:f32, zmin:f32,
zmax:f32, nx:usize, nz: usize, scale:f32, scaley:f32) -> Vec<Vertex> {
    let(pos, normal, color, _uv1) =
        surface::simple_surface_data(f, colormap_name, xmin, xmax, zmin, zmax, nx, nz, scale, scaley);
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], color[i]));
    }
    data.to_vec()
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 3] =
        wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4, 2=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

pub struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
    num_vertices: u32,
}

impl State {
    pub async fn new(window: &Window, vertex_data: &Vec<Vertex>, light_data: Light) -> Self {
        let init = transforms::InitWebGPU::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("shader.wgsl").into()),
        });

        // uniform data
        let camera_position = (3.5, 1.75, 3.5).into();
        let look_direction = (0.0,0.0,0.0).into();
        let up_direction = cgmath::Vector3::unit_y();

        let (view_mat, project_mat, _view_project_mat) =
            transforms::create_view_projection(camera_position, look_direction, up_direction,
            init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

        // create vertex uniform buffer
        // model_mat and view_projection_mat will be stored in vertex_uniform_buffer
        // inside the update function
        let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
            label: Some("Vertex Uniform Buffer"),
            size: 192,
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
            mapped_at_creation: false,
        });
    }
}

```

## 196 | Practical GPU Graphics with wgpu and Rust

```
});

// create fragment uniform buffer. here we set eye_position = camera_position and
// light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 36,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout =
    init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 2,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
        ],
    });

```

```

        }
    ],
    label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
};

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
    })
});

```

## 198 | Practical GPU Graphics with wgpu and Rust

```
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
}

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = vertex_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    view_mat,
    project_mat,
    num_vertices,
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
pub fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

pub fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [dt.sin(), dt.cos(), 0.0],
  [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;

    let normal_mat = (model_mat.invert().unwrap()).transpose();

    let model_ref:&[f32; 16] = model_mat.as_ref();
    let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
    let normal_ref:&[f32; 16] = normal_mat.as_ref();

    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
                                bytemuck::cast_slice(view_projection_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));
}

pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
```

```

let output = self.init.surface.get_current_texture()?;
let view = output
    .texture
    .create_view(&wgpu::TextureViewDescriptor::default());

let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
    size: wgpu::Extent3d {
        width: self.init.config.width,
        height: self.init.config.height,
        depth_or_array_layers: 1,
    },
    mip_level_count: 1,
    sample_count: 1,
    dimension: wgpu::TextureDimension::D2,
    format: wgpu::TextureFormat::Depth24Plus,
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    label: None,
});
let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

let mut encoder = self
    .init.device
    .create_command_encoder(&wgpu::CommandEncoderDescriptor {
        label: Some("Render Encoder"),
    });
{
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {
                    r: 0.2,
                    g: 0.247,
                    b: 0.314,
                    a: 1.0,
                }),
                store: true,
            },
        }],
        //depth_stencil_attachment: None,
        depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
            view: &depth_view,
            depth_ops: Some(wgpu::Operations {
                load: wgpu::LoadOp::Clear(1.0),
                store: false,
            }),
            stencil_ops: None,
        }),
    });
    render_pass.set_pipeline(&self.pipeline);
    render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
    render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
    render_pass.draw(0..self.num_vertices, 0..1);
}
self.init.queue.submit(iter::once(encoder.finish()));

```

## 200 | Practical GPU Graphics with wgpu and Rust

```
    output.present();

    Ok(())
}

}

pub fn run(vertex_data: &Vec<Vertex>, light_data: Light, colormap_name: &str, title: &str) {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = window::WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("ch09_{}: {}", title, colormap_name));

    let mut state = pollster::block_on(State::new(&window, &vertex_data, light_data));
    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                            ..
                        } => *control_flow = ControlFlow::Exit,
                        WindowEvent::Resized(physical_size) => {
                            state.resize(*physical_size);
                        }
                        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                            state.resize(**new_inner_size);
                        }
                        _ => {}
                    }
                }
            }
            Event::RedrawRequested(_) => {
                let now = std::time::Instant::now();
                let dt = now - render_start_time;
                state.update(dt);

                match state.render() {
                    Ok(_) => {}
                    Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
                    Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
                    Err(e) => eprintln!(":{}:", e),
                }
            }
            Event::MainEventsCleared => {
                window.request_redraw();
            }
            _ => {}
        }
    })
}
```

```
    });
}
```

This code is similar to that used in the lighting examples presented in the preceding chapter, except that the `Vertex` struct in this example contains three fields: `position`, `normal`, and `color`. Inside the `create_vertices` function, we get the vertex data by calling the `simple_surface_data` method implemented in the `surface_data.rs` file in the `examples/common/` folder, and then convert it into `Vec<Vertex>` format so that it can be used to create our 3D simple surfaces.

The `State::new` function allows us to specify the vertex data and light parameters. This means that we can use the same code to create different 3D surfaces by simply changing the attributes and data.

Here, to avoid code duplication, we also convert the content that is usually found in the `main` function into a new `run` function.

### 9.4.3 Sinc Surface

This example evaluates and plots a *sinc* function:

$$f(x, z) = \frac{\sin r}{r}, \text{ with } r = \sqrt{x^2 + z^2}$$

The *sinc* function is the Fourier transform of the rectangular function.

Add a new function called *sinc* to the `math_func.rs` file in the `examples/common/` folder with the following code:

```
pub fn sinc (x:f32, z:f32) -> [f32; 3] {
    let r = (x*x + z*z).sqrt();
    let y = if r == 0.0 { 1.0 } else { r.sin()/r };
    [x, y, z]
}
```

This method computes the *sinc* function and returns a `[f32; 3]` point.

Now, add a new Rust file called `sinc.rs` to the `examples/ch09/` folder and type the following code into it:

```
mod common;
#[path="../common/math_func.rs"]
mod math;
fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }

    let vertex_data = common::create_vertices(&math::sinc, colormap_name,
        -8.0, 8.0, -8.0, 8.0, 30, 30, 2.0, 0.3);
    let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
    common::run(&vertex_data, light_data, colormap_name, "sinc");
}
```

## 202 | Practical GPU Graphics with wgpu and Rust

Within the `main` function, we first introduce two user input strings, one for the `colormap_name`, and the other for the `is_two_side` lighting variable. They have default values of “jet” and “1”. We then generate the vertex data and light parameters by calling the `common::create_vertices` and `common::light` methods.

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch09_sinc"
path = "examples/ch09/sinc.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch09_sinc
```

This produces the results shown in Fig.9-2.

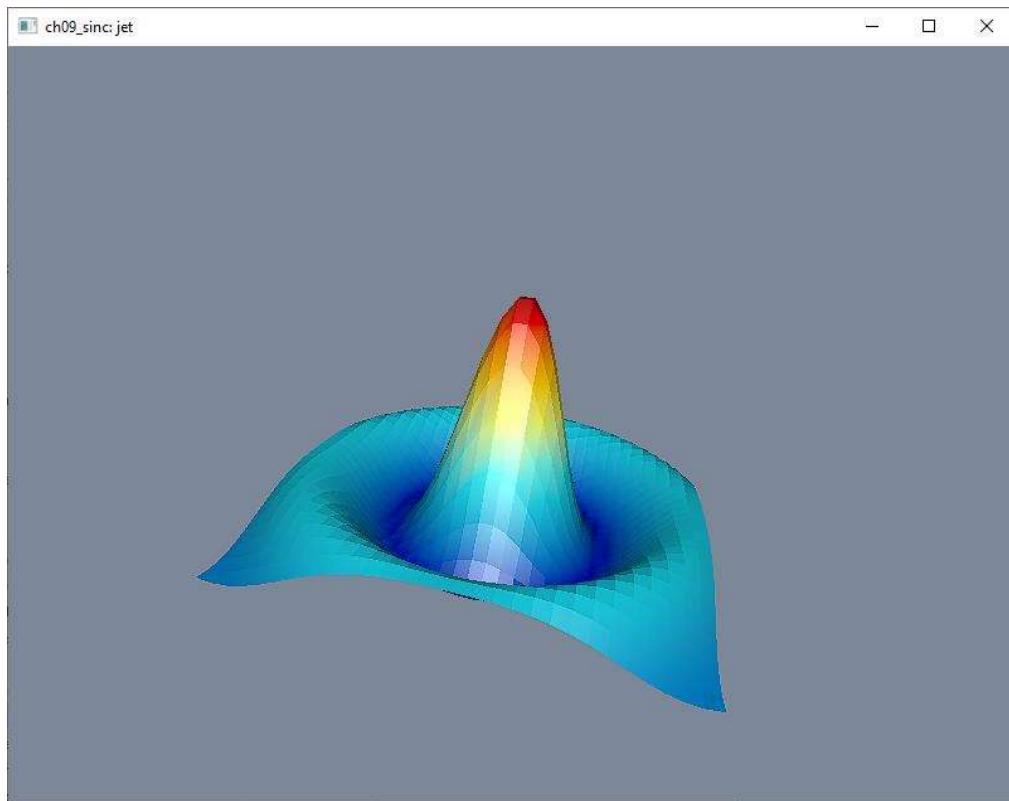


Fig.9-2. Sinc surface with default lighting and colormap.

You can play with the `sinc` surface by changing the `colormap_name`:

```
cargo run --example ch09_sinc "hsv"
```

This will create the same `sinc` surface shown in Fig.9-2, but with a different colormap of “`hsv`”.

We can also apply lighting to only the front-facing side of the surface by setting the user input parameter `is_two_side` to “0”:

```
cargo run --example ch09_sinc "hsv" "0"
```

In this case, there will be no diffuse and specular lighting on the back-facing side of the surface, as shown in Fig.9-3.

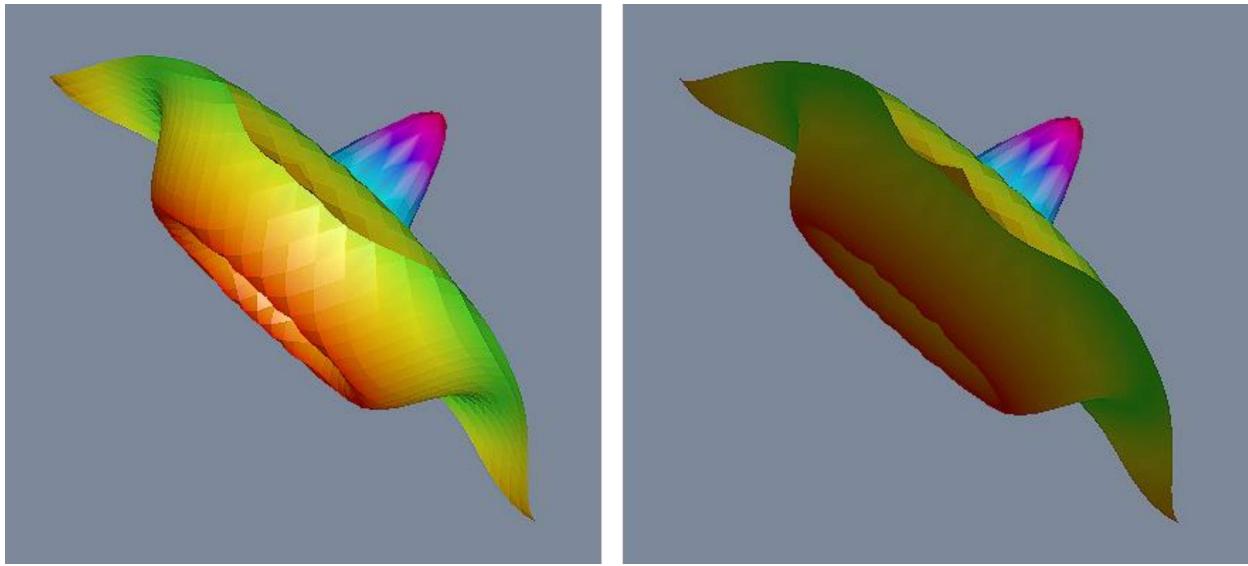


Fig.9-3. Back-facing side of the sinc surface with (left) or without (right) diffuse and specular lighting.

#### 9.4.4 Peaks Surface

In this section, we want to create a *peaks* surface described by the following mathematical formula:

$$f(x, z) = 3(1 - x)^2 e^{-[x^2 + (1+z)^2]} - 10 \left( \frac{x}{5} - x^3 - z^5 \right) e^{-(x^2 + z^2)} - \frac{1}{3} e^{-[(1+x)^2 + z^2]}$$

The *peaks* function can be obtained by translating and scaling Gaussian distributions, which is useful for demonstrating 3D surface plots.

Add a new method called *peaks* to the *math\_func.rs* file in the *examples/common/* folder and type the following code into it:

```
pub fn peaks (x:f32, z:f32) -> [f32; 3] {
    let y = 3.0*(1.0-x)*(1.0-x)*(-(x*x)-(z+1.0)*(z+1.0)).exp()-
        10.0*(x/5.0-x*x-z*z*z*z)*(-x*x-z*z).exp() - 1.0/3.0*(-(x+1.0)*(x+1.0)-z*z).exp();
    [x, y, z]
}
```

This method computes the *peaks* function and returns a *[f32; 3]* point on the *peaks* surface.

Add a new Rust file called *peaks.rs* to the *examples/ch09/* folder and enter the following content into it:

```
mod common;
#[path="../common/math_func.rs"]
mod math;

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }
}
```

```

let vertex_data = common::create_vertices(&math::peaks, colormap_name,
    -3.0, 3.0, -3.0, 3.0, 51, 51, 2.0, 0.0);
let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
common::run(&vertex_data, light_data, colormap_name, "peaks");
}

```

This code is similar to that used in our *sinc* surface example, except that we replace the *sinc* math function within the *main* method with the *peaks* math function.

Now, add the following code snippet to the *Cargo.toml* file:

```

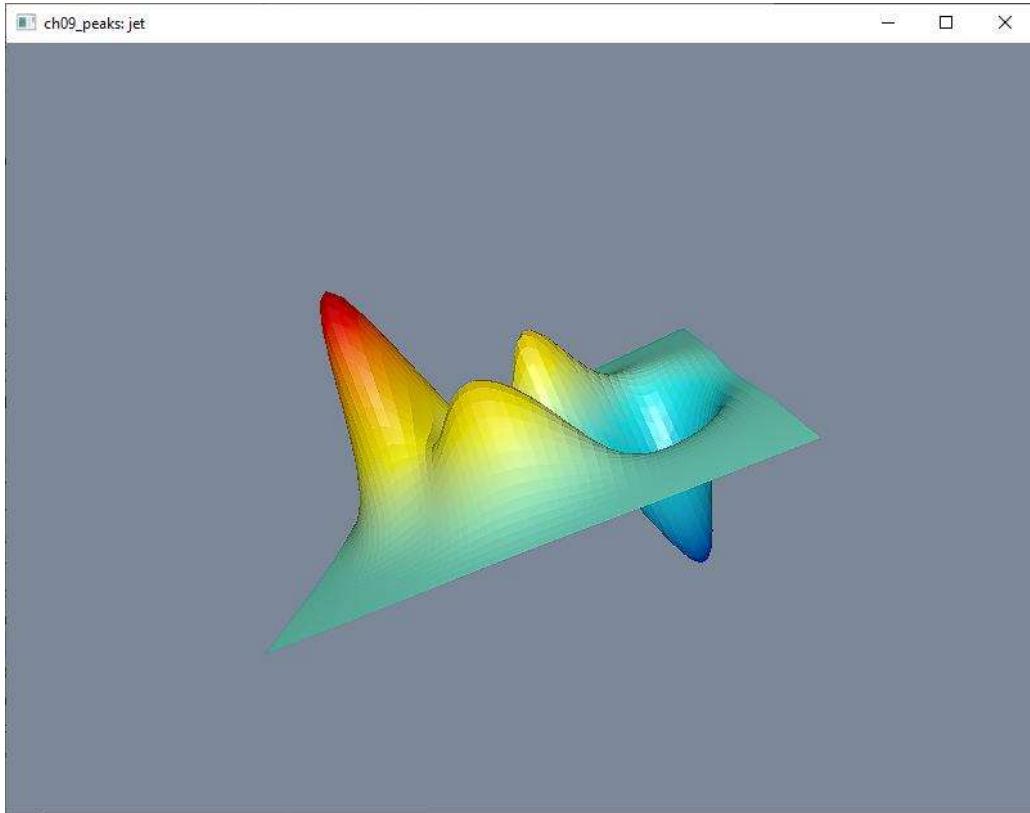
[[example]]
name = "ch09_peaks"
path = "examples/ch09/peaks.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_peaks
```

This produces the results shown in Fig.9-4.



*Fig.9-4. Peaks surface with default lighting and colormap.*

You can also create your own simple 3D surface plots using the framework we created by providing your own math functions.

## 9.5 Parametric 3D Surfaces

In the preceding sections, you learned how to create simple 3D surfaces. A key feature of this type of surface is that there is at most one  $y$  value for each pair of  $x$  and  $z$  values. However, sometimes you may want to create a complex surface of a certain shape. Complex surfaces cannot be represented by a simple mathematical formula because they may have more than one  $y$  value for certain values of  $x$  and  $z$ . This means that you cannot use the approach discussed in the preceding sections to store and display their data.

Instead, one way to represent such a surface is to use a set of parametric equations that define the  $x$ ,  $y$ , and  $z$  coordinates of points on the surface in terms of the parametric variables  $u$  and  $v$ . Many complex surfaces can be represented using parametric equations. For example, the sphere, torus, Klein bottle, and quadric surfaces are all parametric surfaces.

### 9.5.1 Vertex and Normal Data

In this section, you will learn how to generate the vertex and normal data for a parametric 3D surface described by a set of parametric equations:

$$\begin{aligned}x &= x(u, v) \\y &= y(u, v) \\z &= z(u, v)\end{aligned}$$

We will also use a rectangular mesh or quad mesh in the  $u$ - $v$  space to approximate the parametric 3D surface. Add a new method called `parametric_surface_data` to the `surface_data.rs` file in the `examples/common/` folder and type the following code into it:

```
pub fn parametric_surface_data(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, umin:f32, umax:f32,
vmin:f32, vmax:f32, nu:usize, nv: usize, xmin:f32, xmax:f32, zmin:f32, zmax:f32, scale:f32, scaley:f32)
-> (Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>) {
    let du = (umax-umin)/(nu as f32-1.0);
    let dv = (vmax-vmin)/(nv as f32-1.0);
    let mut ymin1: f32 = 0.0;
    let mut ymax1: f32 = 0.0;

    let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nv]; nu];
    for i in 0..nu {
        let u = umin + i as f32 * du;
        let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nv);
        for j in 0..nv {
            let v = vmin + j as f32 * dv;
            let pt = f(u, v);
            pt1.push(pt);
            ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
            ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
        }
        pts[i] = pt1;
    }

    let ymin = ymin1 - scaley * (ymax1 - ymin1);
    let ymax = ymax1 + scaley * (ymax1 - ymin1);

    for i in 0..nu {
        for j in 0..nv {
```

```

        pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
    }
}

let cmin = normalize_point([0.0, ymin1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];
let cmax = normalize_point([0.0, ymax1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];

let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let mut colors: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let uvs: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let uv1: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);

for i in 0..nu - 1 {
    for j in 0..nv - 1 {
        let p0 = pts[i][j];
        let p1 = pts[i+1][j];
        let p2 = pts[i+1][j+1];
        let p3 = pts[i][j+1];

        let (mut pos, mut norm, mut col, _uv) =
            create_quad(p0, p1, p2, p3, cmin, cmax, colormap_name);

        // positions
        positions.append(&mut pos);

        // normals
        normals.append(&mut norm);

        // colors
        colors.append(&mut col);
    }
}

(positions, normals, colors, uvs, uv1)
}

```

This method is essentially similar to the `simple_surface_data` method, except that we define the input function using the parametric variables  $u$  and  $v$  instead of  $x$  and  $z$ . We also pass the parameters  $umin$ ,  $umax$ ,  $vmin$ , and  $vmax$  to this method in order to specify the parametric region of interest. This means that in the parametric  $u$ - $v$  space, we create a constant  $u$ - $v$  grid with equal spacing in the respective  $u$  and  $v$  directions. The input function  $f(u, v)$  in this parametric space has at most one value for each pair of  $u$  and  $v$  values. Thus, we actually create a simple surface in the  $u$ - $v$  space using the same approach we used to create simple surfaces.

The trick to creating a parametric surface is to map the simple surface in the  $u$ - $v$  space back to the  $x$ - $y$ - $z$  coordinate system. This mapping is governed by the parametric equations. The resulting surface in real-world space can be very different from the surface in parametric space.

In the following examples, I will demonstrate how to use the `parametric_surface_data` method to create several beautiful parametric surfaces.

## 9.5.2 Klein Bottle

In topology, the Klein bottle is an example of a non-orientable surface. It can be constructed by joining two sides of a paper sheet to form a cylinder, then looping the two ends of the cylinder back through itself so that the inside and outside of the cylinder are glued together. This will form a one-sided surface,

which, if traveled upon, could be followed back to the point of origin while flipping the traveler upside down. The Klein bottle can be described using a set of parametric equations:

$$x(u, v) = \frac{2}{15} (3 + 5 \cos u \sin u) \sin v$$

$$y(u, v) = -\frac{1}{15} \sin u [3 \cos v - 3(\cos u)^2 \cos v - 48(\cos u)^4 + 48(\cos u)^6 \cos v - 60 \sin u + 5 \cos u \cos v \sin u - 5(\cos u)^3 \cos v \sin u - 80(\cos u)^5 \cos v \sin u + 80(\cos u)^7 \cos v \sin u]$$

$$z(u, v) = -\frac{2}{15} \cos u [3 \cos v - 30 \sin u + 90 (\cos u)^4 \sin u - 60(\cos u)^6 \sin u + 5 \cos u \cos v \sin u]$$

Where  $u$  and  $v$  are in the region:  $0 \leq u < \pi$  and  $0 \leq v < 2\pi$ .

Add a new method called `klein_bottle` to the `math_func.rs` file in the `examples/common` folder with the following code:

```
pub fn klein_bottle(u:f32, v:f32) -> [f32; 3] {
    let x = 2.0/15.0*(3.0+5.0*u.cos()*u.sin())*v.sin();

    let y = -1.0/15.0*u.sin()*(3.0*v.cos()-3.0*(u.cos()).powf(2.0)*v.cos()-
        48.0*(u.cos()).powf(4.0)*v.cos()+48.0*(u.cos()).powf(6.0)*v.cos()-
        60.0*u.sin()+5.0*u.cos()*v.cos()*u.sin()-5.0*(u.cos()).powf(3.0)*v.cos()*u.sin()-
        80.0*(u.cos()).powf(5.0)*v.cos()*u.sin()+80.0*(u.cos()).powf(7.0)*v.cos()*u.sin()));

    let z = -2.0/15.0*u.cos()*(3.0*v.cos()-30.0*u.sin() +
        90.0*(u.cos()).powf(4.0)*u.sin()-60.0*(u.cos()).powf(6.0)*u.sin() + 5.0*u.cos()*v.cos()*u.sin()));

    [x, y, z]
}
```

This method computes the Klein bottle function and returns a `[f32; 3]` point.

Now, we need to add a new `create_vertices_param` function to the `common.rs` file in the `examples/ch09/` folder. This function will generate vertex data by calling the `parametric_surface_data` method implemented in the `vertex_data.rs` file. Here is the code for the `create_vertices_param` function:

```
pub fn create_vertices_param(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, umin:f32, umax:f32,
    vmin:f32, vmax:f32, nu:usize, nv: usize, xmin:f32, xmax:f32, zmin:f32, zmax:f32, scale:f32, scaley:f32)
-> Vec<Vertex> {
    let(pos, normal, color, _uv, _uv1) = surface::parametric_surface_data(f, colormap_name,
        umin, umax, vmin, vmax, nu, nv, xmin, xmax, zmin, zmax, scale, scaley);
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], color[i]));
    }
    data.to_vec()
}
```

Next, add a new Rust file called `klein_bottle.rs` to the `examples/ch09/` folder and type the following code into it:

```
use std::f32::consts::PI;
mod common;
#[path="../common/math_func.rs"]
mod math;
```

```

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }

    let vertex_data = common::create_vertices_param(&math::klein_bottle, colormap_name, 0.0, PI,
        0.0, 2.0*PI, 70, 30, -2.0, 2.0, -2.0, 3.0, 2.5, 0.0);
    let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
    common::run(&vertex_data, light_data, colormap_name, "klein_bottle");
}

```

This code is similar to that used in our *sinc* surface example, except that we now generate vertex data for a Klein bottle by calling the *common::create\_vertices\_param* method with *klein\_bottle* as its input math function.

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch09_klein_bottle"
path = "examples/ch09/klein_bottle.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_klein_bottle
```

This produces the results shown in Fig.9-5.

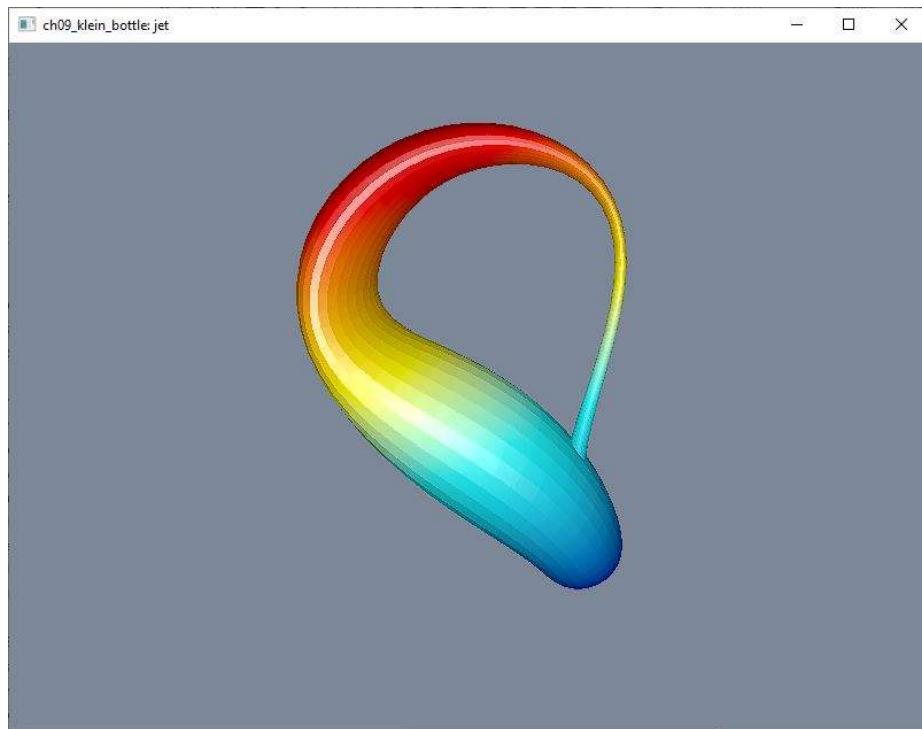


Fig.9-5. Klein bottle with default lighting and colormap.

### 9.5.3 Wellenkugel Surface

The Wellenkugel surface is represented by the following parametric equations:

$$\begin{aligned}x &= u \cos(\cos u) \sin v \\y &= u \sin(\cos u) \\z &= u \cos(\cos u) \cos v\end{aligned}$$

Where  $u$  and  $v$  are in the region:  $0 \leq u < 14.5$  and  $0 \leq v < 2\pi$ .

Wellenkugel is a German word that means “wave ball” in English. Here we will create a 3D “wave ball” surface.

Add a new method called *Wellenkugel* to the *math\_func.rs* file in the *examples/common* folder with the following code:

```
pub fn wellenkugel(u:f32, v:f32) -> [f32; 3] {
    let x = u*(u.cos().cos()*v.sin());
    let y = u*(u.cos()).sin();
    let z = u*(u.cos().cos())*v.cos();
    [x, y, z]
}
```

This method computes the *wellenkugel* function and returns a  $[f32; 3]$  point.

Add a new Rust file called *wellenkugel.rs* to the *examples/ch09/* folder and enter the following content into it:

```
mod common;
#[path="../common/math_func.rs"]
mod math;

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }

    let vertex_data = common::create_vertices_param(&math::wellenkugel, colormap_name, 0.0, 14.5,
        0.0, 5.0, 100, 50, -10.0, 10.0, -10.0, 10.0, 1.5, 0.0);
    let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
    common::run(&vertex_data, light_data, colormap_name, "Wellenkugel");
}
```

This code is similar to that used in our Klein bottle example, except that we now generate vertex data for a Wellenkugel surface using the *wellenkugel* math function.

Note that normally, the value range for the  $v$  parameter would be  $[0, 2\pi]$ , but here we set it to  $[0, 5]$  because we want to see the inside of the surface.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch09_wellenkugel"
path = "examples/ch09/wellenkugel.rs"
```

## 210 | Practical GPU Graphics with wgpu and Rust

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_wellenkugel "cool"
```

This produces results shown in Fig.9-6.

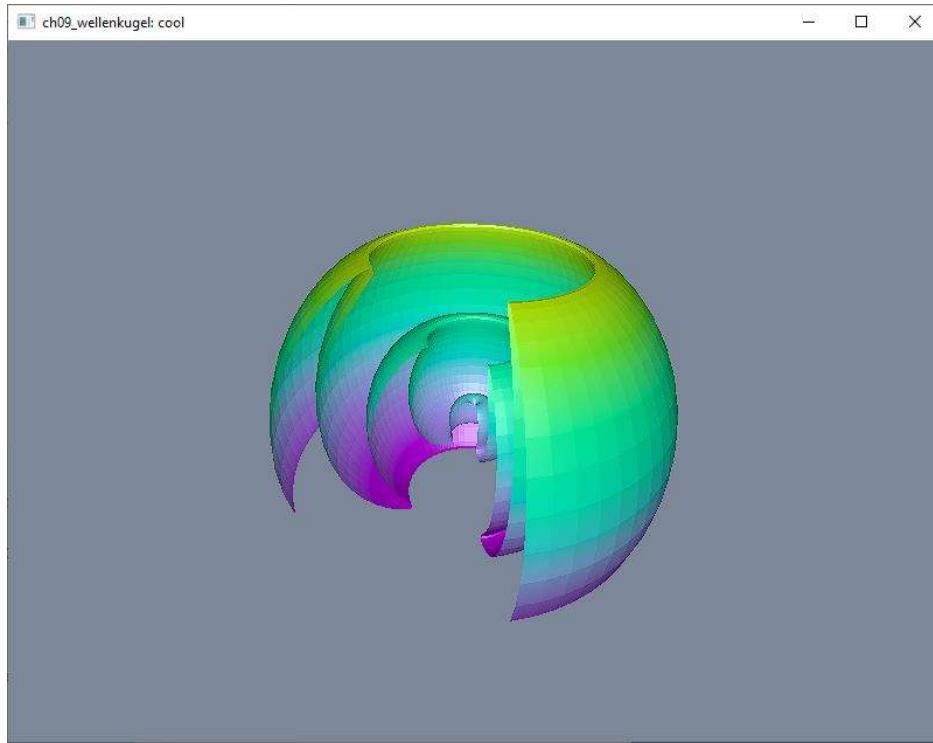


Fig.9-6. Wellenkugel surface with lighting and colormap “cool”.

### 9.5.4 Seashell Surface

The seashell surface is a type of conical surface that can be described by the following parametric equations:

$$\begin{aligned}x &= 2 \left[ -1 + \exp\left(\frac{u}{6\pi}\right) \right] \sin u \left[ \cos\left(\frac{v}{2}\right) \right]^2 \\y &= 1 - \exp\left(\frac{u}{3\pi}\right) - \sin v + \exp\left(\frac{u}{6\pi}\right) \sin v \\z &= 2 \left[ 1 - \exp\left(\frac{u}{6\pi}\right) \right] \cos u \left[ \cos\left(\frac{v}{2}\right) \right]^2\end{aligned}$$

Where  $u$  and  $v$  are in the region:  $0 \leq u < 6\pi$  and  $0 \leq v < 2\pi$ .

Add a new method called *seashell* to the *math\_func.rs* file in the *examples/common/* folder with the following code:

```
pub fn seashell(u:f32, v:f32) -> [f32; 3] {
    let x = 2.0*(-1.0+(u/(6.0*PI)).exp())*u.sin()*((v/2.0).cos()).powf(2.0));
    let y = 1.0 - (u/(3.0*PI)).exp()-v.sin() + (u/(6.0*PI)).exp()*v.sin();
    let z = 2.0*(1.0-(u/(6.0*PI)).exp())*u.cos()*(v/2.0).cos().powf(2.0);
    [x, y, z]
}
```

This method computes the *seashell* function and returns a  $[f32; 3]$  point.

Add a new Rust file called *seashell.rs* to the *examples/ch09/* folder and enter the following content into it:

```
use std::f32::consts::PI;
mod common;
#[path="../common/math_func.rs"]
mod math;

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }

    let vertex_data = common::create_vertices_param(&math::seashell, colormap_name, 0.0, 6.0*PI,
        0.0, 2.0*PI, 200, 40, -4.0, 4.0, -4.0, 4.0, 2.5, 0.0);
    let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
    common::run(&vertex_data, light_data, colormap_name, "seashell");
}
```

This code is similar to that used in our Klein bottle example, except that we now generate vertex data for a seashell surface using the *seashell* math function.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch09_seashell"
path = "examples/ch09/seashell.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_seashell "hsv"
```

This produces the results shown in Fig.9-7.

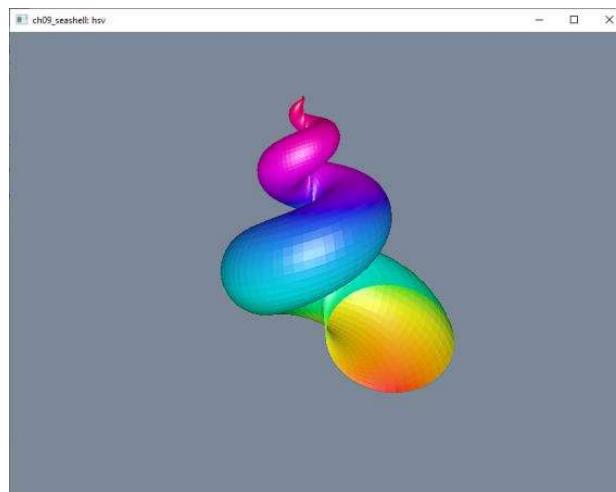


Fig.9-7. Seashell surface with lighting and colormap “hsv”.

## 9.5.5 Sievert-Enneper Surface

The Sievert-Enneper surface is a special type of spherical surfaces. It is locally isometric to the unit sphere. It can be made to fit the shape of the unit sphere, but it can never form a complete sphere – it will always have holes at its poles.

The Sievert-Enneper surface can be described by the following parametric equations:

$$x = \frac{1}{\sqrt{a}} \left[ \log \left( \tan \frac{v}{2} \right) + \frac{2(1+a) \cos v}{1+a-a \sin^2 v \cos^2 u} \right]$$

$$y = \frac{2 \sin v \sqrt{\left(1+\frac{1}{a}\right)(1+a \sin^2 u)}}{1+a-a \sin^2 v \cos^2 u} \sin \left[ \operatorname{atan}(\sqrt{1+a} \tan u) - \frac{u}{\sqrt{1+a}} \right]$$

$$z = \frac{2 \sin v \sqrt{\left(1+\frac{1}{a}\right)(1+a \sin^2 u)}}{1+a-a \sin^2 v \cos^2 u} \cos \left[ \operatorname{atan}(\sqrt{1+a} \tan u) - \frac{u}{\sqrt{1+a}} \right]$$

Where  $u$  and  $v$  are in the region:  $-\pi/2 \leq u < \pi/2$  and  $0 \leq v < \pi$ , and the constant  $a$  is in the range  $[0.05, 30]$ .

Add a new method called `sievert_enneper` to the `math_func.rs` file in the `examples/common/` folder with the following code:

```
pub fn sievert_enneper(u:f32, v:f32) -> [f32; 3] {
    const A:f32 = 1.0;

    let pu = -u/(1.0+A).sqrt() + (u.tan()*(1.0+A).sqrt()).atan();
    let auv = 2.0/(1.0+A-A*v.sin()*v.sin()*u.cos()*u.cos());
    let ruv = auv*v.sin()*((1.0+1.0/A)*(1.0+A*u.sin()*u.sin())).sqrt();

    let x = (((v/2.0).tan()).ln() + (1.0+A)*auv*v.cos()) / A.sqrt();
    let y = ruv*pu.cos();
    let z = ruv*pu.sin();

    [x, y, z]
}
```

This method computes the Sievert-Enneper function and returns a  $[f32; 3]$  point.

Add a new Rust file called `sievert_enneper.rs` to the `examples/ch09/` folder and enter the following content into it:

```
use std::f32::consts::PI;
mod common;
#[path="../common/math_func.rs"]
mod math;

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }
}
```

```

    }

let vertex_data = common::create_vertices_param(&math::siever_enneper, colormap_name,
    -PI/2.001, PI/2.001, 0.000001, PI, 60, 200, -2.0, 2.0, -2.0, 2.0, 2.5, 0.3);
let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
common::run(&vertex_data, light_data, colormap_name, "siever_enneper");
}

```

This code is similar to that used in our Klein bottle example, except that we now generate vertex data for a Sievert-Enneper surface using the *siever\_enneper* math function.

Now, add the following code snippet to the *Cargo.toml* file:

```

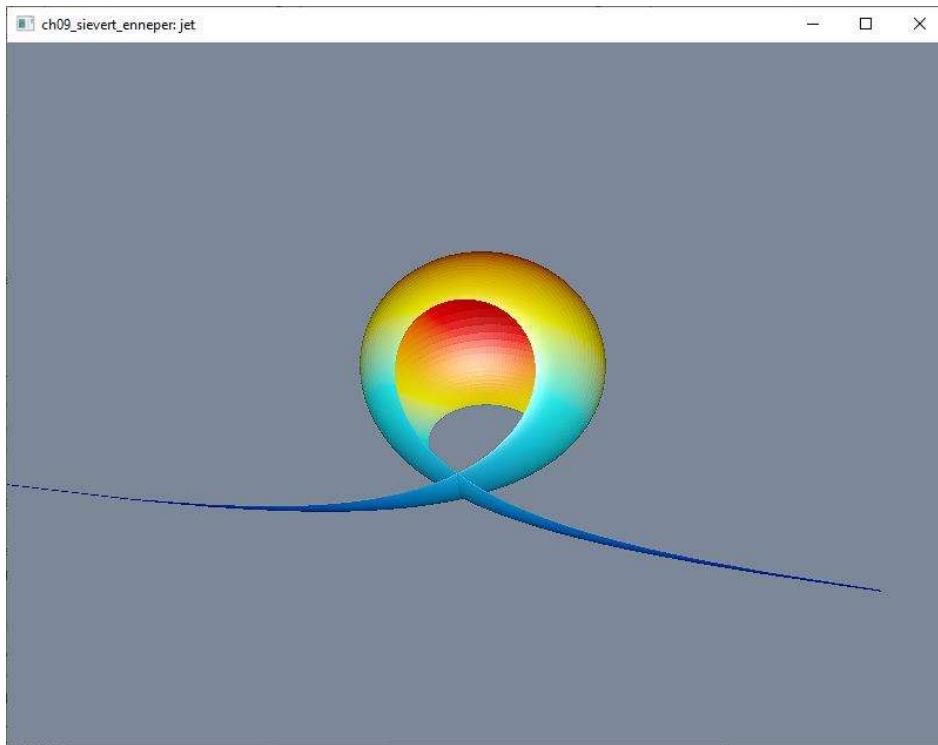
[[example]]
name = "ch09_siever_enneper"
path = "examples/ch09/siever_enneper.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_siever_enneper
```

This produces the results shown in Fig.9-8.



*Fig.9-8. Sievert-Enneper surface with default lighting and colormap.*

## 9.5.6 Breather Surface

The breather surface is a special type of pseudo-spherical surfaces. In physics, a breather is a nonlinear wave in which energy concentrates in a localized and oscillatory fashion. Most breathers are localized in space and oscillate in time, somewhat as they are “breathing”; hence the name. The breather surface discussed in this example corresponds to a solution of a non-linear wave equation.

This surface can be simulated with this set of parametric equations:

$$x = -u + \frac{2(1-a^2) \cosh(au) \sinh(au)}{a[(1-a^2) \cosh^2(au) + a^2 \sin^2(\sqrt{1-a^2} v)]}$$

$$y = \frac{2\sqrt{1-a^2} \cosh(au) [-\sqrt{1-a^2} \cos v \cos(\sqrt{1-a^2} v) - \sin v \sin(\sqrt{1-a^2} v)]}{a[(1-a^2) \cosh^2(au) + a^2 \sin^2(\sqrt{1-a^2} v)]}$$

$$z = \frac{2\sqrt{1-a^2} \cosh(au) [-\sqrt{1-a^2} \sin v \cos(\sqrt{1-a^2} v) - \cos v \sin(\sqrt{1-a^2} v)]}{a[(1-a^2) \cosh^2(au) + a^2 \sin^2(\sqrt{1-a^2} v)]}$$

Where  $u$  and  $v$  are in the region:  $-14 \leq u < 14$  and  $-12\pi \leq v < 12\pi$ , and the constant  $a$  is in the range  $[0, 1]$ .

Add a new method called *breather* to the *math\_func.rs* file in the *examples/common/* folder with the following code:

```
pub fn breather(u:f32, v:f32) -> [f32; 3] {
    const A:f32 = 0.4; // where 0 < A < 1

    let de = A*((1.0-A*A)*( (A*u).cosh()).powf(2.0)+A*A*(((1.0-A*A).sqrt()*v).sin()).powf(2.0)));

    let x = -u+(2.0*(1.0-A*A)*(A*u).cosh()*(A*u).sinh())/de;

    let y = (2.0*(1.0-A*A).sqrt()*(A*u).cosh()*(-(1.0-A*A).sqrt()*v).cos()*(1.0-A*A).sqrt()*v).cos() -
        v.sin()*(1.0-A*A).sqrt()*v).sin())/de;

    let z = (2.0*(1.0-A*A).sqrt()*(A*u).cosh()*(-(1.0-A*A).sqrt()*v).sin()*(1.0-A*A).sqrt()*v).cos() +
        v.cos()*(1.0-A*A).sqrt()*v).sin())/de;

    [x, y, z]
}
```

This method computes the *breather* function and returns a  $[f32; 3]$  point.

Add a new Rust file called *breather.rs* to the *examples/ch09/* folder and enter the following content into it:

```
use std::f32::consts::PI;
mod common;
#[path="../common/math_func.rs"]
mod math;

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }

    let vertex_data = common::create_vertices_param(&math::breather, colormap_name, -14.0, 14.0,
        -12.0*PI, 12.0*PI, 200, 200, -6.0, 6.0, -6.0, 6.0, 2.0, 0.0);
    let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
    common::run(&vertex_data, light_data, colormap_name, "breather");
}
```

```
}
```

This code is similar to that used in our Klein bottle example, except that we now generate vertex data for a breather surface using the *breather* math function.

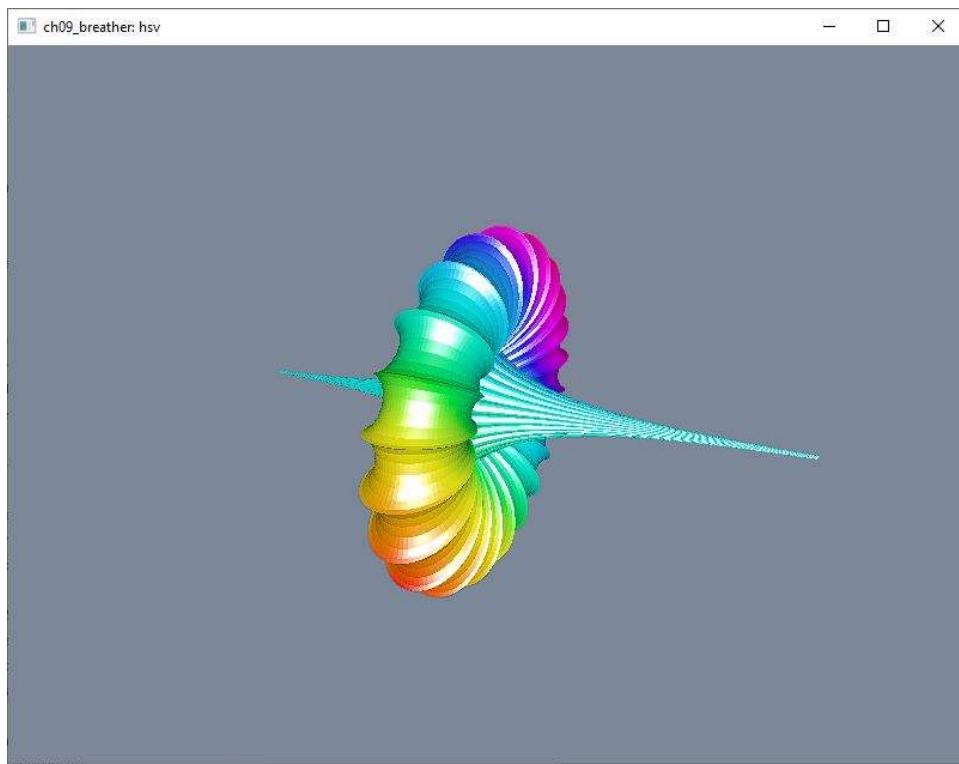
Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch09_breather"
path = "examples/ch09/breather.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch09_breather "hsv"
```

This produces the results shown in Fig.9-9.



*Fig.9-9. Breather surface with lighting and colormap “hsv”.*

In this chapter, we discussed color models and custom colormaps, as well as how to apply both lighting and colormaps to 3D surfaces. From this, we developed a general framework for creating simple and parametric 3D surfaces in *wgpu* applications. With this framework setup, you can easily create various 3D surfaces simply by providing your own math functions.



# 10 Textures

In the preceding chapter, we used colormaps and vertex colors to create various 3D shapes and surfaces. The resulting 3D objects look nice, but they are still not very realistic. If we want a more interesting and realistic look, we can do so by mapping image textures onto the surfaces of our 3D objects.

In addition, all the 3D objects we have discussed so far were created by directly drawing triangles. While we can generate complicated 3D shapes with just triangles, trying to draw every highly detailed object in computer games using triangles would be impossible due to the limitations of GPU devices. We can get around this issue with textures.

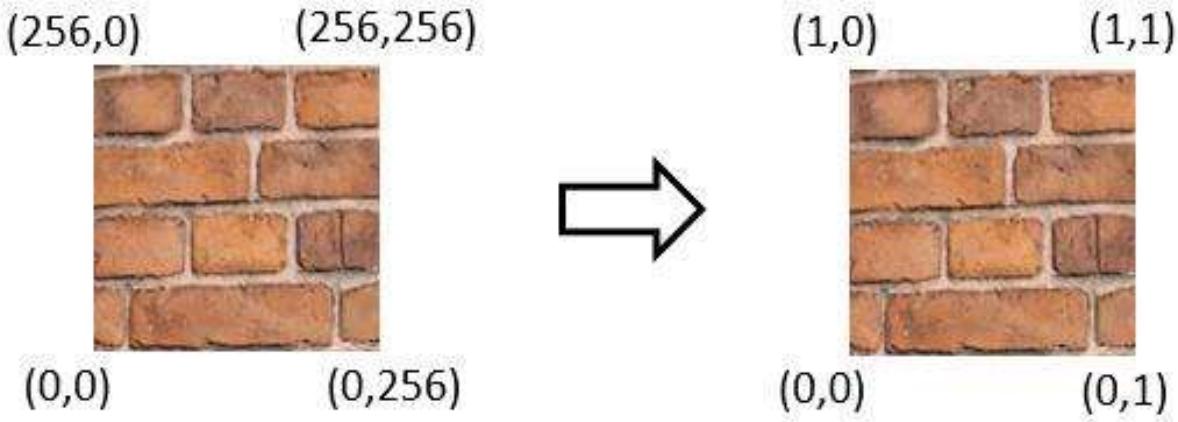
Textures play such an important role in 3D graphics that modern GPUs have support for image textures built in on the hardware level. In this chapter, we will consider 2D image textures, in which we essentially “paint” an image on to a surface. In the following sections, I will show you how to map 2D image textures onto various surfaces in *wgpu* applications

## 10.1 Texture Coordinates

Although there are multiple approaches to texture mapping, all require a sequence of steps that involve mappings between several different coordinate systems. At various stages in the process, we will be working with screen coordinates, which we use to produce the final images; object coordinates, which we use to describe the objects upon which the textures will be mapped; texture coordinates, which we use to locate positions within the texture; and parametric coordinates, which we use to specify parametric surfaces.

Texture coordinates refer to the texture image itself. A texture image is a 2D image object, usually with dimensions that are powers of 2, such as 64, 128, 256, 512, etc. With modern GPU hardware, this is no longer a hard restriction, but it is still a good practice to use textures with dimensions set to power of two.

Every texture image has its own 2D coordinate system. We represent this coordinate system with the notation  $T(u, v)$ , where  $u$  is used for the horizontal coordinate on the image and  $v$  is used for the vertical coordinate. Note that in OpenGL and WebGL,  $s$  and  $t$  are usually used for texture coordinates, while *wgpu* uses  $u$  and  $v$  for texture coordinates. Please do not confuse these with the  $u$  and  $v$  used in parametric equations. With no loss of generality, we usually scale our texture coordinates to vary over the interval  $[0, 1]$ . Fig.10-1 shows the coordinates of an image and its equivalent texture coordinates.



*Fig.10-1. Image coordinates (left) and corresponding texture coordinates (right).*

You can see from the figure that for an image size of  $256 \times 256$  pixels, all points will be divided by 256 in order to lie within the  $[0, 1]$  range. The texture coordinate  $(0.5, 0.25)$  for a  $256 \times 256$  image would refer to the image coordinate  $(128, 64)$ .

Note that values of  $u$  or  $v$  outside of the range of 0 and 1 do not lie inside the image, but such values are still valid as texture coordinates. Values over the range of  $[0, 1]$  can be used for repeating the texture image.

To map a texture onto a surface, we need a pair of texture coordinates  $(u, v)$  for each vertex. These texture coordinates indicate which point in the image is mapped to the vertex. Therefore, along with color and normal vectors, every vertex also has a pair of texture coordinates as an attribute, which are usually provided to the shader program as an attribute variable. Alternatively, texture coordinates can be generated within the shader program. If we want to use this approach, we would compute the texture coordinates from the object coordinates of the object: that is, the original vertex coordinates before transformation. This way, the texture and the object will be transformed together, and it will look like the texture is actually part of the object.

When we map an image texture onto a surface, we determine the color of each vertex by sampling the image based on a pair of texture coordinates. Thus, we can think of the image as a function that takes texture coordinates as input and returns a color as output. This means we can also use a procedural approach to map textures if we use a function that takes texture coordinates as input, and instead of looking up an image, uses a code snippet to compute a color value as output.

In *wgpu*, we can define such procedural textures in the fragment shader. For an image texture, the shader program takes a *vec2* representing a set of texture coordinates and uses a *sampler* to look up a color from the image. For a procedural texture, the shader program would simply use the *vec2* as input to practically any mathematical function that computes a *vec4* representing a color. The only limitation on the function used is that the components of the computed *vec4* are in the range  $[0, 1]$ .

## 10.2 Texture Mapping in WGPU

In *wgpu*, the most common approach used to populate texture data is reading from an image file. In WebGPU, we can use an HTML Image object to load the image texture, or use other objects such as

*HTMLCanvasElement* or *HTMLVideoElement* to set the texture data. On the other hand, in *wgpu*, we can use the *image* crate to load our image textures:

```
use image::io::Reader as ImageReader;

let img = ImageReader::open!("brick.png")?.decode()?;

This code creates a DynamicImage object by opening an image file.
```

Next, we convert the image into a vector of 8-bit RGBA bytes, which represents the actual pixel data, by calling the *img.as\_rgba8* function:

```
let rgba = img.as_rgba8().unwrap();
```

We can then use the following API function to create the texture:

```
let texture = device.create_texture(
    &wgpu::TextureDescriptor {
        label: Some("Image Texture"),
        size,
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format: wgpu::TextureFormat::Rgba8UnormSrgb,
        usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
    }
);
```

The *wgpu* API uses a *Sampler* object to encode transformation and filter information that can be used in a shader to interpret texture resource data. Filter information refers to the filtering process, which is necessary because the pixels in a texture are not usually a one-to-one match with the pixels on a surface, resulting in the texture being stretched or distorted when it is mapped onto the surface. For example, multiple texture pixels, or texels, may end up being mapped to the same surface pixel. Then the program must decide which color is applied to the surface pixel by calculating from the colors of all of the texels that map to it: in other words, the program must “filter” the texels. In this case, we would use a minification filter because we need to shrink the texture. In the reverse case where one texel covers more than one pixel on the surface, we need to magnify the texture, so we would use a magnification filter.

In *wgpu*, we can set two properties, *min\_filter* and *mag\_filter*, which let us specify the sampling behavior. The *min\_filter* property tells our sampler what to do when the texels mapped to a pixel, or sample footprint, is larger than one texel, while *mag\_filter* tells it what to do when the sample footprint is smaller than or equal to one texel. We can choose one of two options for the *FilterMode*: *nearest* and *linear*. The *nearest* mode returns the value of the texel nearest to the texture coordinates, while the *linear* mode selects two texels in each dimension and returns a linear interpolation between their values.

Generally, linear filtering gives more accurate results, but it can become very inefficient when a large texture is applied to a much smaller surface, and it then has to compute the average of many texels for a single surface pixel. The solution to this issue is to use mipmaps. A mipmap is simply a scaled-down version of a texture. A complete set of mipmaps contains the full-size texture, a half-sized version whose dimensions are divided by two, a quarter-sized version, a one-eighth-sized version, and so on, all the way down to the final mipmap, which consists of a single pixel. Mipmaps become very small very fast, and in fact a full set of mipmaps only uses one-third more memory than the original texture.

Note that we use mipmaps only for minification filtering. We can also choose a *FilterMode* that specifies how our sampler behaves when sampling between two mipmap levels by setting the *mipmap\_filter* property. The following code snippet shows how to create a sampler in *wgpu*:

## 220 | Practical GPU Graphics with wgpu and Rust

```
let sampler = device.create_sampler(  
    &wgpu::SamplerDescriptor {  
        address_mode_u: u_mode,  
        address_mode_v: v_mode,  
        mag_filter: wgpu::FilterMode::Linear,  
        min_filter: wgpu::FilterMode::Nearest,  
        mipmap_filter: wgpu::FilterMode::Nearest,  
        ..Default::default()  
    }  
);
```

Here, the `address_mode` property describes the behavior of the sampler if the sample footprint extends beyond the bounds of the sampled texture. There is a separate address mode for each direction in the texture coordinate system. The address mode has three options:

- `ClampToEdge`: Texture coordinates are clamped between [0, 1].
- `Repeat`: Texture coordinates wrap to the other side of the texture.
- `MirrorRepeat`: Texture coordinates wrap to the other side of the texture, but the texture is flipped when the integer part of the coordinate is odd. This mode can eliminate visible seams between the texture copies.

In `wgpu`, texture coordinates are usually input to the vertex shader as an attribute of type `vec2`, and communicated to the fragment shader as a varying variable. The vertex shader will then copy the value of the attribute into the varying variable. In the fragment shader, the texture coordinates are used to sample the texture. The WGLS function for sampling an ordinary texture can be expressed as:

```
textureSample(texture_data, texture_sampler, texture_coordinates);
```

Where `texture_data` is a uniform variable of type `texture_2d<f32>` that represents the texture, `texture_sampler` is a uniform variable of type `sampler`, and `texture_coordinates` is a `vec2` object that contains the texture coordinates. The returned value is an RGBA color, represented as a value of data type `vec4`.

Here, I will put our texture mapping code into a common file called `texture_data.rs` in the `examples/common/` folder. Here is the code for this common file:

```
use image::GenericImageView;  
use image::io::Reader as ImageReader;  
use anyhow::*;

pub struct Texture {  
    pub texture: wgpu::Texture,  
    pub view: wgpu::TextureView,  
    pub sampler: wgpu::Sampler,  
}

impl Texture {  
    pub fn create_texture_data(device:&wgpu::Device, queue: &wgpu::Queue, img_file: &str,  
        u_mode:wgpu::AddressMode, v_mode:wgpu::AddressMode,) -> Result<Self> {  
  
        let img = ImageReader::open(img_file)?.  
            decode()?  
        let rgba = img.as_rgba8().unwrap();  
        let dimensions = img.dimensions();  
  
        let size = wgpu::Extent3d {  
            width: dimensions.0,  
            height: dimensions.1,
```

```

        depth_or_array_layers: 1,
    );
    let texture = device.create_texture(
        &wgpu::TextureDescriptor {
            label: Some("Image Texture"),
            size,
            mip_level_count: 1,
            sample_count: 1,
            dimension: wgpu::TextureDimension::D2,
            format: wgpu::TextureFormat::Rgba8UnormSrgb,
            usage: wgpu::TextureUsages::TEXTURE_BINDING | wgpu::TextureUsages::COPY_DST,
        }
    );
    queue.write_texture(
        wgpu::ImageCopyTexture {
            aspect: wgpu::TextureAspect::All,
            texture: &texture,
            mip_level: 0,
            origin: wgpu::Origin3d::ZERO,
        },
        rgba,
        wgpu::ImageDataLayout {
            offset: 0,
            bytes_per_row: std::num::NonZeroU32::new(4 * dimensions.0),
            rows_per_image: std::num::NonZeroU32::new(dimensions.1),
        },
        size,
    );
}

let view = texture.create_view(&wgpu::TextureViewDescriptor::default());
let sampler = device.create_sampler(
    &wgpu::SamplerDescriptor {
        address_mode_u: u_mode,
        address_mode_v: v_mode,
        mag_filter: wgpu::FilterMode::Linear,
        min_filter: wgpu::FilterMode::Nearest,
        mipmap_filter: wgpu::FilterMode::Nearest,
        ..Default::default()
    }
);
Ok(Self { texture, view, sampler })
}
}

```

We have already discussed much of the content of this method. Here, we first define the *Texture* struct, which includes three public fields: *texture*, *view*, and *sampler*; then we implement a public method called *create\_texture\_data* for our *Texture* struct. This method accepts *img\_file* and *AddressMode* in the *u* and *v* directions as its input arguments. It returns the texture, texture view, and sampler, which will be used to create textured 3D objects in our *wgpu* applications.

Note that the *wgpu::ImageDataLayout* attribute in the *write\_texture* method contains the bytes per row and rows per image, which may be tricky to figure out. A row is one row of pixels or compressed blocks in the *x* direction. The *bytes\_per\_row* field is required if there are multiple rows, and it must be a multiple of 256. An image is one layer in the *z* direction of a 3D image or 2D array texture. The *rows\_per\_image* field indicates the number of rows that make up a single image. This field is required if there are multiple images, that is, if the depth is greater than one.

## 10.3 Shaders with Texture

Our shader program is about to become more complicated because we now want to incorporate texture and lighting into the shaders.

Add a new sub-folder called `ch10` to the `examples/` folder and then add a `shader.wgsl` file to this newly created folder with the following code:

```
// vertex shader

[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
    normal_mat : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Input {
    [[location(0)]] pos : vec4<f32>;
    [[location(1)]] normal : vec4<f32>;
    [[location(2)]] uv : vec2<f32>;
};

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_position : vec4<f32>;
    [[location(1)]] v_normal : vec4<f32>;
    [[location(2)]] v_uv : vec2<f32>;
};

[[stage(vertex)]]
fn vs_main(in: Input) -> Output {
    var output: Output;
    let m_position:vec4<f32> = uniforms.model_mat * in.pos;
    output.v_position = m_position;
    output.v_normal = uniforms.normal_mat * in.normal;
    output.v_uv = in.uv;
    output.position = uniforms.view_project_mat * m_position;
    return output;
}

// fragment shader

[[block]] struct Uniforms {
    light_position : vec4<f32>;
    eye_position : vec4<f32>;
};

[[binding(1), group(0)]] var<uniform> frag_uniforms : Uniforms;

[[block]] struct Uniforms {
    specular_color : vec4<f32>;
    ambient_intensity: f32;
    diffuse_intensity :f32;
    specular_intensity: f32;
    specular_shininess: f32;
    is_two_side: i32;
};

[[binding(2), group(0)]] var<uniform> light_uniforms : Uniforms;
```

```
[[binding(0), group(1)]] var texture_data : texture_2d<f32>;
[[binding(1), group(1)]] var texture_sampler : sampler;

[[stage(fragment)]]
fn fs_main(in:Output) -> [[location(0)]] vec4<f32> {
    let texture_color:vec4<f32> = textureSample(texture_data, texture_sampler, in.v_uv);

    let N:vec3<f32> = normalize(in.v_normal.xyz);
    let L:vec3<f32> = normalize(frag_uniforms.light_position.xyz - in.v_position.xyz);
    let V:vec3<f32> = normalize(frag_uniforms.eye_position.xyz - in.v_position.xyz);
    let H:vec3<f32> = normalize(L + V);

    // front side
    var diffuse:f32 = light_uniforms.diffuse_intensity * max(dot(N, L), 0.0);
    var specular: f32 = light_uniforms.specular_intensity *
        pow(max(dot(N, H),0.0), light_uniforms.specular_shininess);

    // back side
    var is_two_side:i32 = light_uniforms.is_two_side;
    if(is_two_side == 1) {
        diffuse = diffuse + light_uniforms.diffuse_intensity * max(dot(-N, L), 0.0);
        specular = specular + light_uniforms.specular_intensity *
            pow(max(dot(-N, H),0.0), light_uniforms.specular_shininess);
    }

    let ambient:f32 = light_uniforms.ambient_intensity;
    let final_color:vec3<f32> = texture_color.rgb*(ambient + diffuse) +
        light_uniforms.specular_color.xyz * specular;
    return vec4<f32>(final_color, 1.0);
}
```

This shader code is very similar to the lighting shader used in Chapter 8, except that in addition to the vertex position and normal data, here we also pass the UV data at *location(2)* to our vertex shader. Within the *vs\_main* method, we process this UV data and store the output in the *v\_uv* field of the *Output* struct.

In the fragment shader, we use the processed UV data (*v\_uv*) as our input to get the RGB color from the texture:

```
let texture_color:vec4<f32> = textureSample(texture_data, texture_sampler, in.v_uv);
```

The final color is obtained by

```
let final_color:vec3<f32> = texture_color.rgb*(ambient + diffuse) +
    light_uniforms.specular_color.xyz * specular;
```

## 10.4 Common Code

As we did in the preceding chapter, to avoid code duplication, we will implement a common file called *common.rs* that can be reused to create different textured 3D shapes.

Add a new Rust file called *common.rs* to the *examples/ch10/* folder and enter the following content into it:

```
#![allow(dead_code)]
use std::{iter, mem};
use cgmath::{Matrix, Matrix4, SquareMatrix};
use wgpu::util::DeviceExt;
use winit::{
    event::*,
```

## 224 | Practical GPU Graphics with wgpu and Rust

```
window::Window,
event_loop::{ControlFlow, EventLoop},
};

use bytemuck:: {Pod, Zeroable, cast_slice};
#[path="../common/transforms.rs"]
pub mod transforms;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/texture_data.rs"]
mod texture;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
    is_two_side: i32,
}

pub fn light(sc:[f32;3], ambient: f32, diffuse: f32, specular: f32, shininess: f32, two_side: i32) -> Light {
    Light {
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ambient,
        diffuse_intensity: diffuse,
        specular_intensity: specular,
        specular_shininess: shininess,
        is_two_side: two_side,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
    pub uv: [f32; 2],
}

pub fn vertex(p:[f32;3], n:[f32; 3], t:[f32; 2]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
        uv: [t[0], t[1]],
    }
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 3] =
        wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4, 2=>Float32x2];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
```

```

        attributes: &Self::ATTRIBUTES,
    }
}
}

pub struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
    num_vertices: u32,
}

impl State {
    pub async fn new(window: &Window, vertex_data: &Vec<Vertex>, light_data: Light, img_file: &str,
        u_mode:wgpu::AddressMode, v_mode:wgpu::AddressMode) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        // create image texture
        let image_texture = texture::Texture::create_texture_data(&init.device,
            &init.queue,img_file, u_mode, v_mode).unwrap();
        let texture_bind_group_layout =
            init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
                entries: &[
                    wgpu::BindGroupLayoutEntry {
                        binding: 0,
                        visibility: wgpu::ShaderStages::FRAGMENT,
                        ty: wgpu::BindingType::Texture {
                            multisampled: false,
                            view_dimension: wgpu::TextureViewDimension::D2,
                            sample_type: wgpu::TextureSampleType::Float { filterable: true },
                        },
                        count: None,
                    },
                    wgpu::BindGroupLayoutEntry {
                        binding: 1,
                        visibility: wgpu::ShaderStages::FRAGMENT,
                        ty: wgpu::BindingType::Sampler {
                            comparison: false,
                            filtering: true,
                        },
                        count: None,
                    },
                ],
                label: Some("Texture Bind Group Layout"),
            });
        let texture_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
            layout: &texture_bind_group_layout,
            entries: &[
                wgpu::BindGroupEntry {
                    binding: 0,
                    resource: wgpu::BindingResource::TextureView(&image_texture.view),
                },
            ]
        });
    }
}

```

## 226 | Practical GPU Graphics with wgpu and Rust

```
wgpu::BindGroupEntry {
    binding: 1,
    resource: wgpu::BindingResource::Sampler(&image_texture.sampler),
},
],
label: Some("Texture Bind Group"),
});

let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!("shader.wgsl").into()),
});

// uniform data
let camera_position = (2.5, 1.25, 2.5).into();
let look_direction = (0.0, 0.0, 0.0).into();
let up_direction = cgmath::Vector3::unit_y();

let (view_mat, project_mat, _view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
        init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

// create vertex uniform buffer
// model_mat and view_projection_mat will be stored in vertex_uniform_buffer inside
// the update function
let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 192,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// create fragment uniform buffer. here we set eye_position = camera_position and
// light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 36,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout =
    init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
```

```

entries: &[
    wgpu::BindGroupLayoutEntry {
        binding: 0,
        visibility: wgpu::ShaderStages::VERTEX,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    },
    wgpu::BindGroupLayoutEntry {
        binding: 1,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    },
    wgpu::BindGroupLayoutEntry {
        binding: 2,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    }
],
label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout, &texture_bind_group_layout],
    push_constant_ranges: &[],
});

```

## 228 | Practical GPU Graphics with wgpu and Rust

```
let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
};

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = vertex_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    view_mat,
    project_mat,
    num_vertices,

    image_texture,
    texture_bind_group
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
    }
}
```

```

        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(new_size.width as f32 /
            new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
pub fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

pub fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [dt.sin(), dt.cos(), 0.0],
        [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;

    let normal_mat = (model_mat.invert().unwrap()).transpose();

    let model_ref:&[f32; 16] = model_mat.as_ref();
    let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
    let normal_ref:&[f32; 16] = normal_mat.as_ref();

    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
        bytemuck::cast_slice(view_projection_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));
}

pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format: wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });
    {

```

## 230 | Practical GPU Graphics with wgpu and Rust

```
let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
    label: Some("Render Pass"),
    color_attachments: &[wgpu::RenderPassColorAttachment {
        view: &view,
        resolve_target: None,
        ops: wgpu::Operations {
            load: wgpu::LoadOp::Clear(wgpu::Color {
                r: 0.2,
                g: 0.247,
                b: 0.314,
                a: 1.0,
            }),
            store: true,
        },
    }],
    //depth_stencil_attachment: None,
    depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
        view: &depth_view,
        depth_ops: Some(wgpu::Operations {
            load: wgpu::LoadOp::Clear(1.0),
            store: false,
        }),
        stencil_ops: None,
    }),
});
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.set_bind_group(1, &self.texture_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

pub fn run(vertex_data: &Vec<Vertex>, light_data: Light, file_name: &str, u_mode:wgpu::AddressMode, v_mode:wgpu::AddressMode, title: &str) {
    let path = "examples/ch10/assets/";
    let img_file = [path, file_name].join("");

    env_logger::init();
    let event_loop = EventLoop::new();
    let window = winit::window::WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("ch10_{}: {}", title, file_name));

    let mut state = pollster::block_on(State::new(&window, &vertex_data, light_data, &img_file, u_mode, v_mode));
    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
```

```

if !state.input(event) {
    match event {
        WindowEvent::CloseRequested
        | WindowEvent::KeyboardInput {
            input:
                KeyboardInput {
                    state: ElementState::Pressed,
                    virtual_keycode: Some(VirtualKeyCode::Escape),
                    ..
                },
                ..
        } => *control_flow = ControlFlow::Exit,
        WindowEvent::Resized(physical_size) => {
            state.resize(*physical_size);
        }
        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
            state.resize(**new_inner_size);
        }
        _ => {}
    }
}
Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{:?}", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}

```

This code is similar to that used in the colormap examples presented in the preceding chapter. Note that the code calls the `Textures::create_texture_data` method to generate texture and sampler data. This method was implemented in the `texture_data.rs` file in the preceding section.

Next, we create a separate `texture_bind_group` whose layout has two entries: one is for a sampled texture at binding 0, and the other for a sampler at binding 1. Both of these bindings are visible only to the fragment shader as specified by the `visibility` attribute:

```
    visibility: wgpu::ShaderStages::FRAGMENT,
```

Now, our pipeline layout needs to include these two bind groups: the uniform bind group and texture bind group, as specified in the following code snippet:

```

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout, &texture_bind_group_layout],
    push_constant_ranges: &[],
});
```

Finally, we need to add our texture bind group to the render pass:

```
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.set_bind_group(1, &self.texture_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
```

Thus, we can access the texture data in the fragment shader from *group(1)*, which represents the texture bind group:

```
[[binding(0), group(1)]] var texture_data : texture_2d<f32>;
[[binding(1), group(1)]] var texture_sampler : sampler;
```

Here, to avoid code duplication, we also convert the content that is usually found in the *main* function into a new *run* function.

## 10.5 Simple 3D Shapes

In this section, I will demonstrate how to map 2D texture images onto the surfaces of several 3D shapes, including a cube, sphere, and cylinder. In order to create textures, I have added nine 512×512 pictures in PNG format to the *examples/ch10/assets/* folder.

### 10.5.1 Cube with Texture

Texture mapping requires the texture coordinates, which are very easy to create for a cube object. Here, we want to map our entire image to each face of the cube.

Previously, we implemented the *cube\_data* method in the *vertex\_data.rs* file in the *examples/common/* folder. Now, add the following code to this method to create the UV (texture coordinates) data:

```
let uvs= [
    // front
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // right
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // back
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // left
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // top
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
    // bottom
    [0, 0], [1, 0], [0, 1], [0, 1], [1, 0], [1, 1],
];
```

Now, add a new Rust file called *cube.rs* to the *examples/ch10/* folder and type the following code into it:

```
mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;
```

```

fn vertex(p:[i8; 3], n: [i8; 3], t:[i8; 2]) -> common::Vertex {
    common::Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        normal: [n[0] as f32, n[1] as f32, n[2] as f32, 1.0],
        uv: [t[0] as f32, t[1] as f32],
    }
}

fn create_vertices() -> Vec<common::Vertex> {
    let(pos, _col, uv, normal) = vertex_data::cube_data();
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "brick.png";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }

    let vertex_data = create_vertices();
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "cube");
}

```

Here, we first create the `vertex` method which converts `i8` data into `f32` data, and then implement the `create_vertices` method which gets the vertex data by calling the `vertex_data::cube_data` function and converts the data into the proper format to be used in creating our textured cube.

Inside the `main` function, we set a user specified field that allows the user to select different image files. We then generate the vertex data and lighting model data by calling the `create_vertices` and `common::light` methods respectively.

Now, add the following code snippet to the `Cargo.toml` file:

```

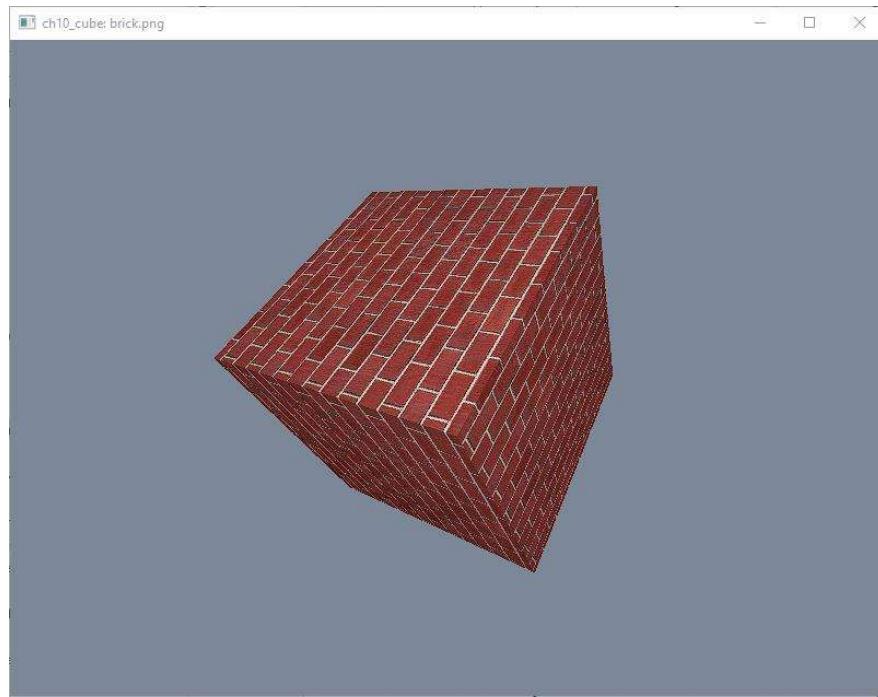
[[example]]
name = "ch10_cube"
path = "examples/ch10/cube.rs"

```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch10_cube
```

This produces the results shown in Fig.10-2.

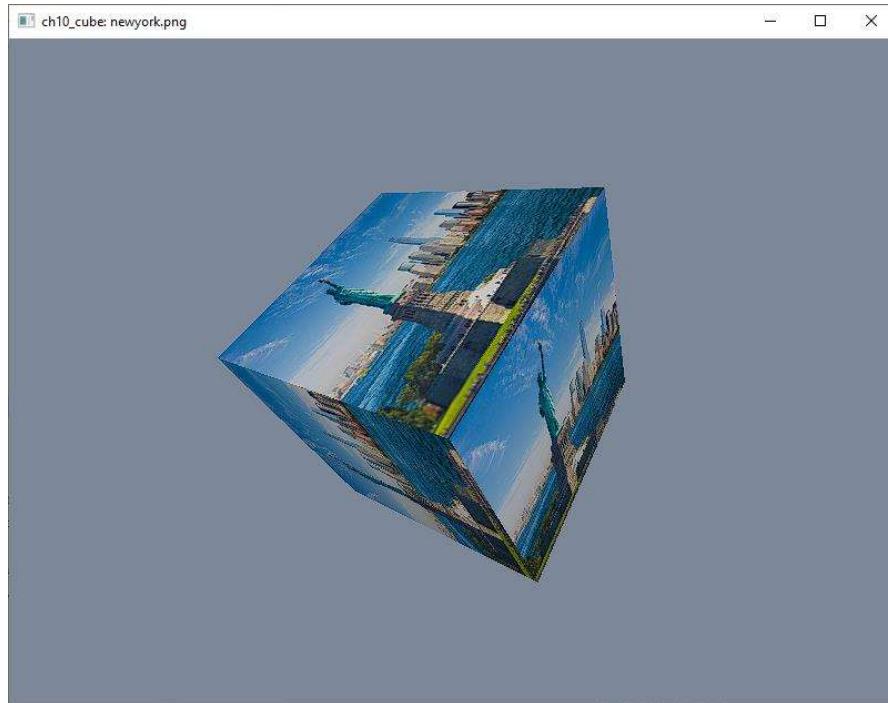


*Fig.10-2. Cube with brick image texture.*

We can also change the texture image by specifying a different image file:

```
cargo run --example ch10_cube "newyork.png"
```

This produces the results shown in Fig.10-3.



*Fig.10-3. Cube with New York image texture.*

## 10.5.2 Sphere with Texture

Previously, we implemented the `sphere_data` method in the `vertex_data.rs` file. Now, add the following highlighted code to this method to create the UV (texture coordinates) data for our sphere:

```
pub fn sphere_data(r: f32, u:usize, v:usize) -> (Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 2]>) {
    let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4*(u - 1)*(v - 1)) as usize);
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4*(u - 1)*(v - 1)) as usize);
    let mut uvs: Vec<[f32; 2]> = Vec::with_capacity((4*(u - 1)*(v - 1)) as usize);

    for i in 0..u - 1 {
        for j in 0..v - 1 {
            let theta = i as f32 * 180.0/(u as f32 - 1.0);
            let phi = j as f32 * 360.0/(v as f32 - 1.0);
            let theta1 = (i as f32 + 1.0) * 180.0/(u as f32 - 1.0);
            let phi1 = (j as f32 + 1.0) * 360.0/(v as f32 - 1.0);
            let p0 = math_func::sphere_position(r, Deg(theta), Deg(phi));
            let p1 = math_func::sphere_position(r, Deg(theta1), Deg(phi));
            let p2 = math_func::sphere_position(r, Deg(theta1), Deg(phi1));
            let p3 = math_func::sphere_position(r, Deg(theta), Deg(phi1));

            // positions
            positions.push(p0);
            positions.push(p1);
            positions.push(p3);
            positions.push(p1);
            positions.push(p2);
            positions.push(p3);

            // normals
            normals.push([p0[0]/r, p0[1]/r, p0[2]/r]);
            normals.push([p1[0]/r, p1[1]/r, p1[2]/r]);
            normals.push([p3[0]/r, p3[1]/r, p3[2]/r]);
            normals.push([p1[0]/r, p1[1]/r, p1[2]/r]);
            normals.push([p2[0]/r, p2[1]/r, p2[2]/r]);
            normals.push([p3[0]/r, p3[1]/r, p3[2]/r]);

            // uvs
            let u0 = 0.5+((p0[0]/r).atan2(p0[2]/r))/PI/2.0;
            let u1 = 0.5+((p1[0]/r).atan2(p1[2]/r))/PI/2.0;
            let u2 = 0.5+((p2[0]/r).atan2(p2[2]/r))/PI/2.0;
            let u3 = 0.5+((p3[0]/r).atan2(p3[2]/r))/PI/2.0;
            let v0 = 0.5-(p0[1]/r).asin()/PI;
            let v1 = 0.5-(p1[1]/r).asin()/PI;
            let v2 = 0.5-(p2[1]/r).asin()/PI;
            let v3 = 0.5-(p3[1]/r).asin()/PI;

            uvs.push([u0, v0]);
            uvs.push([u1, v1]);
            uvs.push([u3, v3]);
            uvs.push([u1, v1]);
            uvs.push([u2, v2]);
            uvs.push([u3, v3]);
        }
    }
    (positions, normals, uvs)
}
```

## 236 | Practical GPU Graphics with wgpu and Rust

Note that for any point  $P(x, y, z)$  on a sphere with radius  $r$ , the UV coordinates can be calculated using the following formula:

$$u = 0.5 + \frac{\arctan2(x/r, z/r)}{2\pi}$$

$$v = 0.5 - \frac{\arcsin(y/r)}{\pi}$$

Now, add a new Rust file called *sphere.rs* to the *examples/ch10/* folder and enter the following content into it:

```
mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices() -> Vec<common::Vertex> {
    let(pos, normal, uv) = vertex_data::sphere_data(1.7, 30, 50);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "earth.png";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }

    let vertex_data = create_vertices();
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "sphere");
}
```

This code first implements the *create\_vertices* method which gets the vertex data by calling the *vertex\_data::sphere\_data* function and converts the data into the proper format to be used in creating our textured sphere.

Inside the *main* function, we set a user specified field for the texture image file. We then generate the vertex data and lighting model data by calling the *create\_vertices* and *common::light* methods respectively.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch10_sphere"
path = "examples/ch10/sphere.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch10_sphere
```

This produces the results shown in Fig.10-4.

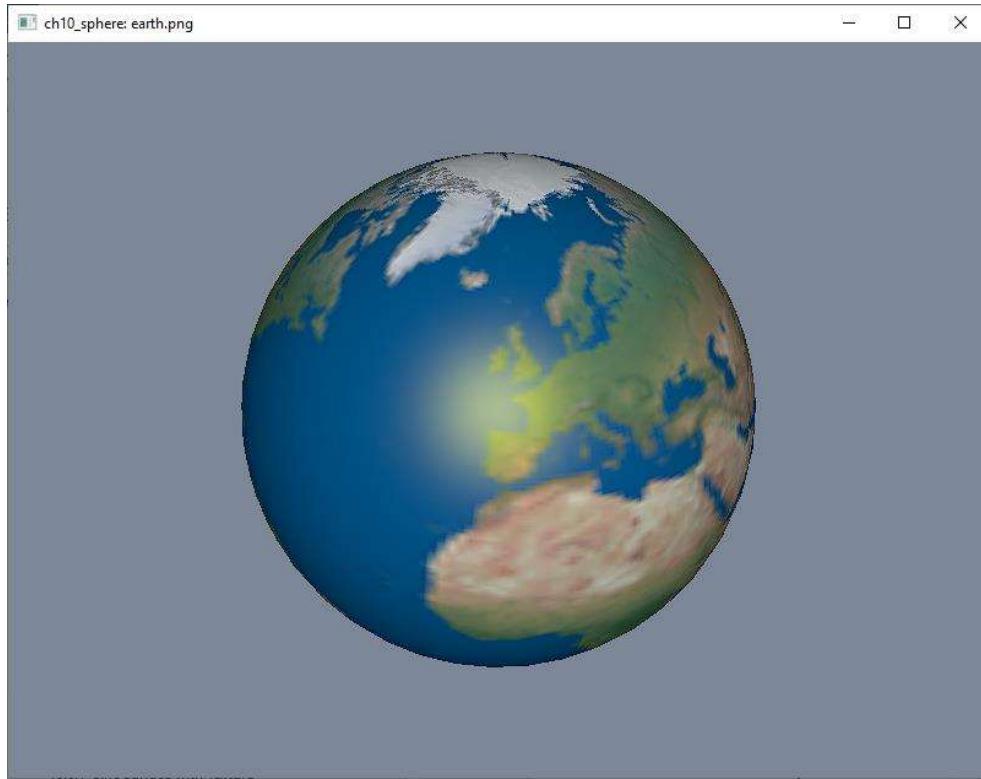


Fig. 10-4. Sphere with earth image texture.

### 10.5.3 Cylinder with Texture

Although, we have already implemented the `cylinder_data` method in the `vertex_data.rs` file, instead of adding the UV data to this method, we will add a new method called `cylinder_data_uv` to the `vertex_data.rs` file. This new method will allow us to change the texture coordinate range. Here is the code for this new method:

```
pub fn cylinder_data_uv(rin: f32, rout: f32, height: f32, n: usize, ul: f32, vl: f32) -> Vec<[f32; 2]> {
    let h = height / 2.0;
    let mut uvs: Vec<[f32; 2]> = Vec::with_capacity(24 * (n - 1));

    for i in 0..n - 1 {
        let theta = i as f32 * 360.0 / (n as f32 - 1.0);
        let theta1 = (i as f32 + 1.0) * 360.0 / (n as f32 - 1.0);
        let p0 = math_func::cylinder_position(rout, h, Deg(theta));
        let p1 = math_func::cylinder_position(rout, -h, Deg(theta));
        let p2 = math_func::cylinder_position(rin, -h, Deg(theta));
        let p3 = math_func::cylinder_position(rin, h, Deg(theta));
        let p4 = math_func::cylinder_position(rout, h, Deg(theta1));
        let p5 = math_func::cylinder_position(rout, -h, Deg(theta1));
        let p6 = math_func::cylinder_position(rin, -h, Deg(theta1));
        let p7 = math_func::cylinder_position(rin, h, Deg(theta1));

        let u0 = ul * (0.5 + (p0[0]/rout).atan2(p0[2]/rout)/PI/2.0);
        let u1 = ul * (0.5 + (p1[0]/rout).atan2(p1[2]/rout)/PI/2.0);
        let u2 = ul * (0.5 + (p2[0]/rin).atan2(p2[2]/rin)/PI/2.0);
        let u3 = ul * (0.5 + (p3[0]/rin).atan2(p3[2]/rin)/PI/2.0);
        let u4 = ul * (0.5 + (p4[0]/rout).atan2(p4[2]/rout)/PI/2.0);
```

```

let u5 = ul*(0.5 + (p5[0]/rout).atan2(p5[2]/rout)/PI/2.0);
let u6 = ul*(0.5 + (p6[0]/rin).atan2(p6[2]/rin)/PI/2.0);
let u7 = ul*(0.5 + (p7[0]/rin).atan2(p7[2]/rin)/PI/2.0);

let vt = v1*(rout-rin)/height;
let u2i = u2*rin/rout;
let u3i = u3*rin/rout;
let u6i = u6*rin/rout;
let u7i = u7*rin/rout;

// top face
uvs.push([u0, vt]);
uvs.push([u4, vt]);
uvs.push([u7, 0.0]);
uvs.push([u7, 0.0]);
uvs.push([u3, 0.0]);
uvs.push([u0, vt]);

// bottom face
uvs.push([u1, vt]);
uvs.push([u2, 0.0]);
uvs.push([u6, 0.0]);
uvs.push([u6, 0.0]);
uvs.push([u5, vt]);
uvs.push([u1, vt]);

//outer face
uvs.push([u0, v1]);
uvs.push([u1, 0.0]);
uvs.push([u5, 0.0]);
uvs.push([u5, 0.0]);
uvs.push([u4, v1]);
uvs.push([u0, v1]);

// inner face
uvs.push([u2i, 0.0]);
uvs.push([u3i, v1]);
uvs.push([u7i, v1]);
uvs.push([u7i, v1]);
uvs.push([u6i, 0.0]);
uvs.push([u2i, 0.0]);
}

uvs
}
}

```

This code can be used to create the UV data for our cylinder. Note that for any point  $P(x, y, z)$  on a cylinder of radius  $r$ , the U coordinates in the range  $[0, ul]$  can be calculated using the following formula:

$$u = ul \left[ 0.5 + \frac{\arctan2(x/r, z/r)}{2\pi} \right]$$

Note that if a point is on the inner face of the cylinder,  $r = rin$ , while  $r = rout$  if the point is on the outer face.

As discussed previously, UV coordinates usually have a value range of  $[0, 1]$ , but they are still valid if their value goes beyond this range. If the value is larger than one, we can set the address mode to “*Repeat*” or “*MirrorRepeat*”, which will repeat or mirror-repeat the texture pattern. On the other hand, if we set  $ul < 1$  and  $vl < 1$ , mipmaps will be used to shrink the texture.

Now, add a new Rust file called `cylinder.rs` to the `examples/ch10/` folder and type the following code into it:

```
mod common;
#[path="../common/vertex_data.rs"]
mod vertex_data;

fn create_vertices(ul:f32, vl:f32) -> Vec<common::Vertex> {
    let(pos, normal, _uv) = vertex_data::cylinder_data(0.8, 1.5, 2.0, 50);
    let uv = vertex_data::cylinder_data_uv(0.8, 1.5, 2.0, 50, ul, vl);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "brick.png";
    let mut ul:f32 = 1.0;
    let mut vl:f32 = 1.0;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        ul = args[2].parse().unwrap();
    }
    if args.len() > 3 {
        vl = args[3].parse().unwrap();
    }

    let vertex_data = create_vertices(ul, vl);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::Repeat;
    let v_mode = wgpu::AddressMode::Repeat;

    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "cylinder");
}
```

Note that the `create_vertices` method accepts `ul` and `vl` as its input arguments, which set the value range of the texture coordinates in the U and V directions respectively. Within this method, we get the vertex position and normal vector data by calling the `cylinder_data` function, and obtain the UV data by calling the `cylinder_data_uv` function.

Inside the `main` function, we set `AddressMode` to “`Repeat`” for both `u_mode` and `v_mode`. In addition to the image file name, the user can also specify the `ul` and `vl` values from the terminal window.

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch10_cylinder"
path = "examples/ch10/cylinder.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch10_cylinder
```

This produces the results shown in Fig.10-5.

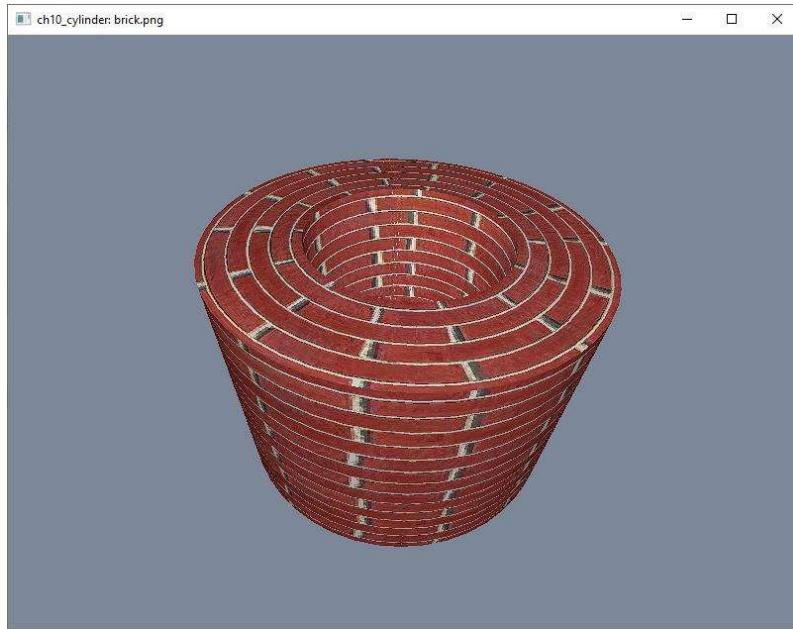


Fig.10-5. Textured cylinder created with default parameters.

You can see from the figure that the texture on the cylinder is stretched too much along the  $u$  direction when the default parameters  $ul = 1$  and  $vl = 1$  are used, resulting in distorted bricks that do not look like real bricks. Now, set  $ul = 2$  and  $vl = 0.5$  by running the following command:

```
cargo run --example ch10_cylinder "brick.png" "2.0" "0.5"
```

This produces the results shown in Fig.10-6. You can see that the texture looks much better than it did under the default parameters.

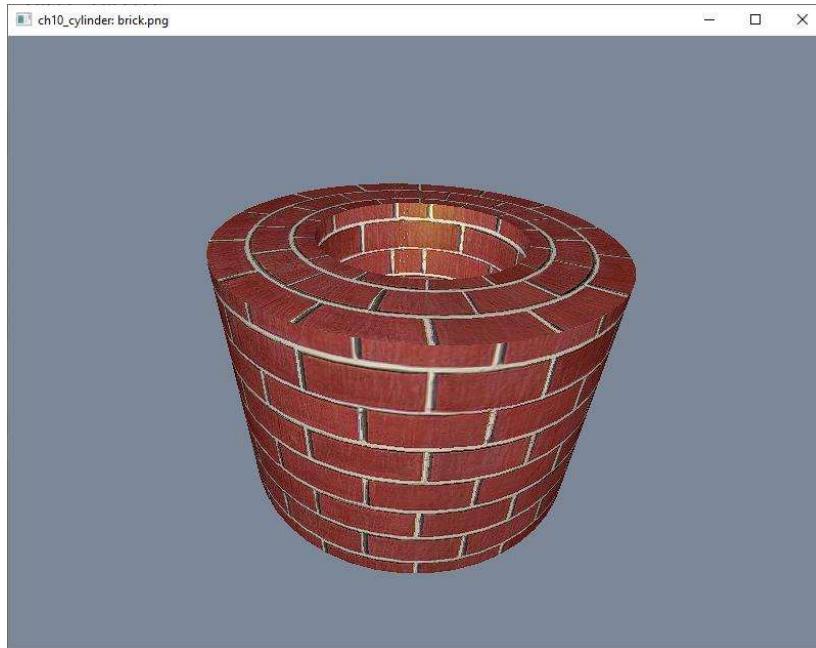


Fig.10-6. Textured cylinder created with  $ul = 2$  and  $vl = 0.5$ .

## 10.6 Simple 3D Surfaces

Previously, we implemented the `simple_surface_data` method in the `surface_data.rs` file. Now, we need to modify this method to create the UV (texture coordinates) data for simple 3D surfaces.

First, we need to make changes to the `create_quad` method in the `surface_data.rs` file, using the following highlighted code:

```
fn create_quad(p0:[f32;3], p1:[f32;3], p2:[f32;3], p3:[f32;3], ymin:f32, ymax:f32, colormap_name: &str,
scale: f32) ->
(Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 3]>, Vec<[f32; 2]>) {
    // position
    let mut position:Vec<[f32; 3]> = Vec::with_capacity(6);
    position.push(p0);
    position.push(p1);
    position.push(p2);
    position.push(p2);
    position.push(p3);
    position.push(p0);

    // normal
    let ca = Vector3::new(p2[0]-p0[0], p2[1]-p0[1], p2[2]-p0[2]);
    let db = Vector3::new(p3[0]-p1[0], p3[1]-p1[1], p3[2]-p1[2]);
    let cp = (ca.cross(db)).normalize();
    let mut normal:Vec<[f32;3]> = Vec::with_capacity(6);
    normal.push([cp[0], cp[1], cp[2]]);
    normal.push([cp[0], cp[1], cp[2]]);

    // color
    let c0 = colormap::color_lerp(colormap_name, ymin, ymax, p0[1]);
    let c1 = colormap::color_lerp(colormap_name, ymin, ymax, p1[1]);
    let c2 = colormap::color_lerp(colormap_name, ymin, ymax, p2[1]);
    let c3 = colormap::color_lerp(colormap_name, ymin, ymax, p3[1]);
    let mut color:Vec<[f32;3]> = Vec::with_capacity(6);
    color.push(c0);
    color.push(c1);
    color.push(c2);
    color.push(c2);
    color.push(c3);
    color.push(c0);

    // uv
    let mut uv:Vec<[f32;2]> = Vec::with_capacity(6);
    let uv0 = normalize_uv(p0, scale);
    let uv1 = normalize_uv(p1, scale);
    let uv2 = normalize_uv(p2, scale);
    let uv3 = normalize_uv(p3, scale);
    uv.push(uv0);
    uv.push(uv1);
    uv.push(uv2);
    uv.push(uv2);
    uv.push(uv3);
    uv.push(uv0);
```

```
(position, normal, color, uv)
}
```

Note that the input points were originally normalized to the region  $[-1, 1]$  for all components. In order to compute the texture coordinates, we need to renormalize the  $x$ , and  $z$  components into the range of  $[0, 1]$  by calling the *normalize\_uv* method:

```
fn normalize_uv(pt:[f32;3], scale:f32) -> [f32; 2] {
    let px = (1.0 + pt[0]/scale)/2.0;
    let pz = (1.0 + pt[2]/scale)/2.0;
    [px, pz]
}
```

Now, we can create the UV data in the *simple\_surface\_data* method with the following highlighted code snippet:

```
pub fn simple_surface_data(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, xmin:f32, xmax:f32,
zmin:f32, zmax:f32,
nx:usize, nz: usize, scale:f32, scaley:f32) -> (Vec<[f32;3]>, Vec<[f32;3]>,
Vec<[f32;3]>,Vec<[f32;2]>,Vec<[f32;2]>) {

    let dx = (xmax-xmin)/(nx as f32-1.0);
    let dz = (zmax-zmin)/(nz as f32-1.0);
    let mut ymin1: f32 = 0.0;
    let mut ymax1: f32 = 0.0;

    let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nz]; nx];
    for i in 0..nx {
        let x = xmin + i as f32 * dx;
        let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nz);
        for j in 0..nz {
            let z = zmin + j as f32 * dz;
            let pt = f(x, z);
            pt1.push(pt);
            ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
            ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
        }
        pts[i] = pt1;
    }

    let ymin = ymin1 - scaley * (ymax1 - ymin1);
    let ymax = ymax1 + scaley * (ymax1 - ymin1);

    for i in 0..nx {
        for j in 0..nz {
            pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
        }
    }

    let cmin = normalize_point([0.0, ymin1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];
    let cmax = normalize_point([0.0, ymax1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];

    let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
    let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
    let mut colors: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
    let mut uvs: Vec<[f32; 2]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
    let uv1: Vec<[f32; 2]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);

    for i in 0..nx - 1 {
        for j in 0.. nz - 1 {
```

```

let p0 = pts[i][j];
let p1 = pts[i][j+1];
let p2 = pts[i+1][j+1];
let p3 = pts[i+1][j];

let ( mut pos, mut norm, mut col, mut uv) = create_quad(p0, p1, p2, p3, cmin, cmax,
    colormap_name, scale);

// positions
positions.append(&mut pos);

// normals
normals.append(&mut norm);

// colors
colors.append(&mut col);

// uvs
uvs.append(&mut uv);
}
}

(positions, normals, colors, uvs, uv1)
}
}

```

In the following sections, I will demonstrate how to use this method to create two simple surfaces, *sinc* and *peaks*, with textures.

### 10.6.1 Sinc Surface with Texture

Add a new Rust file called *sinc.rs* to the *examples/ch10/* folder and type the following code into it:

```

mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices() -> Vec<common::Vertex> {
    let (pos, normal, _color, uv, _uv1) =
        surface::simple_surface_data(&math::sinc, "jet", -8.0, 8.0, -8.0, 8.0, 40, 40, 1.5, 0.3);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "brick.png";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }

    let vertex_data = create_vertices();
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;
}

```

## 244 | Practical GPU Graphics with wgpu and Rust

```
    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "sinc");  
}
```

This code first implements a `create_vertices` method which calls the `surface::simple_surface_data` function to generate the vertex position, normal vector, and UV data, and then converts that data into a proper format to be used in creating our textured sinc surface.

Inside the main function, we set a user input field that lets you specify different image files.

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]  
name = "ch10_sinc"  
path = "examples/ch10/sinc.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch10_sinc
```

This produces the results shown in Fig.10-7.

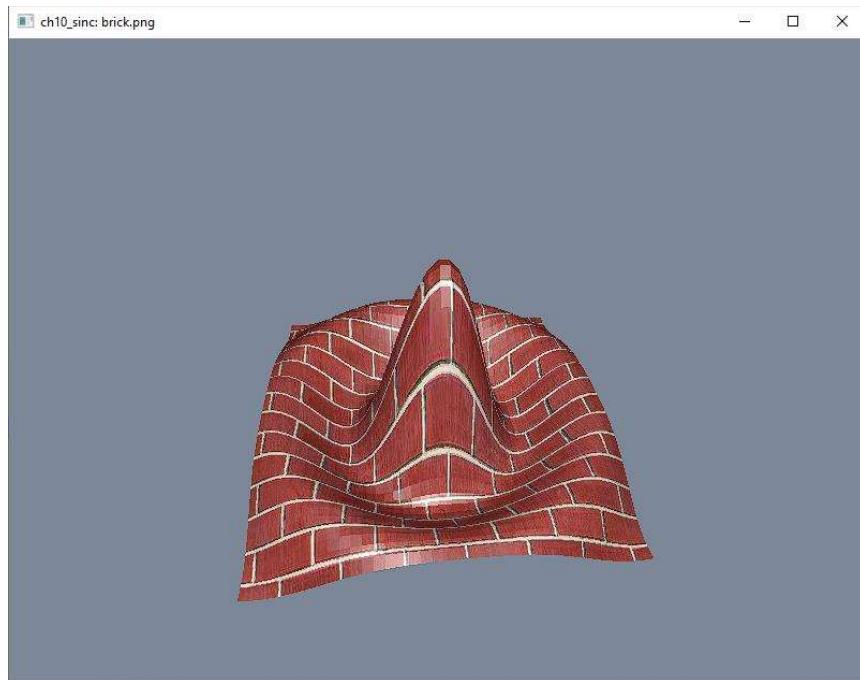


Fig.10-6. Sinc surface with brick texture.

You can change the texture image by running the following command:

```
cargo run --example ch10_sinc "jiuzhaigou.png"
```

This produces the results shown in Fig.10-8.

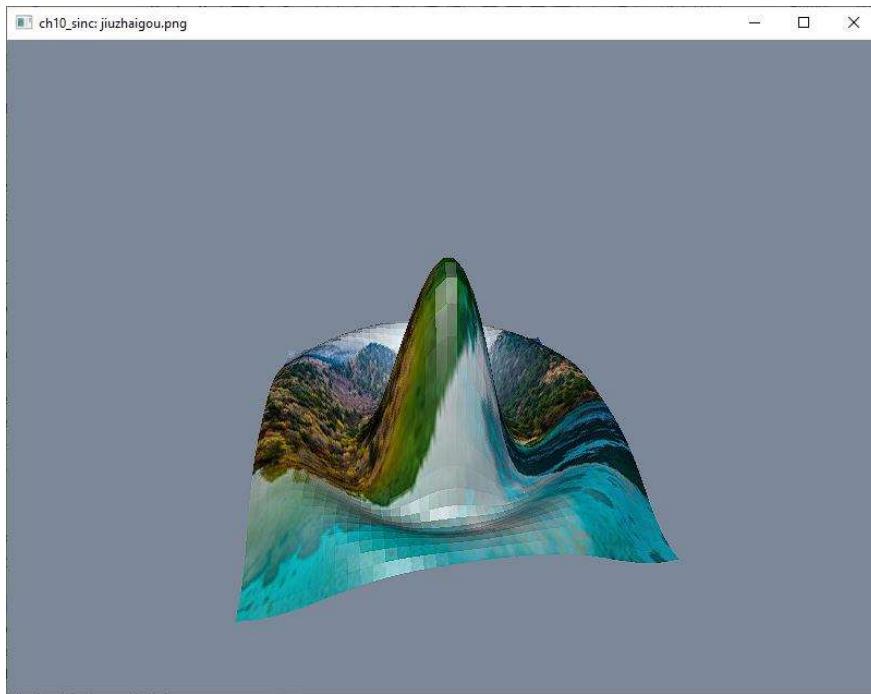


Fig.10-8. Sinc surface with Jiu Zhai Gou (China) texture.

### 10.6.2 Peaks Surface with Texture

Add a new Rust file called `peak.rs` to the `examples/ch10/` folder with the following code:

```
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices() -> Vec<common::Vertex> {
    let (pos, normal, _color, uv, _uv1) =
        surface::simple_surface_data(&math::peaks, "jet", -3.0, 3.0, -3.0, 3.0, 51, 51, 1.5, 0.0);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "jiuzhaigou.png";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }

    let vertex_data = create_vertices();
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;
```

```
common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "peaks");
}
```

This code first implements a `create_vertices` method that calls the `surface::simple_surface_data` function to generate the vertex position, normal vector, and UV data and then converts that data into a proper format to be used in creating our textured peaks surface.

Inside the `main` function, we set a user input field that lets you specify different image files.

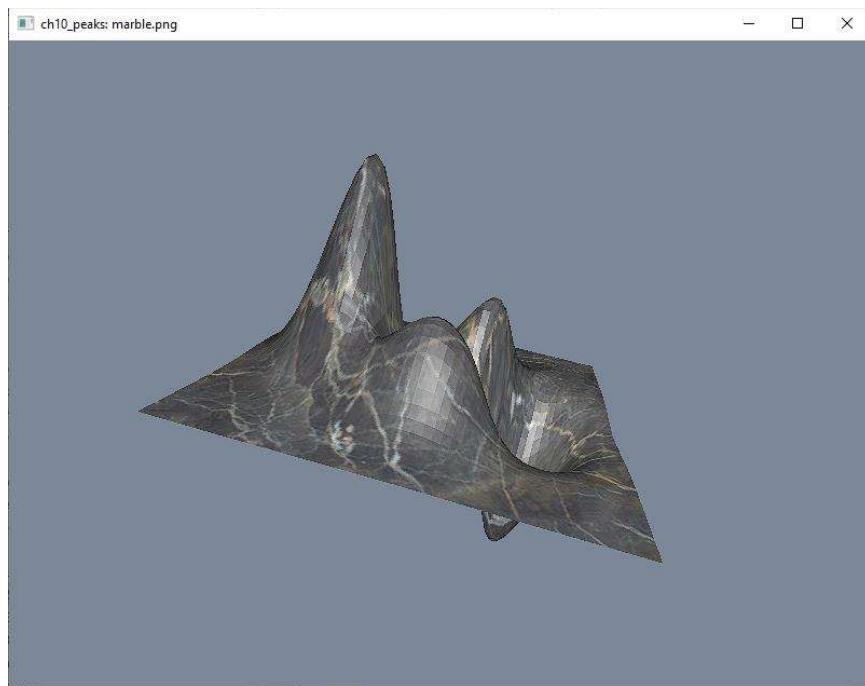
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch10_peaks"
path = "examples/ch10/peaks.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch10_peaks "marble"
```

This produces the results shown in Fig.10-9.



*Fig.10-9. Peaks surface with marble texture.*

## 10.7 Parametric 3D Surfaces

Previously, we implemented the `parametric_surface_data` method in the `surface_data.rs` file. However, instead of modifying this method to create UV data for parametric 3D surfaces, here, we will add a new method called `parameteric_surface_data_uv` to the `surface_data.rs` file. This new method will allow you to change the value range of the texture coordinates.

To do this, we first need to add a new private function called `create_quad_uv` to the `surface_data.rs` file with the following code:

```
fn create_quad_uv(p0:[f32;2], p1:[f32;2], p2:[f32;2], p3:[f32;2]) -> Vec<[f32;2]> {
    let mut uv:Vec<[f32;2]> = Vec::with_capacity(6);
    uv.push(p0);
    uv.push(p1);
    uv.push(p2);
    uv.push(p2);
    uv.push(p3);
    uv.push(p0);
    uv
}
```

This method is very simple – it just uses four  $[f32; 2]$  points to create UV coordinates for two triangles. We also need to add another method called *normalize\_uv\_parametric* to the *surface\_data.rs* file with the following code:

```
fn normalize_uv_parametric(u:f32, v:f32, umin:f32, umax:f32, vmin:f32, vmax:f32, u1:f32, v1:f32) -> [f32;2] {
    let uu = u1*(u-umin)/(umax-umin);
    let vv = v1*(v-vmin)/(vmax-vmin);
    [uu, vv]
}
```

Where  $u$  and  $v$  are the parametric variables, while  $uu$  and  $vv$  are the texture coordinates. Please do not confuse the parametric variables with the texture coordinates.

We can then add our *parametric\_surface\_data\_uv* method to the *surface\_data.rs* file with the following code:

```
pub fn parametric_surface_data_uv(umin:f32, umax:f32, vmin:f32, vmax:f32, nu:usize, nv: usize,
u1:f32, v1:f32) -> Vec<[f32;2]> {
    let du = (umax-umin)/(nu as f32-1.0);
    let dv = (vmax-vmin)/(nv as f32-1.0);
    let mut pts:Vec<Vec<[f32; 2]>> = vec![vec![Default::default(); nv]; nu];
    for i in 0..nu {
        let u = umin + i as f32 * du;
        let mut pt1:Vec<[f32; 2]> = Vec::with_capacity(nv);
        for j in 0..nv {
            let v = vmin + j as f32 * dv;
            let pt = normalize_uv_parametric(u, v, umin, umax, vmin, vmax, u1, v1);
            pt1.push(pt);
        }
        pts[i] = pt1;
    }

    let mut uvs: Vec<[f32; 2]> = Vec::with_capacity((4* (nu - 1)*(nv -1)) as usize);

    for i in 0..nu - 1 {
        for j in 0.. nv - 1 {
            let p0 = pts[i][j];
            let p1 = pts[i+1][j];
            let p2 = pts[i+1][j+1];
            let p3 = pts[i][j+1];

            let mut uv = create_quad_uv(p0, p1, p2, p3);
            uvs.append(&mut uv);
        }
    }
}
```

Here, we use the `pts` variable to store the normalized  $u$  and  $v$  parametric variables. In fact, the normalized  $u$  and  $v$  parametric variables are exactly the same as the texture coordinates. The reason is that only in parametric space does there exist a constant 2D grid, so we can only compute the UV data or texture coordinates in parametric space.

### 10.7.1 Klein Bottle with Texture

Add a new Rust file called `klein_bottle.rs` to the `examples/ch10/` folder and type the following code into it:

```
use std::f32::consts::PI;
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(ul:f32, vl:f32) -> Vec<common::Vertex> {
    let (pos, normal, _color, _uv, _uv1) = surface::parametric_surface_data(&math::klein_bottle, "jet",
        0.0, PI, 0.0, 2.0*PI, 70, 30, -2.0, 2.0, -2.0, 3.0, 1.5, 0.0);
    let uv = surface::parametric_surface_data_uv(0.0, PI, 0.0, 2.0*PI, 70, 30, ul, vl);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "brick.png";
    let mut ul:f32 = 1.0;
    let mut vl:f32 = 1.0;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        ul = args[2].parse().unwrap();
    }
    if args.len() > 3 {
        vl = args[3].parse().unwrap();
    }

    let vertex_data = create_vertices(ul, vl);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::Repeat;
    let v_mode = wgpu::AddressMode::Repeat;

    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "klein_bottle");
}
```

This code first implements a `create_vertices` method that calls the `surface::parametric_surface_data` function to generate the vertex position and normal vector data, and then calls the `surface::parametric_surface_data_uv` function to produce the UV data. It then converts these data into a proper format to be used in creating our textured Klein bottle.

Inside the `main` function, we set a user input field that lets you specify different image files, along with input fields for specifying  $ul$  and  $vl$ .

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch10_klein_bottle"
path = "examples/ch10/klein_bottle.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

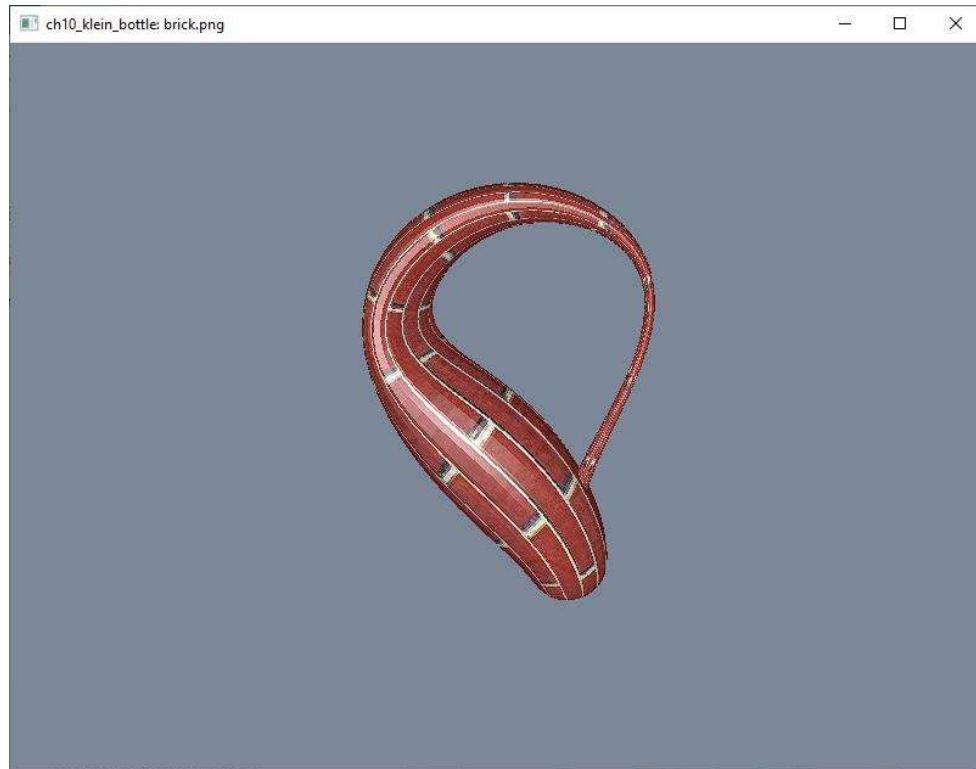
```
cargo run --example ch10_klein_bottle
```

This produces the results shown in Fig.10-10.

To create a better-looking textured Klein bottle, we can change the value ranges of texture coordinates by running the following command:

```
cargo run --example ch10_klein_bottle "brick.png" "4" "1"
```

This generates the results shown in Fig.10-11.



*Fig.10-8. Textured Klein bottle created with default parameters.*

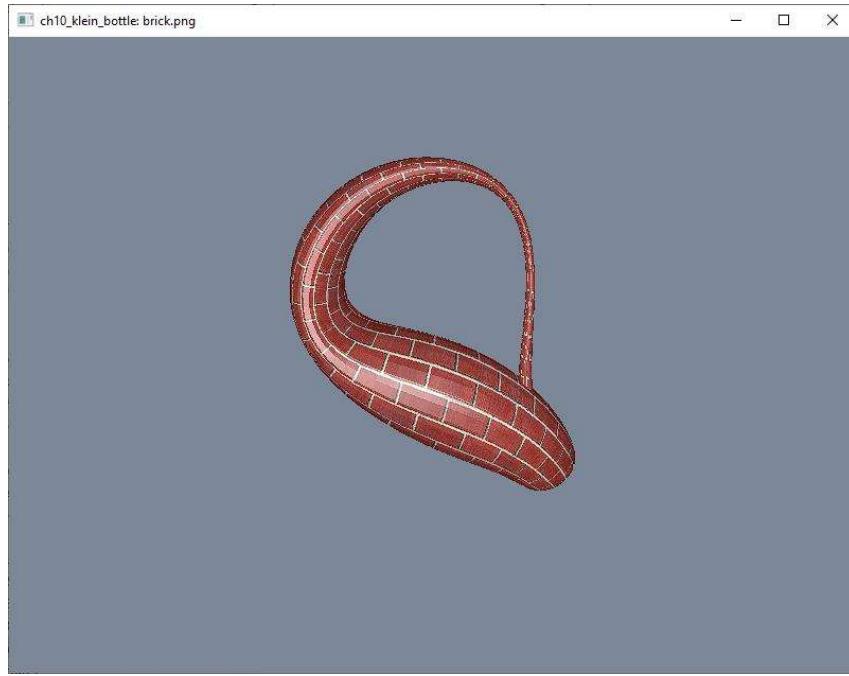


Fig. 10-11. Textured Klein bottle with parameters  $ul = 4$  and  $vl = 1$ .

### 10.7.2 Wellenkugel Surface with Texture

Add a new Rust file called `wellenkugel.rs` to the `examples/ch10/` folder and enter the following content into it:

```
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(ul:f32, vl:f32) -> Vec<common::Vertex> {
    let (pos, normal, _color, _uv, _uv1) = surface::parametric_surface_data(&math::wollenkugel, "jet",
        0.0, 14.5, 0.0, 5.0, 100, 50, -10.0, 10.0, -10.0, 10.0, 1.2, 0.0);
    let uv = surface::parametric_surface_data_uv(0.0, 14.5, 0.0, 5.0, 100, 50, ul, vl);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "brick.png";
    let mut ul:f32 = 1.0;
    let mut vl:f32 = 1.0;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        ul = args[2].parse().unwrap();
    }
}
```

```

}
if args.len() > 3 {
    v1 = args[3].parse().unwrap();
}

let vertex_data = create_vertices(u1, v1);
let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
let u_mode = wgpu::AddressMode::Repeat;
let v_mode = wgpu::AddressMode::Repeat;

common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "wellenkugel");
}

```

This code first implements a *create\_vertices* method that calls the *surface::parametric\_surface\_data* function to generate the vertex position and normal vector data, and then calls the *surface::parametric\_surface\_data\_uv* function to produce the UV data. It then converts these data into a proper format to be used in creating our textured Wellenkugel surface.

Inside the *main* function, we set user input fields that let you specify different image files and different values for *u1* and *v1*.

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch10_wellenkugel"
path = "examples/ch10/wellenkugel.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch10_wellenkugel "brick.png" "4" "1"
```

This produces the results shown in Fig.10-12.

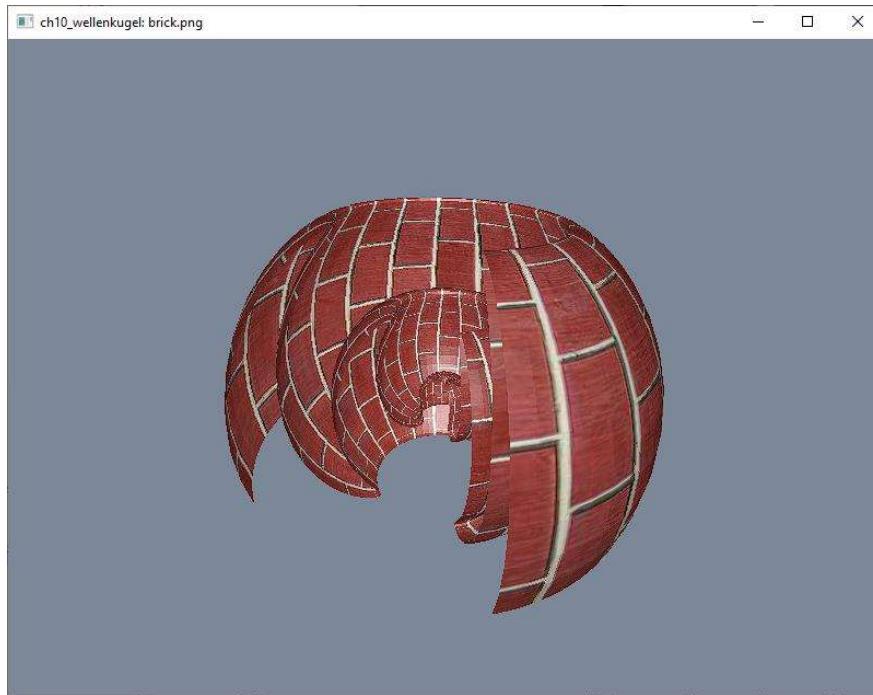


Fig.10-12. Wellenkugel surface with texture.

## 10.8 Multiple Textures

So far, we have only looked at mapping a single image texture onto a 3D object. However, there are many surface rendering effects that can be implemented using more than one texture image. For example, we may want to apply a different texture to each face of a cube, instead of applying the same texture to all six faces.

There are many ways to apply multiple textures. One way is to change our texture variable into an array and adjust the load texture function to handle multiple textures. We would then need to define a sampler uniform array and pass it to our fragment shader in order to process multiple textures to produce the final color. This approach involves significant code changes and careful treatment of texture coordinates.

In this section, I will show you a shortcut approach: first combine multiple images into a single big image, and then apply different portions of the image to different faces of the cube by properly assigning texture coordinates.

### 10.8.1 Texture Coordinates

I have combined six images into a single  $384 \times 256$  image, which is stored in the `examples/ch10/assets/` folder under the filename `multiple.png`, as shown in Fig.10-13.



*Fig. 10-13. A single image file that contains six small images.*

Now, we can divide the texture coordinates into six parts and associate each part with a face of the cube, as shown in Fig.10-14.

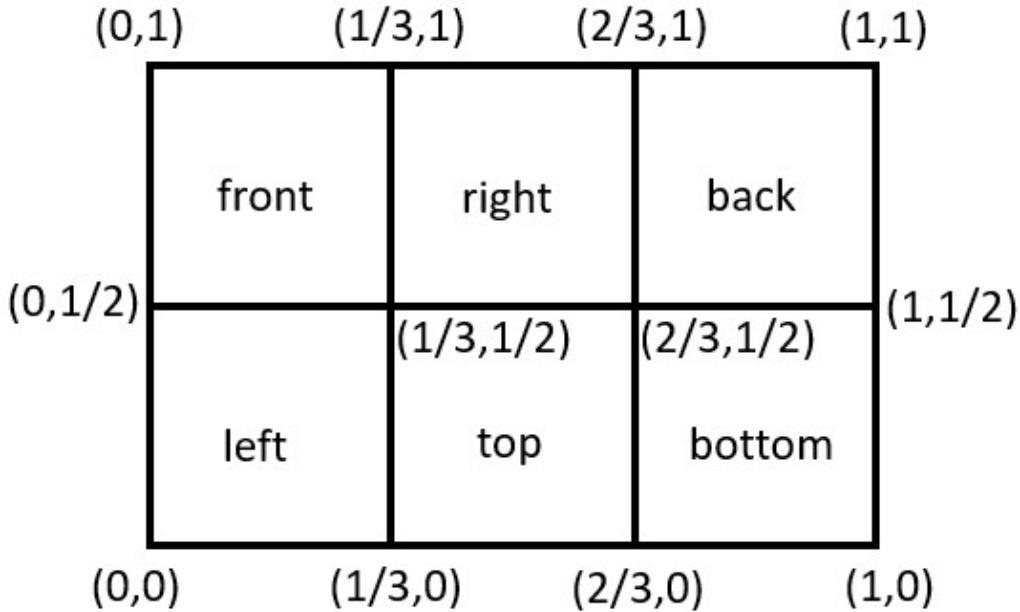


Fig.10-14. Texture coordinates assignment.

Here, we randomly assigned texture coordinates to each face of the cube, meaning you can change it to whatever you would like. After creating the UV data according to Fig.10-14, we can then use the same code for mapping a single texture in preceding examples to map different images onto different faces of the cube.

Previously, we implemented a `cube_data` method in the `vertex_data.rs` file. Now, we will add a new method called `cube_data_uv1` to the `vertex_data.rs` file. This new method will incorporate new UV data according to Fig.10-14 with the following code:

```
pub fn cube_data_uv1() -> Vec<[f32; 2]> {
    [
        //front
        [0.0, 1.0/2.0], [1.0/3.0, 1.0/2.0], [0.0, 1.0], [0.0, 1.0], [1.0/3.0, 1.0/2.0], [1.0/3.0, 1.0],
        //right
        [1.0/3.0, 1.0/2.0], [2.0/3.0, 1.0/2.0], [1.0/3.0, 1.0], [1.0/3.0, 1.0], [2.0/3.0, 1.0/2.0],
        [2.0/3.0, 1.0],
        //back
        [2.0/3.0, 1.0/2.0], [1.0, 1.0/2.0], [2.0/3.0, 1.0], [2.0/3.0, 1.0], [1.0, 1.0/2.0], [1.0, 1.0],
        //left
        [0.0, 0.0], [0.0, 1.0/2.0], [1.0/3.0, 0.0], [1.0/3.0, 0.0], [0.0, 1.0/2.0], [1.0/3.0, 1.0/2.0],
        //top
        [1.0/3.0, 0.0], [2.0/3.0, 0.0], [1.0/3.0, 1.0/2.0], [1.0/3.0, 1.0/2.0], [2.0/3.0, 0.0],
        [2.0/3.0, 1.0/2.0],
        //bottom
        [2.0/3.0, 1.0/2.0], [1.0, 1.0/2.0], [2.0/3.0, 0.0], [2.0/3.0, 0.0], [1.0, 1.0/2.0], [1.0, 0.0],
    ].to_vec()
}
```

This method returns a  $\text{Vec} < [f32; 2] >$  vector that represents our new UV data.

## 10.8.2 Rust Code

Add a new Rust file called *cube\_multiple.rs* to the *examples/ch10/* folder and type the following code into it:

```
mod common;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

fn vertex(p:[i8; 3], n: [i8; 3], t:[f32; 2]) -> common::Vertex {
    common::Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        normal: [n[0] as f32, n[1] as f32, n[2] as f32, 1.0],
        uv: [t[0] as f32, t[1] as f32],
    }
}

fn create_vertices() -> Vec<common::Vertex> {
    let (pos, _col, _uv, normal) = vertex_data::cube_data();
    let uv = vertex_data::cube_data_uv1();
    let mut data: Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}

fn main(){
    let file_name = "multiple.png";

    let vertex_data = create_vertices();
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, u_mode, v_mode, "cube_multiple");
}
```

This code first creates a *vertex* function that converts the position and normal vector data from the *i8* type into *f32* type. It then implements a *create\_vertices* method that calls the *surface::cube\_data* function to generate the vertex position and normal vector data, and then calls the *surface::cube\_data\_uv1* function to produce the UV data. The *create\_vertices* method also converts these data into the proper format to be used in creating our cube with multiple textures.

Inside the *main* function, we specify the image file as “*multiple.png*” and leave out the user input field.

## 10.8.3 Run Application

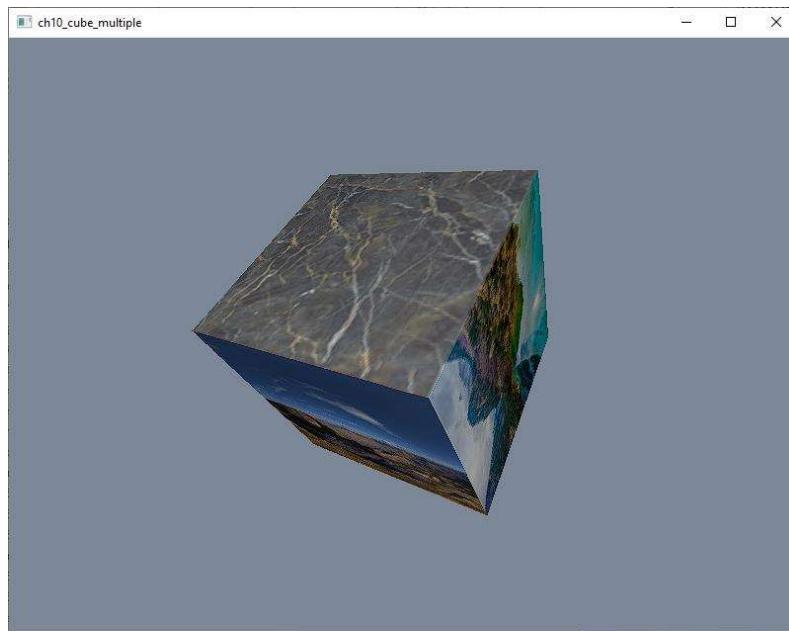
Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch10_cube_multiple"
path = "examples/ch10/cube_multiple.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch10_cube_multiple
```

This produces the results shown in Fig.10-15.



*Fig.10-15. Textured cube: different faces with different image textures.*

In this chapter, we only discussed texture mapping using 2D images. The 3D objects we created with this texture-mapping look very nice. However, their surfaces are still overly smooth. There are several ways to improve our current texture mapping to make our objects look more realistic, including bump mapping, normal mapping, and displacement mapping.

Bump mapping creates the illusion of depth and texture on the surface of a 3D object by perturbing the surface normal vectors of the object and using these perturbed normal to calculate lighting. The result is a surface that appears bumpy rather than smooth, even though the surface of the underlying object is not changed.

Normal mapping is a more modern type of bump mapping. Thus, it also fakes the appearance of bumps, dents, and other details without adding actual resolution to objects. It is more sophisticated than basic bump mapping because it stores  $x$ ,  $y$ , and  $z$  information for the surface normal at every single texel. A common use of this technique is to generate a normal map from a high-resolution mesh model or height map and apply it to a low-resolution mesh model to improve its level of detail.

Displacement mapping is an alternative texture mapping technique that, instead of simply perturbing the surface normals, uses a texture or height map to displace the actual geometric position of each point across surface. This results in a surface with such highly complex depth that parts of it can cast shadows on or occlude other parts of itself. Since it adds a large amount of geometry to objects, this technique is the costliest and thus uncommon in real-time applications.

Here, I will not discuss these advanced texture-mapping techniques because they are beyond the scope of this book.



# 11 3D Surface Charts

Surface charts are plots of 3D data. Rather than displaying the individual data points, surface charts show a functional relationship between a dependent variable  $y$  and two independent variables  $x$  and  $z$ . In preceding chapters, we have created various 3D surfaces, including wireframe plots, surfaces with colormap, and surfaces with lighting and texture.

A real-word 3D surface chart should include a surface with a colormap. In most cases, the surface should also have a wireframe or mesh lines on it. Our task in this chapter is to build a 3D surface chart with both a colormap and wireframe, as shown in Fig.11-1.

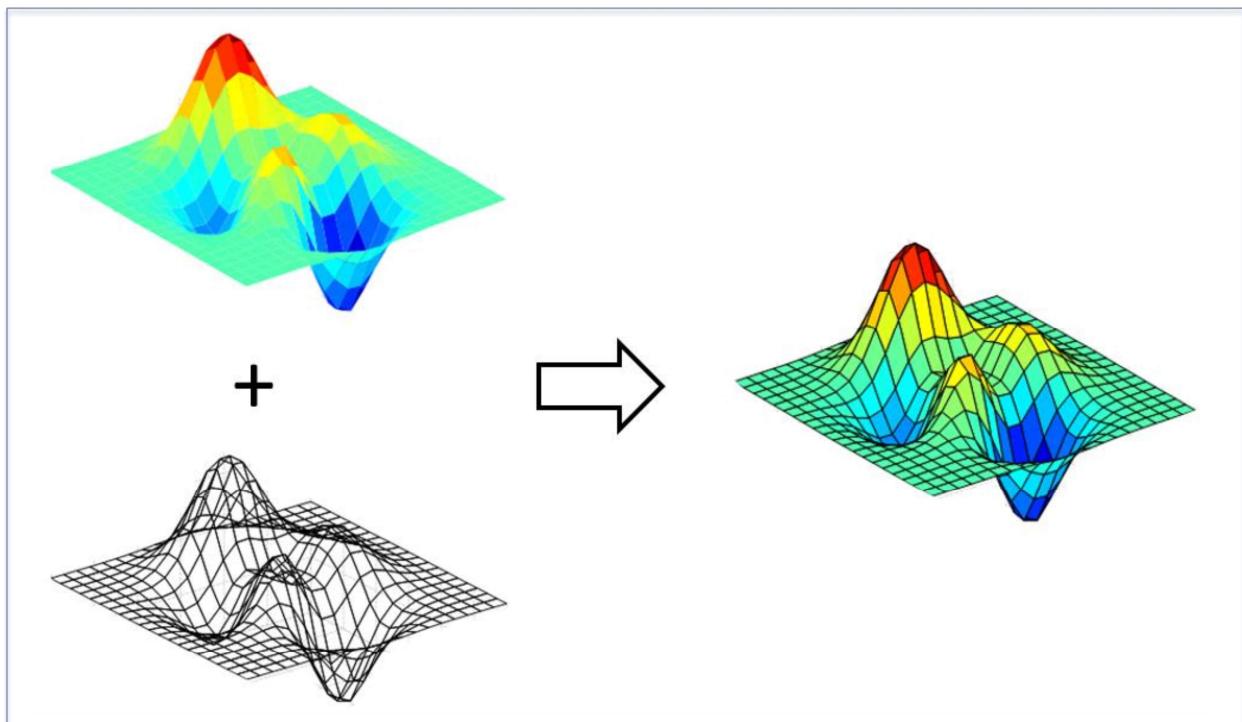


Fig.11-1. Surface with colormap + wireframe = 3D surface chart.

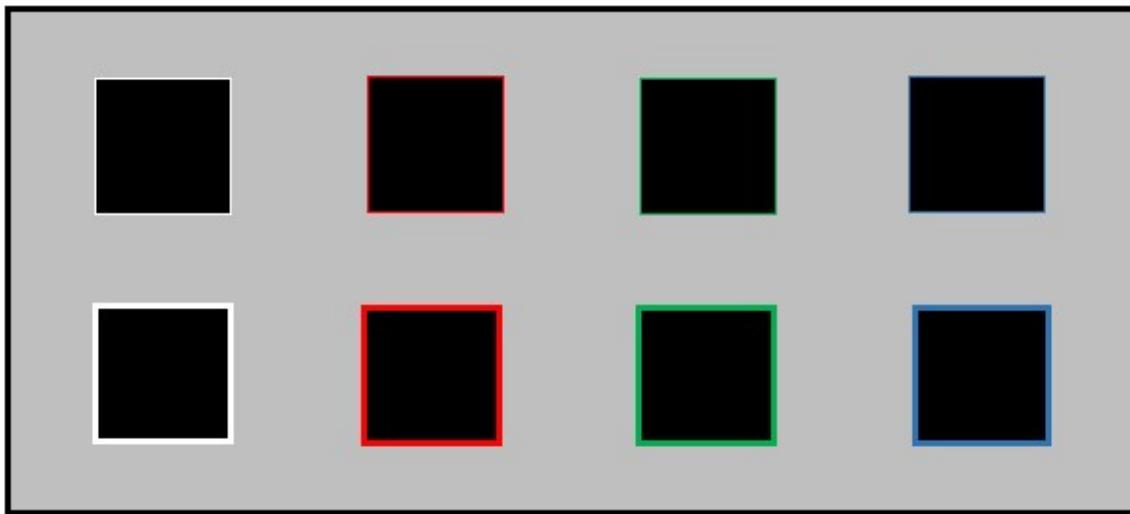
## 11.1 Wireframe as Texture

There are two ways to create a 3D surface chart in *wgpu*. One way is to build the surface chart directly: use the *triangle-list* primitive to create a surface with a colormap, then use the *line-list* primitive to create a surface wireframe, and finally combine them together to build the 3D surface chart. This approach should work, but it will be complicated because it requires two separate rendering pipelines: one for the *triangle-list* primitive and another for the *line-list* primitive. We will discuss this approach in the following chapter.

In this chapter, we will use an alternative approach where we do not attempt to build a complicated model, but instead build a surface with a colormap, as we did previously in Chapter 9, and then add the wireframe as part of the rendering process. This way, we only need to use a single rendering pipeline that creates a surface with a colormap, while the wireframe will be added to the surface as a texture.

### 11.1.1 Square Textures

Fig.11-2 shows eight different squares of four colors – white, red, green and blue. There are two squares for each color, one with a line thickness of 1 pixel and the other with a line thickness of 2 pixels.



*Fig.11-2. Squares as a wireframe texture.*

Note that the size of each square in the figure is  $32 \times 32$  pixels, and all of the squares are filled with black. Previously, we created simple or parametric 3D surfaces using quad meshes, where each quad mesh consists of four vertices and two triangles. Here, our plan is to use one of the black squares shown in Fig.11-2 as an image texture and map it onto a quad mesh. We will build a 3D surface out of many quad meshes, and each quad mesh will be mapped with a square texture; combined all of the square textures will form a wireframe for the surface. The colormap of the surface will be visible through the black fill of the square textures because black has zero for its R, G, and B components. This means that if you add any color to black, the result will be that same color:

$$(r, g, b) \text{ color} + (0.0, 0.0, 0.0) \text{ black color} = (r, g, b) \text{ color}$$

You can see that the black-filled square has a similar feature as a transparent square. Thus, the resulting surface will be a nice 3D surface chart with both a colormap and wireframe

### 11.1.2 Texture Coordinates

Based on our previous analysis, we can easily create UV data for the texture coordinates of simple and parametric 3D surfaces.

As shown in Fig.11-3, the unit cell for a simple or parametric 3D surface can be represented using a quad mesh with four vertices and two triangles.

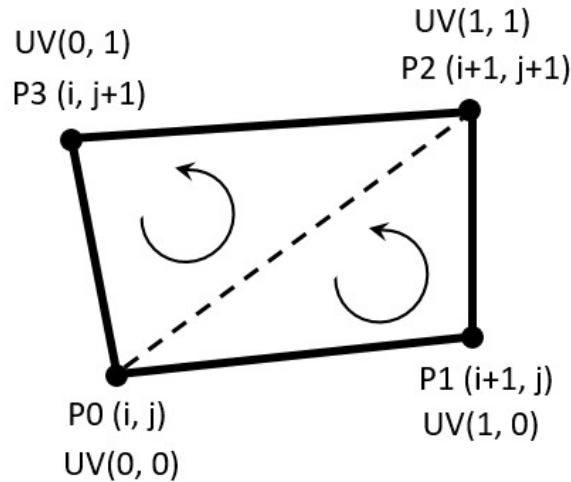


Fig.11-3. Unit cell for a 3D surface.

For any unit cell or quad mesh, the UV data will be identical and can be expressed in the following form:

```
uv_data = (0,0, 0,1, 1,1, 0,1) for quad(p0, p1, p2, p3);
```

## 11.2 Shaders for 3D Charts

In this section, we will implement our shader, which will be even more complicated than our previous shader program because now we want to incorporate three elements: textures, colormaps, and lighting.

Add a new sub-folder called *ch11* to the *examples/* folder, and then add a new *shader.wgsl* file to this newly created folder with the following code:

```
// vertex shader

[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
    normal_mat : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Input {
```

## 260 | Practical GPU Graphics with wgpu and Rust

```
[[location(0)]] pos : vec4<f32>;
[[location(1)]] normal : vec4<f32>;
[[location(2)]] uv : vec2<f32>;
[[location(3)]] color: vec4<f32>;
};

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_position : vec4<f32>;
    [[location(1)]] v_normal : vec4<f32>;
    [[location(2)]] v_uv : vec2<f32>;
    [[location(3)]] v_color: vec4<f32>;
};

[[stage(vertex)]]
fn vs_main(in: Input) -> Output {
    var output: Output;
    let m_position:vec4<f32> = uniforms.model_mat * in.pos;
    output.v_position = m_position;
    output.v_normal = uniforms.normal_mat * in.normal;
    output.v_uv = in.uv;
    output.position = uniforms.view_project_mat * m_position;
    output.v_color = in.color;
    return output;
}

// fragment shader

[[block]] struct Uniforms {
    light_position : vec4<f32>;
    eye_position : vec4<f32>;
};
[[binding(1), group(0)]] var<uniform> frag_uniforms : Uniforms;

[[block]] struct Uniforms {
    specular_color : vec4<f32>;
    ambient_intensity: f32;
    diffuse_intensity :f32;
    specular_intensity: f32;
    specular_shininess: f32;
    is_two_side: i32;
};
[[binding(2), group(0)]] var<uniform> light_uniforms : Uniforms;

[[binding(0), group(1)]] var texture_data : texture_2d<f32>;
[[binding(1), group(1)]] var texture_sampler : sampler;

[[stage(fragment)]]
fn fs_main(in:Output) -> [[location(0)]] vec4<f32> {
    let texture_color:vec4<f32> = textureSample(texture_data, texture_sampler, in.v_uv);

    let N:vec3<f32> = normalize(in.v_normal.xyz);
    let L:vec3<f32> = normalize(frag_uniforms.light_position.xyz - in.v_position.xyz);
    let V:vec3<f32> = normalize(frag_uniforms.eye_position.xyz - in.v_position.xyz);
    let H:vec3<f32> = normalize(L + V);

    // front side
    var diffuse:f32 = light_uniforms.diffuse_intensity * max(dot(N, L), 0.0);
    var specular: f32 = light_uniforms.specular_intensity *
        pow(max(dot(N, H),0.0), light_uniforms.specular_shininess);
```

```
// back side
var is_two_side:i32 = light_uniforms.is_two_side;
if(is_two_side == 1) {
    diffuse = diffuse + light_uniforms.diffuse_intensity * max(dot(-N, L), 0.0);
    specular = specular + light_uniforms.specular_intensity *
        pow(max(dot(-N, H),0.0), light_uniforms.specular_shininess);
}

let ambient:f32 = light_uniforms.ambient_intensity;
let final_color:vec3<f32> = (texture_color.rgb + in.v_color.rgb)*(ambient + diffuse) +
    light_uniforms.specular_color.xyz * specular;
return vec4<f32>(final_color, 1.0);
}
```

This shader is very similar to the one we used in Chapter 10, except that now, in addition to the vertex, normal, and UV data, we also pass the color data at *location* = 3 to our vertex shader. Within the *vs\_main* method, we process the UV data as well as the color data and store the outputs in the *v\_uv* and *v\_color* variables respectively. In the fragment shader, we then use the processed UV data (*v\_uv*) as our input to get the RGB color from the texture:

```
let texture_color:vec4<f32> = textureSample(texture_data, texture_sampler, in.v_uv);
```

The final color is obtained by combining the *texture\_color* with the processed color data (*v\_color*):

```
let final_color:vec3<f32> = (texture_color.rgb + in.v_color.rgb)*(ambient + diffuse) +
    light_uniforms.specular_color.xyz * specular;
```

Here *texture\_color* comes from our black-filled square texture, while *v\_color* comes from our colormaps defined using standard colormap names, such as *jet*, *hsv*, *summer*, etc.

## 11.3 Common Code

As we did in preceding chapters, to avoid code duplication, we will implement a common file called *common.rs* in the *examples/ch11/* folder, which we can then reuse to create a variety of different 3D surface charts.

Add a new Rust file called *common.rs* to the *examples/ch11/* folder and enter the following content into it:

```
#![allow(dead_code)]
use std::{iter, mem};
use cgmath::{Matrix, Matrix4, SquareMatrix};
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
};
use bytemuck:: {Pod, Zeroable, cast_slice};
#[path="../common/transforms.rs"]
pub mod transforms;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/texture_data.rs"]
mod texture;

const ANIMATION_SPEED:f32 = 1.0;
```

## 262 | Practical GPU Graphics with wgpu and Rust

```
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
    is_two_side: i32,
}

pub fn light(sc:[f32;3], ambient: f32, diffuse: f32, specular: f32, shininess: f32, two_side: i32) -> Light {
    Light {
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ambient,
        diffuse_intensity: diffuse,
        specular_intensity: specular,
        specular_shininess: shininess,
        is_two_side: two_side,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
    pub uv: [f32; 2],
    pub color: [f32; 4],
}

pub fn vertex(p:[f32;3], n:[f32; 3], t:[f32; 2], c:[f32; 3]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
        uv: [t[0], t[1]],
        color: [c[0], c[1], c[2], 1.0],
    }
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 4] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4, 2=>Float32x2, 3=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>)() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

pub struct State {
    pub init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
```

```

vertex_uniform_buffer: wgpu::Buffer,
view_mat: Matrix4<f32>,
project_mat: Matrix4<f32>,
num_vertices: u32,

image_texture: texture::Texture,
texture_bind_group: wgpu::BindGroup,
}

impl State {
    pub async fn new(window: &Window, vertex_data: &Vec<Vertex>, light_data: Light, img_file: &str,
        u_mode:wgpu::AddressMode, v_mode:wgpu::AddressMode) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        // create texture and texture bind group
        let image_texture = texture::Texture::create_texture_data(&init.device, &init.queue,img_file,
            u_mode, v_mode).unwrap();
        let texture_bind_group_layout =
            init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
                entries: &[
                    wgpu::BindGroupLayoutEntry {
                        binding: 0,
                        visibility: wgpu::ShaderStages::FRAGMENT,
                        ty: wgpu::BindingType::Texture {
                            multisampled: false,
                            view_dimension: wgpu::TextureViewDimension::D2,
                            sample_type: wgpu::TextureSampleType::Float { filterable: true },
                        },
                        count: None,
                    },
                    wgpu::BindGroupLayoutEntry {
                        binding: 1,
                        visibility: wgpu::ShaderStages::FRAGMENT,
                        ty: wgpu::BindingType::Sampler {
                            comparison: false,
                            filtering: true,
                        },
                        count: None,
                    },
                ],
                label: Some("Texture Bind Group Layout"),
            });
        let texture_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
            layout: &texture_bind_group_layout,
            entries: &[
                wgpu::BindGroupEntry {
                    binding: 0,
                    resource: wgpu::BindingResource::TextureView(&image_texture.view),
                },
                wgpu::BindGroupEntry {
                    binding: 1,
                    resource: wgpu::BindingResource::Sampler(&image_texture.sampler),
                },
            ],
            label: Some("Texture Bind Group"),
        });

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
        })
    }
}

```

```

source: wgpu::ShaderSource::Wgsl(include_str!("shader.wgsl").into()),
});

// uniform data
let camera_position = (2.5, 1.25, 2.5).into();
let look_direction = (0.0, 0.0, 0.0).into();
let up_direction = cgmath::Vector3::unit_y();

let (view_mat, project_mat, _view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
        init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

// create vertex uniform buffer
// model_mat and view_projection_mat will be stored in vertex_uniform_buffer
// inside the update function
let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 192,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// create fragment uniform buffer. here we set eye_position = camera_position and
// light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 36,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout =
    init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            }
        ]
    })
);

```

```

    },
    wgpu::BindGroupLayoutEntry {
        binding: 1,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    },
    wgpu::BindGroupLayoutEntry {
        binding: 2,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    }
],
label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout, &texture_bind_group_layout],
    push_constant_ranges: &[],
});
}

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,

```

## 266 | Practical GPU Graphics with wgpu and Rust

```
entry_point: "fs_main",
targets: &[wgpu::ColorTargetState {
    format: init.config.format,
    blend: Some(wgpu::BlendState {
        color: wgpu::BlendComponent::REPLACE,
        alpha: wgpu::BlendComponent::REPLACE,
    }),
    write_mask: wgpu::ColorWrites::ALL,
}],
},
primitive: wgpu::PrimitiveState{
    topology: wgpu::PrimitiveTopology::TriangleList,
    ..Default::default()
},
//depth_stencil: None,
depth_stencil: Some(wgpu::DepthStencilState {
    format: wgpu::TextureFormat::Depth24Plus,
    depth_write_enabled: true,
    depth_compare: wgpu::CompareFunction::LessEqual,
    stencil: wgpu::StencilState::default(),
    bias: wgpu::DepthBiasState::default(),
}),
multisample: wgpu::MultisampleState::default(),
});

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = vertex_data.len() as u32;

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    view_mat,
    project_mat,
    num_vertices,

    image_texture,
    texture_bind_group
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
pub fn input(&mut self, event: &WindowEvent) -> bool {
```

```

        false
    }

pub fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0,0.0,0.0], [dt.sin(), dt.cos(), 0.0],
  [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;

    let normal_mat = (model_mat.invert().unwrap()).transpose();

    let model_ref:&[f32; 16] = model_mat.as_ref();
    let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
    let normal_ref:&[f32; 16] = normal_mat.as_ref();

    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
                                bytemuck::cast_slice(view_projection_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));
}

pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    //let output = self.init.surface.get_current_frame()?.output;
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,

```

```

        b: 0.314,
        a: 1.0,
    }),
    store: true,
),
],
//depth_stencil_attachment: None,
depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
    view: &depth_view,
    depth_ops: Some(wgpu::Operations {
        load: wgpu::LoadOp::Clear(1.0),
        store: false,
    }),
    stencil_ops: None,
}),
),
);
});

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.set_bind_group(1, &self.texture_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

pub fn run(vertex_data: &Vec<Vertex>, light_data: Light, file_name: &str, colormap_name: &str,
u_mode:wgpu::AddressMode, v_mode:wgpu::AddressMode, title: &str) {
    let path = "examples/ch11/assets/";
    let img_file = [path, file_name].join("");

    env_logger::init();
    let event_loop = EventLoop::new();
    let window = winit::window::WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("ch11_{} - image file: {}, colormap: {}", title, file_name,
        colormap_name));

    let mut state = pollster::block_on(State::new(&window, &vertex_data, light_data, &img_file,
        u_mode, v_mode));
    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                }
                        }
                    }
                }
            }
        }
    });
}
}

```

```

        ..,
        ..
    } => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {:?}", e, e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

This code is essentially similar to that used in preceding chapters, except that our *Vertex* struct now contains position, normal, UV, and colormap data. We assign the *attributes* field of our vertex buffer layout with a *vertex\_attr\_array* that has four elements: vertex positions at location 0, normals at location 1, UVs at location 2, and colormap data at location 3. This example brings almost all of the features we have learned so far, including lighting, colormaps, and textures.

The *State::new* function in the above code allows us to specify the texture image and supply the vertex, normal, color, and UV data to the program. This means that we can use the same code to create different 3D surface charts simply by changing the attributes and data.

Here, to avoid code duplication, we also convert the content that is usually found in the *main* function into a new *run* function.

## 11.4 Simple 3D Surface Charts

Previously, we implemented the *simple\_surface\_data* method in the *surface\_data.rs* file in the *examples/common/* folder. Now, we need to modify this method to create UV (texture coordinates) data for our square texture for each quad mesh. Add the following highlighted code snippet to the *simple\_surface\_data* method:

```

pub fn simple_surface_data(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, xmin:f32, xmax:f32,
zmin:f32, zmax:f32, nx:usize, nz: usize, scale:f32, scaley:f32) ->
(Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;2]>, Vec<[f32;2]>) {

```

```

let dx = (xmax-xmin)/(nx as f32-1.0);
let dz = (zmax-zmin)/(nz as f32-1.0);
let mut ymin1: f32 = 0.0;
let mut ymax1: f32 = 0.0;

let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nz]; nx];
for i in 0..nx {
    let x = xmin + i as f32 * dx;
    let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nz);
    for j in 0..nz {
        let z = zmin + j as f32 * dz;
        let pt = f(x, z);
        pt1.push(pt);
        ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
        ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
    }
    pts[i] = pt1;
}

let ymin = ymin1 - scaley * (ymax1 - ymin1);
let ymax = ymax1 + scaley * (ymax1 - ymin1);

for i in 0..nx {
    for j in 0..nz {
        pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
    }
}

let cmin = normalize_point([0.0, ymin1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];
let cmax = normalize_point([0.0, ymax1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];

let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
let mut colors: Vec<[f32; 3]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
let mut uvs: Vec<[f32; 2]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);
let mut uv1: Vec<[f32; 2]> = Vec::with_capacity((4* (nx - 1)*(nz -1)) as usize);

for i in 0..nx - 1 {
    for j in 0.. nz - 1 {
        let p0 = pts[i][j];
        let p1 = pts[i][j+1];
        let p2 = pts[i+1][j+1];
        let p3 = pts[i+1][j];

        let ( mut pos, mut norm, mut col, mut uv) =
            create_quad(p0, p1, p2, p3, cmin, cmax, colormap_name, scale);

        // positions
        positions.append(&mut pos);

        // normals
        normals.append(&mut norm);

        // colors
        colors.append(&mut col);

        // uvs
        uvs.append(&mut uv);
    }
}

```

```

        // uv1
        uv1.push([0.0, 0.0]);
        uv1.push([0.0, 1.0]);
        uv1.push([1.0, 1.0]);
        uv1.push([1.0, 0.0]);
        uv1.push([1.0, 0.0]);
        uv1.push([0.0, 0.0]);
    }
}
(positions, normals, colors, uvs, uv1)
}

```

Here, the `uv1` array represents the UV data for our square texture. It is very simple: each square texture is mapped onto a single quad mesh made out of two triangles.

In the following sections, I will demonstrate how to use this method to create two simple surfaces, including `sinc` and `peaks`, with both textures and colormaps.

### 11.4.1 Sinc Surface Chart

Add a new Rust file called `sinc.rs` to the `examples/ch11/` folder with the following code:

```

mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::simple_surface_data(&math::sinc, colormap_name,
        -8.0, 8.0, -8.0, 8.0, 30, 30, 1.5, 0.3);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }

    let vertex_data = create_vertices(colormap_name);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "sinc");
}

```

## 272 | Practical GPU Graphics with wgpu and Rust

This code first implements a `create_vertices` method that calls the `surface::simple_surface_data` function to generate the vertex position, normal vector, UV, and colormap data, and then converts these data into the proper format to be used in creating our textured *sinc* chart.

Inside the main function, we set two user input fields which will let you specify different texture image files and colormap names. Here, the default values for these two fields are “`whitesquare2.png`” and “`jet`” respectively.

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch11_sinc"
path = "examples/ch11/sinc.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch11_sinc
```

This produces the results shown in Fig.11-4.

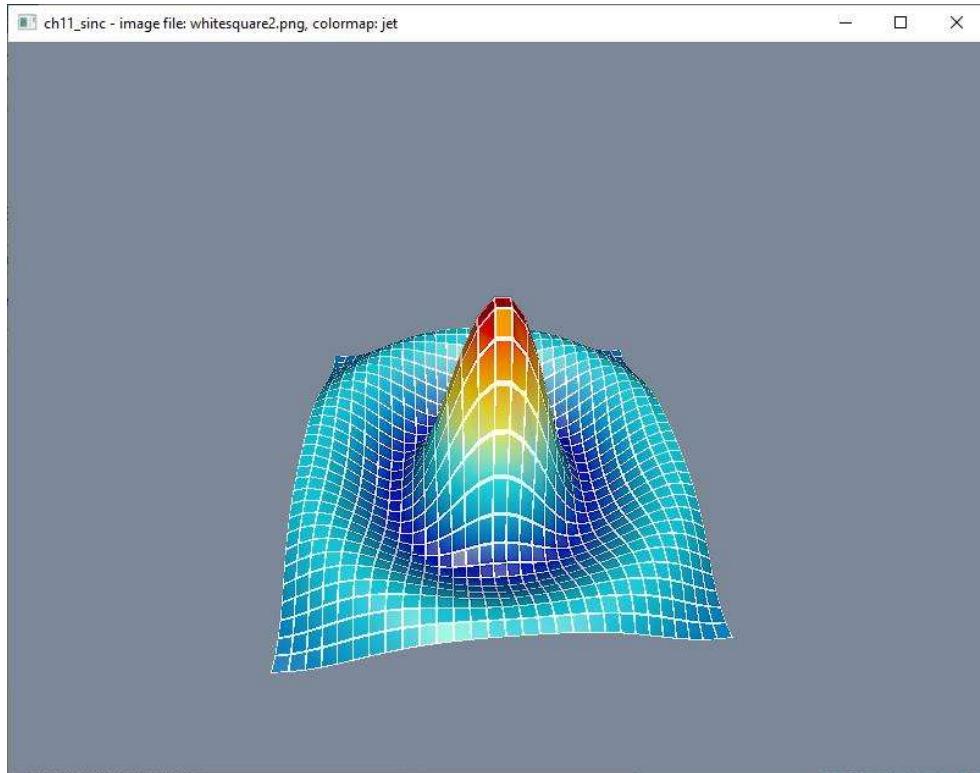
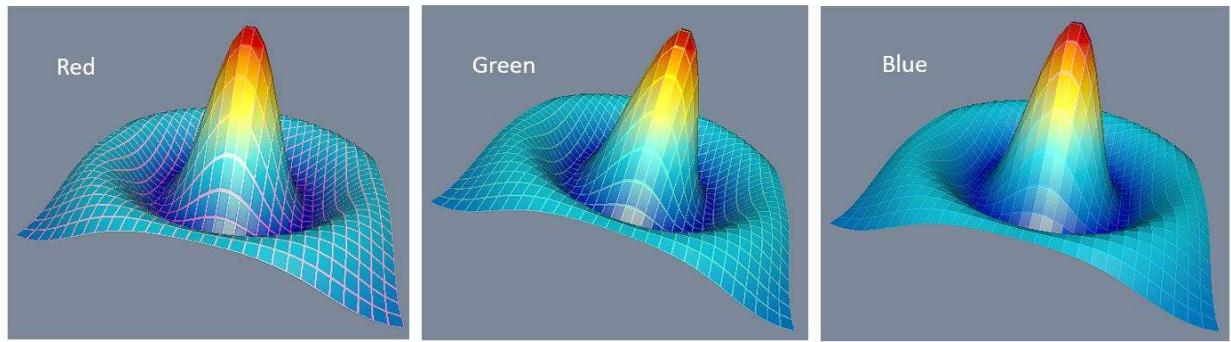


Fig.11-4. Sinc surface chart with default parameters.

Fig.11-5 shows *sinc* surface charts with different image textures, which are generated by executing the following respective commands:

```
cargo run --example ch11_sinc "redsquare2.png"
cargo run --example ch11_sinc "greensquare2.png"
cargo run --example ch11_sinc "bluesquare2.png"
```



*Fig.11-5. Sinc charts with different square textures.*

You may notice that there is an issue associated with the wireframe when you try selecting a square texture image of any color other than white. The wireframes of our *sinc* charts mapped using the red, green, or blue colored square images do not really look like they are a pure red, green, or blue color. In fact, the wireframe color seems to vary depending on the underlying colormaps.

This can be explained if we take a look at our shader program. Remember that inside the fragment shader, the final color of the surface is the sum of the texture color and the colormap:

```
final_color ~ texture_color + v_color
```

Thus, for our square textures with different colors, we have the following results:

```
//white square texture:  
Final_color = texture_color (1.0, 1.0, 1.0) + v_color (r, g, b) = (1.0, 1.0, 1.0);  
  
//red square texture:  
Final_color = texture_color (1.0, 0.0, 0.0) + v_color (r, g, b) = (1.0, g, b);  
  
//green square texture:  
Final_color = texture_color (0.0, 1.0, 0.0) + v_color (r, g, b) = (r, 1.0, b);  
  
//blue square texture:  
Final_color = texture_color (0.0, 0.0, 1.0) + v_color (r, g, b) = (r, g, 1.0);
```

Since the shader program in *wgpu* automatically clamps the color component to 1.0 if it exceeds 1.0, the white-colored square texture always shows up as a pure white color regardless of the contribution from the colormap. But for wireframes mapped with the red, green, or blue square textures, the final color is affected by the underlying colormaps, even though its dominant color component still comes from the square texture. That is why you will see the wireframe with different colors at different locations on our surface chart. Regardless, we still obtain decent looking 3D surface charts using our square textures. In particular, with this approach, we can overcome the difficulties associated with the line-thickness of our wireframe being limited to one pixel because we can easily create a square image with a stroke of arbitrary thickness.

You may notice that we did not use a black square for our wireframe. This is because a black-colored square texture does not contribute anything to the final color because its color components are always zero:

```
//black square texture:  
Final_color = texture_color (0.0, 0.0, 0.0) + v_color (r, g, b) = (r, g, b);
```

This means the black square texture is completely transparent, and a wireframe made using such a square texture totally disappears – it looks like the surface chart has no texture mapping on it, only a colormap.

## 11.4.2 Peaks Surface Chart

Add a new Rust file called *peak.rs* to the *examples/ch11/* folder and type the following code into it:

```
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::simple_surface_data(&math::peaks, colormap_name,
        -3.0, 3.0, -3.0, 3.0, 31, 31, 1.5, 0.0);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }

    let vertex_data = create_vertices(colormap_name);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "peaks");
}
```

This code first implements a *create\_vertices* method that calls the *surface::simple\_surface\_data* function to generate the vertex position, normal vector, UV, and colormap data, and then converts these data into the proper format to be used in creating our textured *peaks* chart.

Inside the main function, we set two user input fields which will let you specify different image files and colormap names. Here, the default values for these two fields are “*whitesquare2.png*” and “*jet*” respectively.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch11_peaks"
path = "examples/ch11/peaks.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch11_peaks "redsquare2.png" "hsv"
```

This produces the results shown in Fig.11-6.

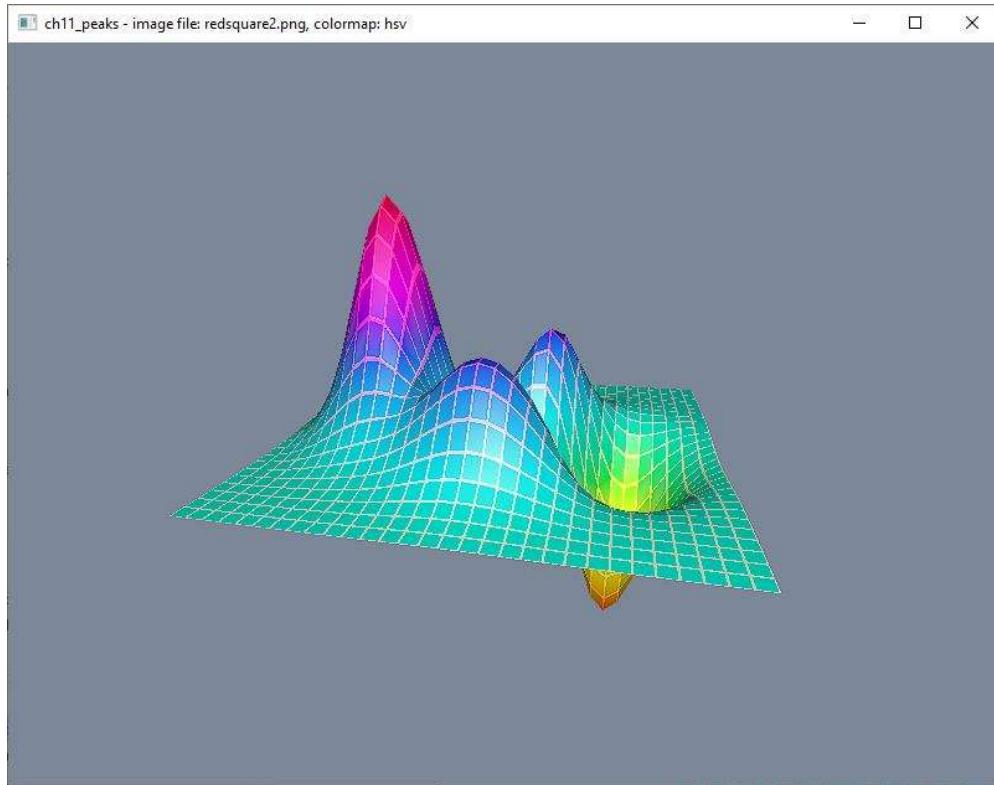


Fig. 11-6. Peaks surface chart.

## 11.5 Parametric 3D Surface Charts

Previously, we implemented the *parametric\_surface\_data* method in the *surface\_data.rs* file in the *examples/common/* folder. Now, we need to modify this method to create UV (texture coordinates) data for our square texture for each quad mesh. Add the following highlighted code snippet to the *parametric\_surface\_data* method:

```
pub fn parametric_surface_data(f: &dyn Fn(f32, f32) -> [f32; 3], colormap_name: &str, umin:f32, umax:f32,
vmin:f32, vmax:f32, nu:usize, nv: usize, xmin:f32, xmax:f32, zmin:f32, zmax:f32, scale: f32, scaley: f32)
-> (Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;3]>, Vec<[f32;2]>, Vec<[f32;2]>) {
    let du = (umax-umin)/(nu as f32-1.0);
    let dv = (vmax-vmin)/(nv as f32-1.0);
    let mut ymin1: f32 = 0.0;
    let mut ymax1: f32 = 0.0;

    let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nv]; nu];
    for i in 0..nu {
        let u = umin + i as f32 * du;
        let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nv);
        for j in 0..nv {
            let v = vmin + j as f32 * dv;
            let pt = f(u, v);
            pt1.push(pt);
            ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
            ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
        }
    }
}
```

## 276 | Practical GPU Graphics with wgpu and Rust

```

        pts[i] = pt1;
    }

let ymin = ymin1 - scaley * (ymax1 - ymin1);
let ymax = ymax1 + scaley * (ymax1 - ymin1);

for i in 0..nu {
    for j in 0..nv {
        pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
    }
}

let cmin = normalize_point([0.0, ymin1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];
let cmax = normalize_point([0.0, ymax1, 0.0], xmin, xmax, ymin, ymax, zmin, zmax, scale)[1];

let mut positions: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let mut normals: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let mut colors: Vec<[f32; 3]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let uvs: Vec<[f32; 2]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);
let mut uv1: Vec<[f32; 2]> = Vec::with_capacity((4*(nu - 1)*(nv - 1)) as usize);

for i in 0..nu - 1 {
    for j in 0..nv - 1 {
        let p0 = pts[i][j];
        let p1 = pts[i+1][j];
        let p2 = pts[i+1][j+1];
        let p3 = pts[i][j+1];

        let (mut pos, mut norm, mut col, _uv) = create_quad(p0, p1, p2, p3, cmin, cmax,
            colormap_name, scale);

        // positions
        positions.append(&mut pos);

        // normals
        normals.append(&mut norm);

        // colors
        colors.append(&mut col);

        // uv1
        uv1.push([0.0, 0.0]);
        uv1.push([1.0, 0.0]);
        uv1.push([1.0, 1.0]);
        uv1.push([1.0, 1.0]);
        uv1.push([0.0, 1.0]);
        uv1.push([0.0, 0.0]);
    }
}
(positions, normals, colors, uvs, uv1)
}

```

Note that the texture (*uv1*) data for the parametric surface is identical to that used in the simple surface, but in this case, the texture coordinates are created in parametric space.

In the following sections, I will demonstrate how to use this method to create several parametric surfaces, including sphere, torus, Klein bottle, and Wellenkugel surfaces, with both textures and colormaps.

## 11.5.1 Sphere Surface Chart

We can use the `parametric_surface_data` method to create a sphere surface. This can be done by simply adding a method called `sphere` to the `math_func.rs` file in the `examples/common/` folder with the following code:

```
pub fn sphere(u:f32, v:f32) -> [f32; 3]{
    let x = v.cos() * u.cos();
    let y = v.sin();
    let z = v.cos() * u.sin();
    [x, y, z]
}
```

Now, add a new Rust file called `sphere.rs` to the `examples/ch11/` folder with the following code:

```
use std::f32::consts::PI;
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::parametric_surface_data(&math::sphere, colormap_name,
        0.0, 2.0*PI, -PI/2.0, PI/2.0, 20, 15, -1.0, 1.0, -1.0, 1.0, 1.5, 0.0);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }

    let vertex_data = create_vertices(colormap_name);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "sphere");
}
```

This code first implements a `create_vertices` method that calls the `surface::parametric_surface_data` function to generate the vertex position, normal vector, UV, and colormap data, and then converts these data into the proper format to be used in creating our textured `sphere` chart.

Inside the main function, we set two user input fields which will let you specify different image files and colormap names. Here, the default values for these two fields are “`whitesquare2.png`” and “`jet`” respectively.

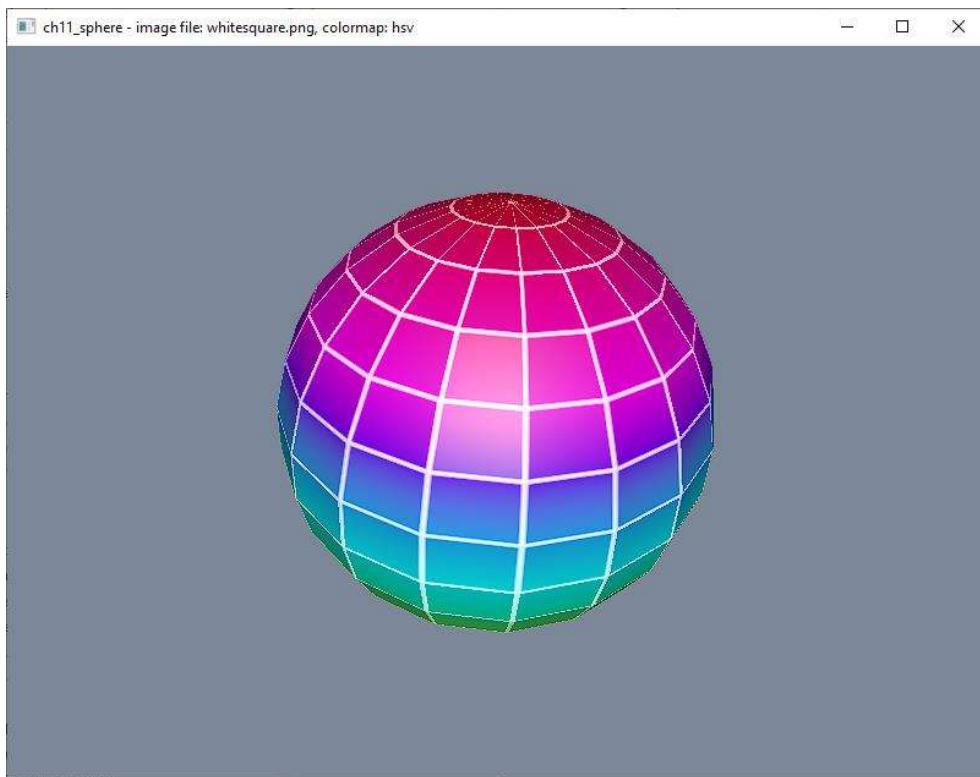
Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch11_sphere"
path = "examples/ch11/sphere.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch11_sphere "whitesquare.png" "hsv"
```

This produces the results shown in Fig.11-7.



*Fig.11-7. Sphere surface chart.*

## 11.5.2 Torus Surface Chart

We can also use the *parametric\_surface\_data* method to create a torus surface chart. Add a new method called *torus* to the *math\_func.rs* file with the following code:

```
pub fn torus(u:f32, v:f32) -> [f32; 3] {
    let x = (1+0.3 * v.cos()) * u.cos();
    let y = 0.3 * v.sin();
    let z = (1 + 0.3 * v.cos()) * u.sin();
    [x, y, z]
}
```

Now, add a new Rust file called *torus.rs* to the *examples/ch11/* folder with the following code:

```
use std::f32::consts::PI;
mod common;
#[path="../common/surface_data.rs"]
```

```

mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::parametric_surface_data(&math::torus, colormap_name,
        0.0, 2.0*PI, 0.0, 2.0*PI, 40, 15, -1.0, 1.0, -1.0, 1.0, 1.5, 1.5);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }

    let vertex_data = create_vertices(colormap_name);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "torus");
}

```

This code first implements a *create\_vertices* method that calls the *surface::parametric\_surface\_data* function to generate the vertex position, normal vector, UV, and colormap data, and then converts these data into a proper format to be used in creating our textured *torus* chart.

Inside the main function, we set two user input fields which will let you specify different image files and colormap names. Here, the default values for these two fields are “*whitesquare2.png*” and “*jet*” respectively.

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch11_torus"
path = "examples/ch11/torus.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch11_torus
```

This produces the results shown in Fig.11-8.

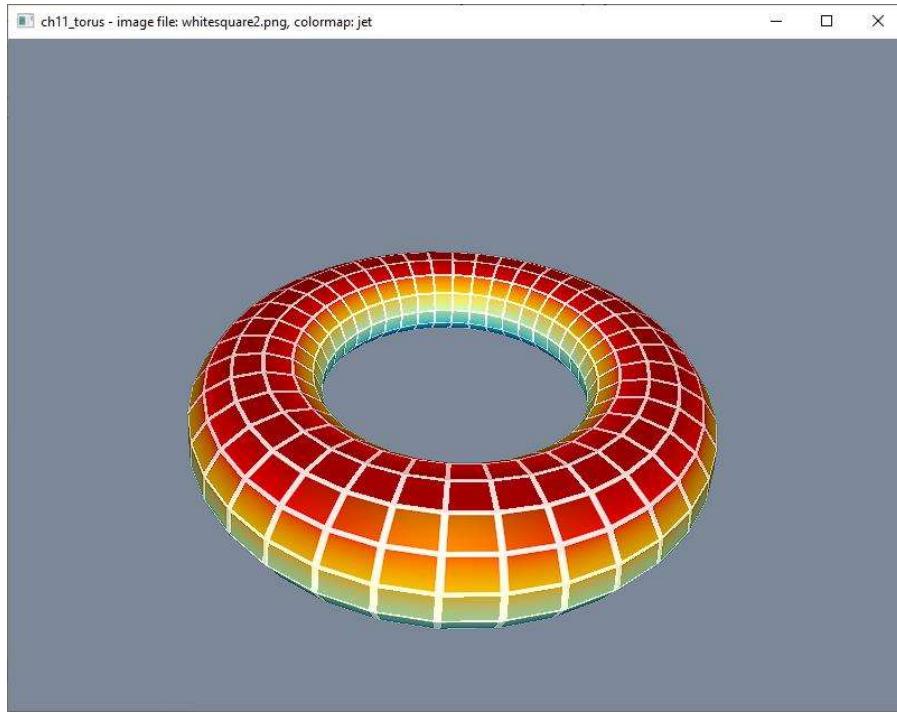


Fig. 11-8. Torus surface chart.

### 11.5.3 Klein Bottle Surface Chart

Add a new Rust file called *klein-bottle.ts* to the *examples/ch11/* folder with the following code:

```
use std::f32::consts::PI;
mod common;
#[path = "../common/surface_data.rs"]
mod surface;
#[path = "../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::parametric_surface_data(&math::klein_bottle,
        colormap_name, 0.0, PI, 0.0, 2.0*PI, 70, 30, -2.0, 2.0, -2.0, 3.0, 2.0, 0.0);
    let mut data: Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }
}
```

```

let vertex_data = create_vertices(colormap_name);
let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
let u_mode = wgpu::AddressMode::ClampToEdge;
let v_mode = wgpu::AddressMode::ClampToEdge;

common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "klein_bottle");
}

```

This code first implements a *create\_vertices* method that calls the *surface::parametric\_surface\_data* function to generate the vertex position, normal vector, UV, and colormap data, and then converts those data into a proper format, which will be used to create our textured Klein bottle chart.

Inside the main function, we set two user input fields which will let you specify different image files and colormap names. Here, the default values for these two fields are “whitesquare2.png” and “jet” respectively.

Now, add the following code snippet to the *Cargo.toml* file:

```

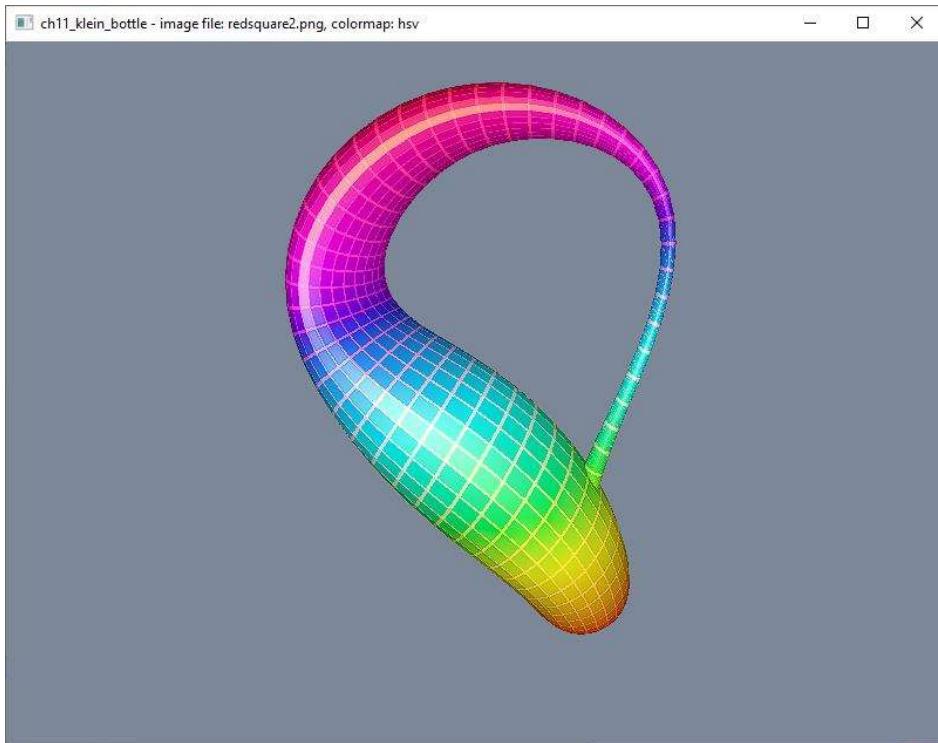
[[example]]
name = "ch11_klein_bottle"
path = "examples/ch11/klein_bottle.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch11_klein_bottle "redsquare2.png" "hsv"
```

This produces the results shown in Fig.11-9.



*Fig.11-9. Klein bottle surface chart.*

## 11.5.4 Wellenkugel Surface Chart

Add a new Rust file called `wellenkugel.rs` to the `examples/ch11/` folder with the following code:

```
mod common;
#[path="../common/surface_data.rs"]
mod surface;
#[path="../common/math_func.rs"]
mod math;

fn create_vertices(colormap_name: &str) -> Vec<common::Vertex> {
    let (pos, normal, color, _uv, uv1) = surface::parametric_surface_data(&math::wellenkugel,
        colormap_name, 0.0, 14.5, 0.0, 5.0, 70, 30, -10.0, 10.0, -10.0, 10.0, 1.2, 0.0);
    let mut data:Vec<common::Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(common::vertex(pos[i], normal[i], uv1[i], color[i]));
    }
    data.to_vec()
}

fn main(){
    let mut file_name = "whitesquare2.png";
    let mut colormap_name = "jet";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        file_name = &args[1];
    }
    if args.len() > 2 {
        colormap_name = &args[2];
    }

    let vertex_data = create_vertices(colormap_name);
    let light_data = common::light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
    let u_mode = wgpu::AddressMode::ClampToEdge;
    let v_mode = wgpu::AddressMode::ClampToEdge;

    common::run(&vertex_data, light_data, file_name, colormap_name, u_mode, v_mode, "wellenkugel");
}
```

This code first implements a `create_vertices` method that calls the `surface::parametric_surface_data` function to generate the vertex position, normal vector, UV, and colormap data, and then converts these data into a proper format to be used in creating our textured Wellenkugel chart.

Inside the main function, we set two user input fields which will let you specify different image files and colormap names. Here, the default values for these two fields are “`whitesquare2.png`” and “`jet`” respectively.

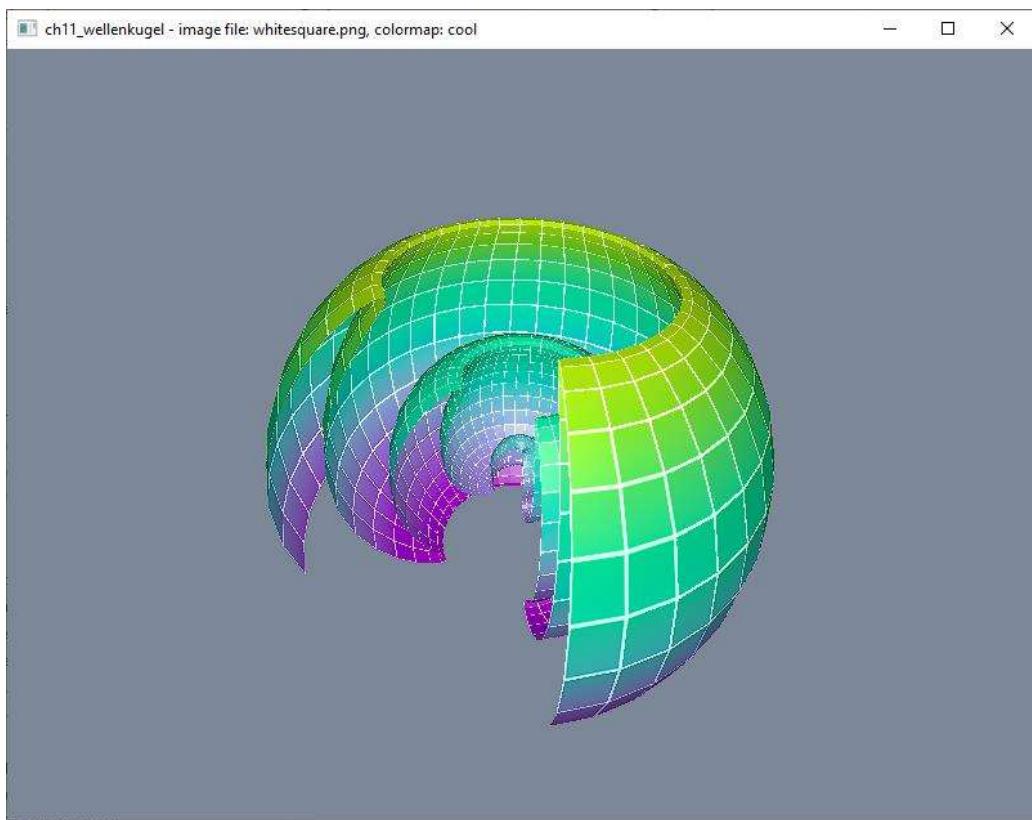
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch11_wellenkugel"
path = "examples/ch11/wellenkugel.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch11_wellenkugel "whitesquare.png" "cool"
```

This produces the results shown in Fig.11-10.



*Fig.11-10. Wellenkugel surface chart.*



# 12 Creating Multiple Objects

You might have noticed that in all of the examples presented so far, we have rendered only a single object per scene. However, a typical 3D scene usually consists of multiple objects. In this chapter, you will learn how to render multiple objects in a single scene.

There are many ways in *wgpu* to create multiple objects. If you want to render the same object multiple times in a scene, you can perform different uniform transformations on the object, then create different uniform bind groups, and finally call the *draw* method multiple times to render multiple copies of that object at different locations in the scene. You can also use instancing to draw the same object multiple times with different attributes, including position, orientation, size, color, etc.

In some situations, you may want to render different objects in a scene, but keep the same environment (lighting, color, texture, shaders, etc.). In this case, the most convenient approach is to first combine the different objects' respective vertex data, color data, normal vector data, and UV data together, and then render them as a single big object. This way, you do not need to make any changes to the corresponding program code, and you can use the same shader program you would use to render a single object under the same environment.

The above-mentioned approaches have some limitations – they are either limited to creating multiples of the same objects or limited to creating different objects under the same environment. However, in more general cases, you often want to render multiple different objects under different environments – some objects need lighting effects, some objects only need solid colors, while other objects need texture mapping. Fortunately, *wgpu* allows you to set different pipelines or different render passes, so you can deal with this general situation.

## 12.1 Creating Two Cubes

In this section, I will show you how to render the same object multiple times in a scene. Specifically, I will create two cubes using uniform transformations. Following the procedure presented here, you can easily render as many of the same objects as you need.

### 12.1.1 Rust Code

This example is based on the previous example of the cube with distinct face colors presented in Chapter 6. Add a new sub-folder named *ch12* in the *examples/* folder. Copy two files, *rotate\_cube.rs* and *cube\_face\_color.wgsl*, from the *examples/ch06/* folder and paste them into the *examples/ch12/* folder. Then, rename these two files with the following names: *two\_cubes.rs* and *two\_cubes.wgsl*. In this

## 286 | Practical GPU Graphics with wgpu and Rust

example, we can use the same shader code without any changes, but we need to modify the Rust code in the `two_cubes.rs` file.

Now, change the `two_cubes.rs` file to contain the following code:

```
use std:: {iter, mem};
use cgmath::Matrix4;
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 4],
    color: [f32; 4],
}

fn vertex(p:[i8;3], c:[i8; 3]) -> Vertex {
    Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
    }
}

fn create_vertices() -> Vec<Vertex> {
    let (pos, col, _uv, _normal) = vertex_data::cube_data();
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], col[i]));
    }
    data.to_vec()
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>) as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
```

```

vertex_buffer: wgpu::Buffer,
uniform_bind_group1:wgpu::BindGroup,
uniform_bind_group2:wgpu::BindGroup,
uniform_buffer: wgpu::Buffer,
uniform_offset: u64,
view_mat: Matrix4<f32>,
project_mat: Matrix4<f32>,
}

impl State {
    async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("two_cubes.wgsl").into()),
        });

        // uniform data
        let camera_position = (3.0, 1.5, 3.0).into();
        let look_direction = (0.0, 0.0, 0.0).into();
        let up_direction = cgmath::Vector3::unit_y();

        let (view_mat, project_mat, _view_project_mat) =
            transforms::create_view_projection(camera_position, look_direction, up_direction,
                init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

        let matrix_size = 4 * 16;
        let uniform_offset = 256; // uniform_bind_group must be 256-byte aligned
        let uniform_buffer_size = uniform_offset + matrix_size;

        let uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
            label: Some("Uniform Buffer"),
            size: uniform_buffer_size,
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
            mapped_at_creation: false,
        });

        let uniform_bind_group_layout = init.device.create_bind_group_layout(
            &wgpu::BindGroupLayoutDescriptor{
                entries: &[wgpu::BindGroupLayoutEntry {
                    binding: 0,
                    visibility: wgpu::ShaderStages::VERTEX,
                    ty: wgpu::BindingType::Buffer {
                        ty: wgpu::BufferBindingType::Uniform,
                        has_dynamic_offset: false,
                        min_binding_size: None,
                    },
                    count: None,
                }],
                label: Some("Uniform Bind Group Layout"),
            });
    }

    let uniform_bind_group1 = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
        layout: &uniform_bind_group_layout,
        entries: &[wgpu::BindGroupEntry {
            binding: 0,
            resource: wgpu::BindingResource::Buffer (
                wgpu::BufferBinding {
                    buffer: &uniform_buffer,

```

## 288 | Practical GPU Graphics with wgpu and Rust

```
        size: Some(core::num::NonZeroU64::new(matrix_size).unwrap()),
        offset: 0,
    },
),
},
label: Some("Uniform Bind Group 1"),
});

let uniform_bind_group2 = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: wgpu::BindingResource::Buffer (
            wgpu::BufferBinding {
                buffer: &uniform_buffer,
                size: Some(core::num::NonZeroU64::new(matrix_size).unwrap()),
                offset: uniform_offset,
            }
        ),
    }],
    label: Some("Uniform Bind Group 2"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
    //depth_stencil: None,
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
```

```

        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
}

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&create_vertices()),
    usage: wgpu::BufferUsages::VERTEX,
});

Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group1,
    uniform_bind_group2,
    uniform_buffer,
    uniform_offset,
    view_mat,
    project_mat,
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    // for cube 1
    let model_mat1 = transforms::create_transforms([-2.0, -1.5, 0.5], [dt.sin(), dt.cos(), 0.0],
        [1.0, 1.0, 1.0]);
    let mvp_mat1 = self.project_mat * self.view_mat * model_mat1;
    let mvp_ref1:&[f32; 16] = mvp_mat1.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref1));

    // for cube 2
    let model_mat2 = transforms::create_transforms([-0.5, 1.0, -1.5], [0.0, dt.sin(), dt.cos()],
        [1.0, 1.0, 1.0]);
    let mvp_mat2 = self.project_mat * self.view_mat * model_mat2;
    let mvp_ref2:&[f32; 16] = mvp_mat2.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer, self.uniform_offset,
        bytemuck::cast_slice(mvp_ref2));
}
}

```

## 290 | Practical GPU Graphics with wgpu and Rust

```
fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format: wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
                view: &depth_view,
                depth_ops: Some(wgpu::Operations {
                    load: wgpu::LoadOp::Clear(1.0),
                    store: false,
                }),
                stencil_ops: None,
            }),
        });
        render_pass.set_pipeline(&self.pipeline);
        render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));

        // draw first cube
        render_pass.set_bind_group(0, &self.uniform_bind_group1, &[]);
        render_pass.draw(0..36, 0..1);
    }
}
```

```

    // draw second cube
    render_pass.set_bind_group(0, &self.uniform_bind_group2, &[]);
    render_pass.draw(0..36, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}

fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&*format!("{}", "ch12_two_cubes"));
    let mut state = pollster::block_on(State::new(&window));

    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                            ..
                        } => *control_flow = ControlFlow::Exit,
                        WindowEvent::Resized(physical_size) => {
                            state.resize(*physical_size);
                        }
                        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                            state.resize(**new_inner_size);
                        }
                        _ => {}
                    }
                }
            }
            Event::RedrawRequested(_) => {
                let now = std::time::Instant::now();
                let dt = now - render_start_time;
                state.update(dt);

                match state.render() {
                    Ok(_) => {}
                    Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
                    Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
                    Err(e) => eprintln!("{}: {}", e, e),
                }
            }
        }
    })
}

```

```

        }
    }
    Event::MainEventsCleared => {
        window.request_redraw();
    }
    _ => {}
);
});
}

```

Here, we create two uniform bind groups with the same uniform buffer and layout, but with different offsets. Note that the offset for uniform bind groups must be 256-byte aligned.

Inside the update function, we create two separate model matrices with different translations and rotations, which will determine the locations and rotations of our two cubes. We then use these two model matrices to construct two corresponding model-view-projection matrices, which are used to create the uniform buffer by calling the `queue.write_buffer` method.

Finally, we draw the two cubes inside the `render` method with the following code snippet:

```

// draw first cube
render_pass.set_bind_group(0, &self.uniform_bind_group1, &[]);
render_pass.draw(0..36, 0..1);

// draw second cube
render_pass.set_bind_group(0, &self.uniform_bind_group2, &[]);
render_pass.draw(0..36, 0..1);

```

Here, we render different cubes by specifying different uniform bind groups. This way, you can create as many cubes as you want.

## 12.1.2 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```

[[example]]
name = "ch12_two_cubes"
path = "examples/ch12/two_cubes.rs"

```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch12_two_cubes
```

This produces the results shown in Fig.12-1.

## 12.2 Creating Multiple Cubes with Instancing

The `wgpu` API has a feature called instanced drawing. It is basically a way to draw more than one of the same object that is faster than drawing each object individually. Let us look at the input arguments for the `draw` method in `wgpu`:

```
pub fn draw(&mut self, vertices: Range<u32>, instances: Range<u32>)
```

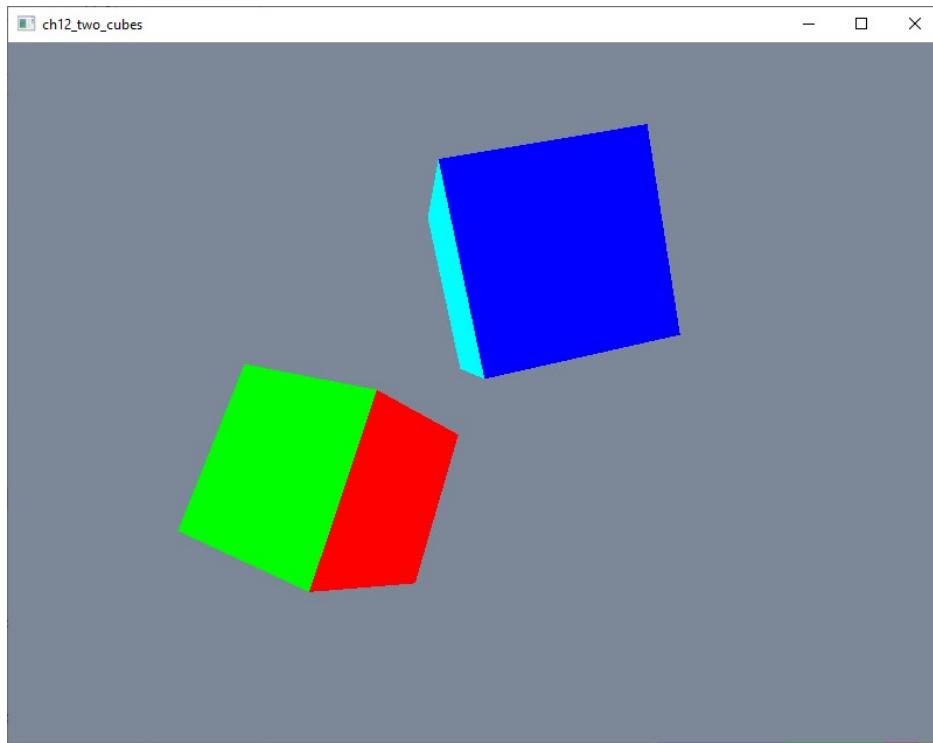


Fig.12-1. Two cubes created with uniform transformations.

The `instances` parameter takes a `Range<u32>` value. This parameter tells the GPU how many copies, or instances, of our object we want to draw with a single call to the `draw` function. So far, we have always specified `0..1`, which instructs the GPU to draw our object once and then stop. If we set it to `0..10`, our program would draw 10 instances, while if we used `3..4`, our code would still only draw one instance of our object. The reason we want to be able to specify a particular range for this parameter is that sometimes we do not want to draw all of our objects.

In the following subsections, I will explain how to draw multiple copies of our cube using instancing.

### 12.2.1 Rust Code

Add a new Rust file called `cube_instance.rs` to the `examples/ch12/` folder with the following content:

```
use std:: {convert::TryInto, iter, mem};
use cgmath:: Matrix4;
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable, cast_slice};

#[path = "../common/transforms.rs"]
mod transforms;
#[path = "../common/vertex_data.rs"]
mod vertex_data;
```

## 294 | Practical GPU Graphics with wgpu and Rust

```
const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;
const NX: usize = 7;
const NY: usize = 5;
const NXY: usize = 35;

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex {
    position: [f32; 4],
    color: [f32; 4],
}

fn vertex(p:[i8;3], c:[i8; 3]) -> Vertex {
    Vertex {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
    }
}

fn create_vertices() -> Vec<Vertex> {
    let (pos, col, _uv, _normal) = vertex_data::cube_data();
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], col[i]));
    }
    data.to_vec()
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

#[repr(C)]
#[derive(Copy, Clone, bytemuck::Pod, bytemuck::Zeroable)]
struct InstanceMat {
    model: [[f32; 4]; 4],
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    uniform_buffer: wgpu::Buffer,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
}

impl State {
    async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;
```

```

let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!("cube_instance.wgsl").into()),
});

// uniform data
let camera_position = (0.0, 0.0, -16.0).into();
let look_direction = (0.0, 0.0, 0.0).into();
let up_direction = cgmath::Vector3::unit_y();

let (view_mat, project_mat, _view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
        init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

let matrix_size = 4 * 16;
let uniform_buffer_size = NXY as u64 * matrix_size;

let uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Uniform Buffer"),
    size: uniform_buffer_size,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

let uniform_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor{
        entries: &[wgpu::BindGroupLayoutEntry {
            binding: 0,
            visibility: wgpu::ShaderStages::VERTEX,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Uniform,
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        }],
        label: Some("Uniform Bind Group Layout"),
    });
};

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group"),
});
};

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
};

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
}
);

```

## 296 | Practical GPU Graphics with wgpu and Rust

```
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        strip_index_format: None,
        cull_mode: Some(wgpu::Face::Back),
        ..Default::default()
    },
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
});
```

```
let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&create_vertices()),
    usage: wgpu::BufferUsages::VERTEX,
});
```

```
Self {
    init,
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    uniform_buffer,
    view_mat,
    project_mat,
}
```

```
}
```

```
pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
```

```

fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = dt.as_secs_f32() * ANIMATION_SPEED;

    let mut mvp_mat:Vec<[f32;16]> = Vec::with_capacity(NXY as usize);

    for x in 0..NX {
        for y in 0..NY{
            let xx = 4.0 * (x as f32 - NX as f32/2.0 + 0.5);
            let yy = 4.0 * (y as f32 - NY as f32/2.0 + 0.5);
            let tx = dt * (x as f32 + 0.5);
            let ty = dt * (y as f32 + 0.5);
            let model_mat = transforms::create_transforms([xx,yy,0.0], [tx.sin(), ty.cos(), 0.0],
                [1.0, 1.0, 1.0]);
            let mvp = self.project_mat * self.view_mat * model_mat;
            let mvp1:&[f32;16] = mvp.as_ref();
            mvp_mat.push(*mvp1);
        }
    }

    let mvp_ref:&[[f32; 16]; NXY] = mvp_mat[..].try_into().unwrap();
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,

```

## 298 | Practical GPU Graphics with wgpu and Rust

```
        resolve_target: None,
        ops: wgpu::Operations {
            load: wgpu::LoadOp::Clear(wgpu::Color {
                r: 0.2,
                g: 0.247,
                b: 0.314,
                a: 1.0,
            }),
            store: true,
        },
    ],
    //depth_stencil_attachment: None,
    depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
        view: &depth_view,
        depth_ops: Some(wgpu::Operations {
            load: wgpu::LoadOp::Clear(1.0),
            store: false,
        }),
        stencil_ops: None,
    }),
},
});
```

```
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..36, 0..NXY as u32);
}
```

```
self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
env_logger::init();
let event_loop = EventLoop::new();
let window = WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&format!("{}","ch12_cube_instance"));
let mut state = pollster::block_on(State::new(&window));

let render_start_time = std::time::Instant::now();

event_loop.run(move |event, _, control_flow| {
match event {
    Event::WindowEvent {
        ref event,
        window_id,
    } if window_id == window.id() => {
        if !state.input(event) {
            match event {
                WindowEvent::CloseRequested
                | WindowEvent::KeyboardInput {
                    input:
                        KeyboardInput {
                            state: ElementState::Pressed,
                            virtual_keycode: Some(VirtualKeyCode::Escape),
                            ..
                        },
                },
            }
        }
    }
})
}
}
```

```

        ..
    } => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}:", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
);
}
}

```

This code introduces three additional constants:  $NX$ ,  $NY$ , and  $NXY (= NX * NY)$ , meaning that we want to draw a total of  $NXY$  instances with  $NX$  columns and  $NY$  rows. You can change the number of instances and their arrangement by changing these three constants.

Inside the *State::new* method, we set the uniform buffer size large enough to hold transformations for every instance:

```

let uniform_buffer_size = NXY as u64 * matrix_size;

let uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Uniform Buffer"),
    size: uniform_buffer_size,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

```

The *update* method contains the key code block for our instanced drawing. Here, we define the model-view-projection matrix as a vector with the type *Vec<[f32; 16]>*:

```
let mut mvp_mat:Vec<[f32; 16]> = Vec::with_capacity(NXY as usize);
```

This vector has a length of  $NXY$ , i.e., the number of instances. Each element in this vector is a  $4 \times 4$  matrix that represents the model-view-projection matrix for each instance.

We then use a double for-loop to set the transformation matrix data for each instance. Here, we use translation and rotation transforms to specify the location and rotation for each instance. Next, we combine these transformation data with the view-projection data to construct the model-view-projection

## 300 | Practical GPU Graphics with wgpu and Rust

matrix for each instance, and store these data in the `mvp_mat` vector. We then write the data from the `mvp_mat` vector to our uniform buffer with the following code snippet:

```
let mvp_ref:&[[f32; 16]; NXY] = mvp_mat[...].try_into().unwrap();
self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(mvp_ref));
```

Finally, inside the `render` method, we call the draw function:

```
render_pass.draw(0..36, 0..NXY as u32);
```

Here, we specify the `instances` parameter with `0..NXY`, indicating that we want to draw a number of `NXY` instances of our cube.

### 12.2.2 Shader Code

We also need to change our shader code for our current example. Add a new `cube_instance.wgsl` shader file to the `examples/ch12/` folder with the following code:

```
// vertex shader
[[block]] struct Uniforms {
    mvpMatrix : [[stride(64)]] array<mat4x4<f32>, 35>;
};
[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;
struct Output {
    [[builtin(position)]] Position : vec4<f32>;
    [[location(0)]] vColor : vec4<f32>;
};
[[stage(vertex)]]
fn vs_main([[builtin(instance_index)]] instanceIdx : u32, [[location(0)]] pos: vec4<f32>,
[[location(1)]] color: vec4<f32>) -> Output {
    var output: Output;
    output.Position = uniforms.mvpMatrix[instanceIdx] * pos;
    output.vColor = color;
    return output;
}

// fragment shader
[[stage(fragment)]]
fn fs_main([[location(0)]] vColor: vec4<f32>) -> [[location(0)]] vec4<f32> {
    return vColor;
}
```

The `Uniforms` struct contains an array of model-view-projection matrices with an explicit element stride of 64 bytes because the 4x4 model-view-projection matrix for each instance takes 64 bytes. Note that the array element stride must be a multiple of the element alignment. Here, we manually set the length of this array to 35, representing the number of our instances. You can also pass the `NXY` constant to the shader using a uniform buffer to avoid setting this manually.

### 12.2.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch12_cube_instance"
path = "examples/ch12/cube_instance.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch12_cube_instance
```

This produces the results shown in Fig.12-2.



*Fig.12-2. Multiple cubes created with instancing.*

## 12.3 Creating Different Objects

In the preceding examples, we demonstrated how to create the same object multiple times using uniform transformations and instancing. However, in most cases, we will want to render multiple objects of different shapes. There is an easy way to create different objects if the environment is the same for all of the objects that we want to create. We can simply combine all of their respective vertex, color, normal, and UV data together and render them as a single object. This way, we can use the same code to render multiple objects that we used to render a single object.

### 12.3.1 Rust Code

This example is based on examples presented in Chapter 9. Specifically, we want to create four different objects, including *sinc*, *peaks*, *sphere*, and *torus* surfaces. We will use the same environment and shader program presented in Chapter 9 – the lighting effects and colormap will be included when rendering

## 302 | Practical GPU Graphics with wgpu and Rust

objects. This means that we need to combine the respective vertex, color, and normal data together. Note that these data are created as a `Vec<Vertex>` type.

Now, add a new Rust file called `different_objects.rs` to the `examples/ch12/` folder and type the following code into it:

```
use std::f32::consts::PI;
#[path="../ch09/common.rs"]
mod common;
#[path="../common/math_func.rs"]
mod math;

fn add_center(mut data:Vec<common::Vertex>, center:[f32;3]) -> Vec<common::Vertex>{
    for i in 0..data.len() {
        let p0 = data[i].position[0] + center[0];
        let p1 = data[i].position[1] + center[1];
        let p2 = data[i].position[2] + center[2];
        data[i].position = [p0, p1, p2, 1.0];
    }
    data
}

fn get_vertex_data() -> Vec<common::Vertex>{
    // create sinc surface
    let mut vertex_data = common::create_vertices(&math::sinc, "jet", -8.0, 8.0, -8.0,
        8.0, 30, 30, 1.2, 0.3);
    vertex_data = add_center(vertex_data, [-3.0,-1.0,0.0]);

    // create peaks surface
    let mut peaks_data = common::create_vertices(&math::peaks, "cool", -3.0, 3.0, -3.0,
        3.0, 31, 31, 1.0, 0.3);
    peaks_data = add_center(peaks_data, [3.0,-1.0,0.0]);

    // create sphere
    let mut sphere_data = common::create_vertices_param(&math::sphere, "hsv", 0.0, 2.0*PI,
        -PI/2.0, PI/2.0, 20, 15, -1.0, 1.0, -1.0, 1.0, 1.2, 0.0);
    sphere_data = add_center(sphere_data, [0.0,1.0,0.0]);

    // create torus
    let mut torus_data = common::create_vertices_param(&math::torus, "hot", 0.0, 2.0*PI,
        0.0, 2.0*PI, 40, 15, -1.0, 1.0, -1.0, 1.0, 1.0, 1.5);
    torus_data = add_center(torus_data, [1.0,-2.0,0.0]);

    // combine vertex data
    vertex_data.extend(peaks_data);
    vertex_data.extend(sphere_data);
    vertex_data.extend(torus_data);
    vertex_data
}

fn main(){
    let mut colormap_name = "jet";
    let mut is_two_side:i32 = 1;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        is_two_side = args[2].parse().unwrap();
    }
}
```

```

let vertex_data = get_vertex_data();
let light_data = common::light([1.0, 1.0, 1.0], 0.1, 0.8, 0.4, 30.0, is_two_side);
common::run(&vertex_data, light_data, colormap_name, "different_objects");
}

```

Here, we first introduce the *common* module from the *examples/ch09/* folder, and then create a function called *add\_center* which is used to specify the centers of different objects at different locations in order to properly display these objects in a single scene. Within the *add\_center* method, we add the center location to the *position* field of the *Vertex* struct. We then implement a *get\_vertex\_data* method used to generate the vertex data, color data, and normal data for four objects – a *sinc* surface, a *peaks* surface, a *sphere*, and a *torus*, and reset their center locations by calling the *add\_center* method. Finally, we combine the respective data together to form the final vertex data using the collection method *extend*.

The code looks very simple because we reuse the program and shader code from the *common.rs* file and *shader.wgsl* file in the *examples/ch09/* folder.

### 12.3.2 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

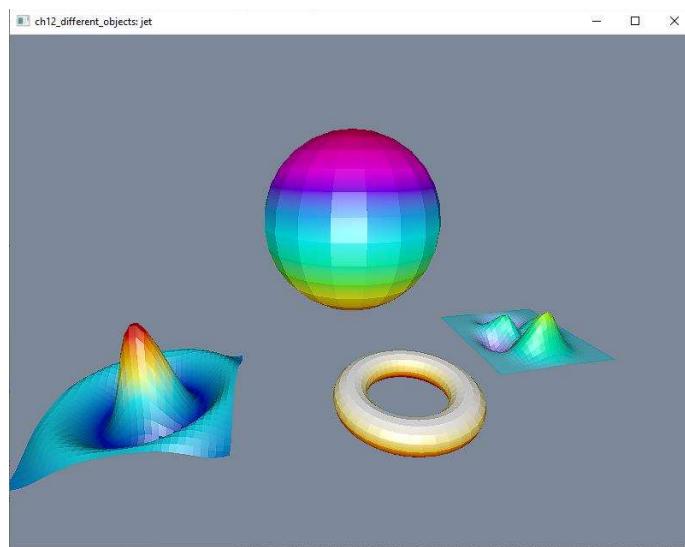
[[example]]
name = "ch12_different_objects"
path = "examples/ch12/different_objects.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch12_different_objects
```

This produces the results shown in Fig. 12-3.



*Fig. 12-3. Objects created by combining their position, color, and normal data.*

You can see from the figure that even though all of the objects are created under the same environment, you can still specify different colormaps and lighting effects for each individual object.

## 12.4 Creating Objects Using Multiple Pipelines

In this section, our task is to render multiple objects in a scene under different environments. Specifically, we want to render the cube with distinct face colors presented in Chapter 6 and the sphere with a texture map presented in Chapter 10 in the same scene. We know that the cube was created using vertex and color data and it does not have lighting and texture mapping, while the sphere was created using vertex, normal, and UV data and it does have lighting effects and texture mapping. Furthermore, both objects use very different shader programs. Thus, we cannot use the same approaches we used to create multiple objects in the preceding examples. Here, you will learn a general approach – using multiple pipelines to render multiple objects in a scene.

### 12.4.1 Rust Code

Add a new Rust file called `multiple_pipelines.rs` to the `examples/ch12/` folder and enter the following content into it:

```
#![allow(dead_code)]
use std:: {iter, mem };
use cgmath::{ Matrix, Matrix4, SquareMatrix };
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
    window::WindowBuilder,
};
use bytemuck:: {Pod, Zeroable, cast_slice};
#[path = "../common/transforms.rs"]
pub mod transforms;
#[path = "../common/surface_data.rs"]
mod surface;
#[path = "../common/texture_data.rs"]
mod texture;
#[path = "../common/vertex_data.rs"]
mod vertex_data;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

// for sphere
#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
    is_two_side: i32,
}

pub fn light(sc:[f32;3], ambient: f32, diffuse: f32, specular: f32, shininess: f32, two_side: i32) ->
Light {
    Light {
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ambient,
```

```

        diffuse_intensity: diffuse,
        specular_intensity: specular,
        specular_shininess: shininess,
        is_two_side: two_side,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
    pub uv: [f32; 2],
}
fn vertex(p:[f32;3], n:[f32; 3], t:[f32; 2]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
        uv: [t[0], t[1]],
    }
}
fn create_vertices() -> Vec<Vertex> {
    let(pos, normal, uv) = vertex_data::sphere_data(1.5, 30, 50);
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], uv[i]));
    }
    data.to_vec()
}
impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 3] = wgpu::vertex_attr_array![0=>Float32x4,
        1=>Float32x4, 2=>Float32x2];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::(<Vertex>)() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}
// for cube
#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
struct Vertex2 {
    position: [f32; 4],
    color: [f32; 4],
}
fn vertex2(p:[i8;3], c:[i8; 3]) -> Vertex2 {
    Vertex2 {
        position: [p[0] as f32, p[1] as f32, p[2] as f32, 1.0],
        color: [c[0] as f32, c[1] as f32, c[2] as f32, 1.0],
    }
}
fn create_vertices2() -> Vec<Vertex2> {

```

## 306 | Practical GPU Graphics with wgpu and Rust

```
let (pos, col, _uv, _normal) = vertex_data::cube_data();
let mut data:Vec<Vertex2> = Vec::with_capacity(pos.len());
for i in 0..pos.len() {
    data.push(vertex2(pos[i], col[i]));
}
data.to_vec()
}

impl Vertex2 {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex2>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

pub struct State {
    pub init: transforms::InitWgpu,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,

    // for sphere
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    num_vertices: u32,
    image_texture: texture::Texture,
    texture_bind_group: wgpu::BindGroup,
    num_vertices2: u32,
}

impl State {
    pub async fn new(window: &Window) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let light_data = light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
        let data = create_vertices();
        let data2 = create_vertices2();

        // the following code is for sphere:

        // create texture and texture bind group
        let image_texture = texture::Texture::create_texture_data(&init.device,
            &init.queue, "examples/ch10/assets/earth.png",
            wgpu::AddressMode::ClampToEdge, wgpu::AddressMode::ClampToEdge).unwrap();
        let texture_bind_group_layout = init.device.create_bind_group_layout(
            &wgpu::BindGroupLayoutDescriptor{
                entries: &[
                    wgpu::BindGroupLayoutEntry {
                        binding: 0,
```

```

        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Texture {
            multisampled: false,
            view_dimension: wgpu::TextureViewDimension::D2,
            sample_type: wgpu::TextureSampleType::Float { filterable: true },
        },
        count: None,
    },
    wgpu::BindGroupLayoutEntry {
        binding: 1,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Sampler {
            comparison: false,
            filtering: true,
        },
        count: None,
    },
],
label: Some("Texture Bind Group Layout"),
});

let texture_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &texture_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: wgpu::BindingResource::TextureView(&image_texture.view),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: wgpu::BindingResource::Sampler(&image_texture.sampler),
        },
    ],
    label: Some("Texture Bind Group"),
});
}

let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!(concat!(env!("CARGO_MANIFEST_DIR"),
        "/examples/ch10/shader.wgsl")).into()),
});
}

// uniform data for sphere
let camera_position = (2.5, 1.25, 2.5).into();
let look_direction = (0.0,0.0,0.0).into();
let up_direction = cgmath::Vector3::unit_y();

let (view_mat, project_mat, view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
        init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

// create vertex uniform buffer for sphere

// model_mat and view_projection_mat will be stored in vertex_uniform_buffer inside
// the update function
let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 192,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
})
}

```

## 308 | Practical GPU Graphics with wgpu and Rust

```
});

// create fragment uniform buffer. here we set eye_position =
// camera_position and light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 36,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 2,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
        ],
    }
);
```

```

        }
    ],
    label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout, &texture_bind_group_layout],
    push_constant_ranges: &[],
});
};

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleList,
        ..Default::default()
    },
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    })
});

```

## 310 | Practical GPU Graphics with wgpu and Rust

```
        }),
        multisample: wgpu::MultisampleState::default(),
    });

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = data.len() as u32;

// the following code is for cube

let shader2 = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!(concat!(env!("CARGO_MANIFEST_DIR"),
        "/examples/ch06/cube_face_color.wgsl")).into()),
});
let model_mat2 = transforms::create_transforms([1.0, 0.5, -2.0], [0.0, 0.0, 0.0], [0.7, 0.7, 0.7]);
let mvp_mat2 = view_project_mat * model_mat2;

let mvp_ref2:&[f32; 16] = mvp_mat2.as_ref();
let uniform_buffer2 = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Uniform Buffer 2"),
    contents: bytemuck::cast_slice(mvp_ref2),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let uniform_bind_group_layout2 = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor{
        entries: &[wgpu::BindGroupLayoutEntry {
            binding: 0,
            visibility: wgpu::ShaderStages::VERTEX,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Uniform,
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        }],
        label: Some("Uniform Bind Group Layout 2"),
    });
let uniform_bind_group2 = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout2,
    entries: &[wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer2.as_entire_binding(),
    }],
    label: Some("Uniform Bind Group 2"),
});

let pipeline_layout2 = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout 2"),
    bind_group_layouts: &[&uniform_bind_group_layout2],
    push_constant_ranges: &[],
});
let pipeline2 = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
```

```

label: Some("Render Pipeline 2"),
layout: Some(&pipeline_layout2),
vertex: wgpu::VertexState {
    module: &shader2,
    entry_point: "vs_main",
    buffers: &[Vertex2::desc()],
},
fragment: Some(wgpu::FragmentState {
    module: &shader2,
    entry_point: "fs_main",
    targets: &[wgpu::ColorTargetState {
        format: init.config.format,
        blend: Some(wgpu::BlendState {
            color: wgpu::BlendComponent::REPLACE,
            alpha: wgpu::BlendComponent::REPLACE,
        }),
        write_mask: wgpu::ColorWrites::ALL,
    }],
}),
primitive: wgpu::PrimitiveState{
    topology: wgpu::PrimitiveTopology::TriangleList,
    strip_index_format: None,
    cull_mode: Some(wgpu::Face::Back),
    ..Default::default()
},
depth_stencil: Some(wgpu::DepthStencilState {
    format: wgpu::TextureFormat::Depth24Plus,
    depth_write_enabled: true,
    depth_compare: wgpu::CompareFunction::LessEqual,
    stencil: wgpu::StencilState::default(),
    bias: wgpu::DepthBiasState::default(),
}),
multisample: wgpu::MultisampleState::default(),
});
};

let vertex_buffer2 = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer 2"),
    contents: cast_slice(&data2),
    usage: wgpu::BufferUsages::VERTEX,
});

let num_vertices2 = data2.len() as u32;

Self {
    init,
    view_mat,
    project_mat,

    // for sphere
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    num_vertices,
    image_texture,
    texture_bind_group,

    // for cube
    pipeline2,
    vertex_buffer2,
}

```

```

        uniform_bind_group2,
        uniform_buffer2,
        num_vertices2,
    }
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(
            new_size.width as f32 / new_size.height as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
pub fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

pub fn update(&mut self, dt: std::time::Duration) {
    let dt = ANIMATION_SPEED * dt.as_secs_f32();

    // update uniform buffer for sphere
    let model_mat = transforms::create_transforms([-2.5, -1.2, 0.5], [dt.sin(), dt.cos(), 0.0],
  [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;

    let normal_mat = (model_mat.invert().unwrap()).transpose();

    let model_ref:&[f32; 16] = model_mat.as_ref();
    let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
    let normal_ref:&[f32; 16] = normal_mat.as_ref();

    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
                                bytemuck::cast_slice(view_projection_ref));
    self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));

    // update uniform buffer for cube
    let model_mat2 = transforms::create_transforms([1.0, 0.5, -2.0], [0.0, dt.sin(), dt.cos()],
  [0.7, 0.7, 0.7]);
    let.mvp_mat2 = self.project_mat * self.view_mat * model_mat2;
    let mvp_ref2:&[f32; 16] = mvp_mat2.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer2, 0, bytemuck::cast_slice(mvp_ref2));
}

pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        }
    });
}

```

```

    },
    mip_level_count: 1,
    sample_count: 1,
    dimension: wgpu::TextureDimension::D2,
    format: wgpu::TextureFormat::Depth24Plus,
    usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
    label: None,
});
let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());
let mut encoder = self
    .init.device
    .create_command_encoder(&wgpu::CommandEncoderDescriptor {
        label: Some("Render Encoder"),
    });
{
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {
                    r: 0.2,
                    g: 0.247,
                    b: 0.314,
                    a: 1.0,
                }),
                store: true,
            },
        }],
        depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
            view: &depth_view,
            depth_ops: Some(wgpu::Operations {
                load: wgpu::LoadOp::Clear(1.0),
                store: false,
            }),
            stencil_ops: None,
        }),
    });
    // draw sphere
    render_pass.set_pipeline(&self.pipeline);
    render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
    render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
    render_pass.set_bind_group(1, &self.texture_bind_group, &[]);
    render_pass.draw(0..self.num_vertices, 0..1);

    // draw cube
    render_pass.set_pipeline(&self.pipeline2);
    render_pass.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
    render_pass.set_bind_group(0, &self.uniform_bind_group2, &[]);
    render_pass.draw(0..self.num_vertices2, 0..1);
}
self.init.queue.submit(iter::once(encoder.finish()));
output.present();
Ok(())
}

```

```

}

fn main() {
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("ch12_multiple_PIPELINES"));

    let mut state = pollster::block_on(State::new(&window));
    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                                ..
                        } => *control_flow = ControlFlow::Exit,
                        WindowEvent::Resized(physical_size) => {
                            state.resize(*physical_size);
                        }
                        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                            state.resize(**new_inner_size);
                        }
                        _ => {}
                    }
                }
            }
            Event::RedrawRequested(_) => {
                let now = std::time::Instant::now();
                let dt = now - render_start_time;
                state.update(dt);

                match state.render() {
                    Ok(_) => {}
                    Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
                    Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
                    Err(e) => eprintln!("{}:{}", e),
                }
            }
            Event::MainEventsCleared => {
                window.request_redraw();
            }
            _ => {}
        });
    });
}

```

As we did in creating our example with two cubes, here we define two sets of uniform matrices used to perform transformations on two different objects. We first construct two translations in the model matrices, which determine the locations of these two objects, and then create a respective uniform binding group with a corresponding uniform buffer and layout. Note that the *uniform\_bind\_group* for the sphere contains not only the *vertex\_uniform\_buffer*, but also the *fragment\_uniform\_buffer*, as well as the uniform sampler and uniform texture. Meanwhile, the *uniform\_bind\_groupd2* for the cube is much simpler – it contains only a vertex uniform buffer called *uniform\_buffer2*.

Next, we create two pipelines, *pipeline* and *pipeline2*, for the sphere and cube respectively. Note that these two pipelines have very different shader programs and *VertexState* attributes. Finally, we render the sphere and cube using the following code snippet:

```
// draw sphere
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.set_bind_group(1, &self.texture_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);

// draw cube
render_pass.set_pipeline(&self.pipeline2);
render_pass.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group2, &[]);
render_pass.draw(0..self.num_vertices2, 0..1);
```

Note how we set different pipelines when creating our sphere and our cube. Even though the *wgpu* API allows you to render multiple objects using different pipelines, these pipelines must be within a single render pass, which requires all of the pipelines to have the same *wgpu::RenderPassDescriptor*.

## 12.4.2 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch12_multiple_pipelines"
path = "examples/ch12/multiple_pipelines.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch12_multiple_pipelines
```

This produces the results shown in Fig.12-4.

## 12.5 Creating 3D Charts with Multiple Pipelines

In the preceding chapter, you learned how to create 3D surface charts with wireframes (or mesh lines) using a black square texture mapped onto each unit cell of the surface. The charts produced this way look nice but have one drawback – except for the white wireframe, none of the other colored wireframes display the true color of the square texture, because the wireframe color depends on the underlying surface colormap.

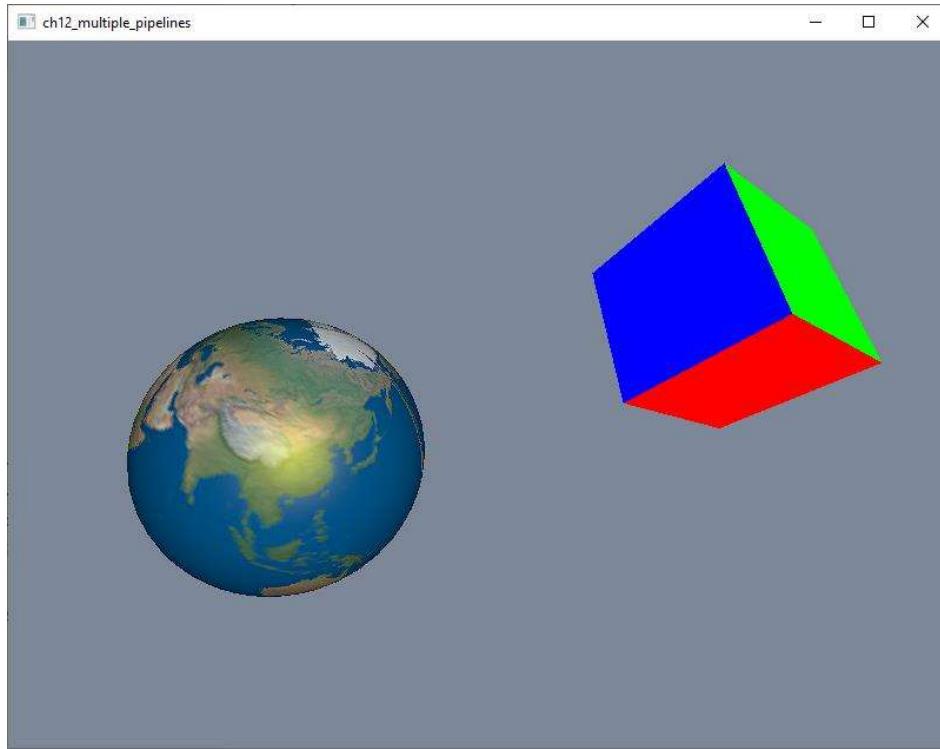


Fig.12-4. Objects created using different pipelines.

In order to avoid this issue, here I will explain how to use a multiple-pipeline approach to create 3D surface charts with wireframes that display the true color specified by the user. We will do it using two pipelines: one pipeline is used to create a 3D surface with a colormap using the *triangle-list* primitives, and the other pipeline is used to create a 3D wireframe for the surface using the *line-list* primitives. It is clear that we cannot use a single pipeline to do this because a single pipeline allows only one type of primitive specified by the *PrimitiveTopology* attribute.

In the following sections, I will use a sinc surface as an example to illustrate how to create a simple 3D surface with a corresponding wireframe.

### 12.5.1 Create Wireframe Data

Following the procedure presented in Chapter 7, we can easily create the wireframe data for a simple 3D surface. Add a new method called *simple\_mesh\_data* to the *surface\_data.rs* file in the *examples/common/* folder with the following code:

```
pub fn simple_mesh_data(f: &dyn Fn(f32, f32) -> [f32; 3], xmin:f32, xmax:f32, zmin:f32, zmax:f32,
    nx:usize, nz: usize, scale:f32, scalley:f32) -> Vec<[f32;3]> {
    let dx = (xmax-xmin)/(nx as f32-1.0);
    let dz = (zmax-zmin)/(nz as f32-1.0);
    let mut ymin1: f32 = 0.0;
    let mut ymax1: f32 = 0.0;

    let mut pts:Vec<Vec<[f32; 3]>> = vec![vec![Default::default(); nz]; nx];
    for i in 0..nx {
        let x = xmin + i as f32 * dx;
        let mut pt1:Vec<[f32; 3]> = Vec::with_capacity(nz);
        for j in 0..nz {
```

```

let z = zmin + j as f32 * dz;
let pt = f(x, z);
pt1.push(pt);
ymin1 = if pt[1] < ymin1 { pt[1] } else { ymin1 };
ymax1 = if pt[1] > ymax1 { pt[1] } else { ymax1 };
}
pts[i] = pt1;
}

let ymin = ymin1 - scaley * (ymax1 - ymin1);
let ymax = ymax1 + scaley * (ymax1 - ymin1);
for i in 0..nx {
    for j in 0..nz {
        pts[i][j] = normalize_point(pts[i][j], xmin, xmax, ymin, ymax, zmin, zmax, scale);
    }
}

let mut mesh: Vec<f32; 3> = Vec::with_capacity((4* (nx - 1)*(nz - 1)) as usize);

for i in 0..nx - 1 {
    for j in 0.. nz - 1 {
        let p0 = pts[i][j];
        let p1 = pts[i][j+1];
        let p2 = pts[i+1][j+1];
        let p3 = pts[i+1][j];

        // mesh data
        mesh.push(p0);
        mesh.push(p1);
        mesh.push(p0);
        mesh.push(p3);

        if i==nx-2 || j==nz-2 {
            mesh.push(p1);
            mesh.push(p2);
            mesh.push(p2);
            mesh.push(p3)
        }
    }
}
mesh
}

```

Note that we draw two line segments for the unit cell in most cases, except for the case at the edge when  $i = nx - 2$  or  $j = nz - 2$ , where we draw four line segments to make a perfect wireframe, since a simple 3D surface is an open surface.

## 12.5.2 Modify Shader Program

Our current example is based on the *sinc* example presented in Chapter 9, and its code and shader program will remain unchanged. In order to add a wireframe to the surface, we will need to make some changes to the code and shaders used in Chapter 7. Add a new file called *mesh.wgsl* to the *examples/ch12/* folder with the following code:

```

[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
    normal_mat : mat4x4<f32>;
}

```

```

    color: vec4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

[[stage(vertex)]]
fn vs_main([[location(0)]] pos: vec4<f32>) -> [[builtin(position)]] vec4<f32> {
    return uniforms.view_projection_mat * uniforms.model_mat * pos;
}

[[stage(fragment)]]
fn fs_main() -> [[location(0)]] vec4<f32> {
    return uniforms.color;
}

```

This code is essentially similar to that used to create wireframes in Chapter 7, except that the color of the wireframe can now be specified by the user. Another change is that here we pass three uniform matrices, *model\_mat*, *view\_projection\_mat*, and *normal\_mat*, to the shader instead of the single *model\_view\_projection\_mat* used in Chapter 7, which is necessary to match the uniform buffer used in creating the surface in this example.

### 12.5.3 Rust Code

Add a new Rust file called *chart\_pipelines.rs* to the *examples/ch12/* folder and type the following code into it:

```

#![allow(dead_code)]
use std:: {iter, mem };
use cgmath::{ Matrix, Matrix4, SquareMatrix };
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
    window::WindowBuilder,
};
use bytemuck:: {Pod, Zeroable, cast_slice};
#[path = "../common/transforms.rs"]
mod transforms;
#[path = "../common/surface_data.rs"]
mod surface;
#[path = "../common/math_func.rs"]
mod math;

const ANIMATION_SPEED:f32 = 1.0;
const IS_PERSPECTIVE:bool = true;

// for surface

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Light {
    specular_color : [f32; 4],
    ambient_intensity: f32,
    diffuse_intensity :f32,
    specular_intensity: f32,
    specular_shininess: f32,
    is_two_side: i32,
}

```

```

pub fn light(sc:[f32;3], ambient: f32, diffuse: f32, specular: f32, shininess: f32, two_side: i32) ->
Light {
    Light {
        specular_color: [sc[0], sc[1], sc[2], 1.0],
        ambient_intensity: ambient,
        diffuse_intensity: diffuse,
        specular_intensity: specular,
        specular_shininess: shininess,
        is_two_side: two_side,
    }
}

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex {
    pub position: [f32; 4],
    pub normal: [f32; 4],
    pub color: [f32; 4],
}

pub fn vertex(p:[f32;3], n:[f32; 3], c:[f32; 3]) -> Vertex {
    Vertex {
        position: [p[0], p[1], p[2], 1.0],
        normal: [n[0], n[1], n[2], 1.0],
        color: [c[0], c[1], c[2], 1.0],
    }
}

pub fn create_vertices(colormap_name: &str) -> Vec<Vertex> {
    let(pos, normal, color, _uv, _uv1) = surface::simple_surface_data(&math::sinc, colormap_name,
        -8.0, 8.0, -8.0, 8.0, 30, 30, 2.0, 0.3);
    let mut data:Vec<Vertex> = Vec::with_capacity(pos.len());
    for i in 0..pos.len() {
        data.push(vertex(pos[i], normal[i], color[i]));
    }
    data.to_vec()
}

impl Vertex {
    const ATTRIBUTES: [wgpu::VertexAttribute; 3] =
        wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4, 2=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

// for mesh

#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex2 {
    position: [f32; 4],
}

pub fn vertex2(p:[f32;3]) -> Vertex2 {

```

## 320 | Practical GPU Graphics with wgpu and Rust

```
Vertex2 {
    position: [p[0], p[1], p[2], 1.0],
}
}

pub fn create_vertices() -> Vec<Vertex2> {
    let mesh = surface::simple_mesh_data(&math::sinc, -8.0, 8.0, -8.0, 8.0, 30, 30, 2.0, 0.3);
    let mut data:Vec<Vertex2> = Vec::with_capacity(mesh.len());
    for i in 0..mesh.len() {
        data.push(vertex2(mesh[i]));
    }
    data.to_vec()
}

impl Vertex2 {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex2>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}

pub struct State {
    pub init: transforms::InitWgpu,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,

    // for surface
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group:wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    num_vertices: u32,

    // for mesh
    pipeline2: wgpu::RenderPipeline,
    vertex_buffer2: wgpu::Buffer,
    num_vertices2: u32,
    color: Vec<f32>,
}

impl State {
    pub async fn new(window: &Window, colormap_name: &str, color: Vec<f32>) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let light_data = light([1.0, 1.0, 0.0], 0.1, 0.8, 0.4, 30.0, 1);
        let data = create_vertices(colormap_name);
        let data2 = create_vertices2();

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!(concat!(env!("CARGO_MANIFEST_DIR"),
                "/examples/ch09/shader.wgsl")).into()),
        });
    });

    // uniform data
    let camera_position = (3.5, 1.75, 3.5).into();
}
```

```

let look_direction = (0.0,0.0,0.0).into();
let up_direction = cgmath::Vector3::unit_y();

let (view_mat, project_mat, _view_project_mat) =
    transforms::create_view_projection(camera_position, look_direction, up_direction,
    init.config.width as f32 / init.config.height as f32, IS_PERSPECTIVE);

// create vertex uniform buffer
// model_mat and view_projection_mat will be stored in vertex_uniform_buffer
// inside the update function
let vertex_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Vertex Uniform Buffer"),
    size: 208,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// create fragment uniform buffer. here we set eye_position = camera_position and
// light_position = eye_position
let fragment_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Fragment Uniform Buffer"),
    size: 32,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light and eye positions
let light_position:&[f32; 3] = camera_position.as_ref();
let eye_position:&[f32; 3] = camera_position.as_ref();
init.queue.write_buffer(&fragment_uniform_buffer, 0, bytemuck::cast_slice(light_position));
init.queue.write_buffer(&fragment_uniform_buffer, 16, bytemuck::cast_slice(eye_position));

// create light uniform buffer
let light_uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Light Uniform Buffer"),
    size: 36,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

// store light parameters
init.queue.write_buffer(&light_uniform_buffer, 0, bytemuck::cast_slice(&[light_data]));

let uniform_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX | wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {

```

## 322 | Practical GPU Graphics with wgpu and Rust

```
        ty: wgpu::BufferBindingType::Uniform,
        has_dynamic_offset: false,
        min_binding_size: None,
    },
    count: None,
},
wgpu::BindGroupLayoutEntry {
    binding: 2,
    visibility: wgpu::ShaderStages::FRAGMENT,
    ty: wgpu::BindingType::Buffer {
        ty: wgpu::BufferBindingType::Uniform,
        has_dynamic_offset: false,
        min_binding_size: None,
    },
    count: None,
}
],
label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: vertex_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: fragment_uniform_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: light_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[Vertex::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
```

```

        alpha: wgpu::BlendComponent::REPLACE,
    }),
    write_mask: wgpu::ColorWrites::ALL,
],
}),
primitive: wgpu::PrimitiveState{
    topology: wgpu::PrimitiveTopology::TriangleList,
    ..Default::default()
},
depth_stencil: Some(wgpu::DepthStencilState {
    format: wgpu::TextureFormat::Depth24Plus,
    depth_write_enabled: true,
    depth_compare: wgpu::CompareFunction::LessEqual,
    stencil: wgpu::StencilState::default(),
    bias: wgpu::DepthBiasState::default(),
}),
multisample: wgpu::MultisampleState::default(),
});
};

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: cast_slice(&data),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices = data.len() as u32;

// for mesh
let shader2 = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl(include_str!("mesh.wgsl").into()),
});
;

let pipeline2 = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline 2"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader2,
        entry_point: "vs_main",
        buffers: &[Vertex2::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader2,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::LineList,
        strip_index_format: None,
        ..Default::default()
    },
    depth_stencil: Some(wgpu::DepthStencilState {
        format: wgpu::TextureFormat::Depth24Plus,
        depth_write_enabled: true,
    })
});
;
```

```

        depth_compare: wgpu::CompareFunction::LessEqual,
        stencil: wgpu::StencilState::default(),
        bias: wgpu::DepthBiasState::default(),
    }),
    multisample: wgpu::MultisampleState::default(),
);
}

let vertex_buffer2 = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer 2"),
    contents: cast_slice(&data2),
    usage: wgpu::BufferUsages::VERTEX,
});
let num_vertices2 = data2.len() as u32;

Self {
    init,
    view_mat,
    project_mat,

    // for surface
    pipeline,
    vertex_buffer,
    uniform_bind_group,
    vertex_uniform_buffer,
    num_vertices,
}

// for mesh
pipeline2,
vertex_buffer2,
num_vertices2,
color
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
        self.project_mat = transforms::create_projection(new_size.width as f32 / new_size.height
            as f32, IS_PERSPECTIVE);
    }
}

#[allow(unused_variables)]
pub fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

pub fn update(&mut self, dt: std::time::Duration) {
    // update uniform buffer
    let dt = ANIMATION_SPEED * dt.as_secs_f32();
    let model_mat = transforms::create_transforms([0.0, 0.0, 0.0], [dt.sin(), dt.cos(), 0.0],
        [1.0, 1.0, 1.0]);
    let view_project_mat = self.project_mat * self.view_mat;

    let normal_mat = (model_mat.invert().unwrap()).transpose();

    let model_ref:&[f32; 16] = model_mat.as_ref();
}

```

```

let view_projection_ref:&[f32; 16] = view_project_mat.as_ref();
let normal_ref:&[f32; 16] = normal_mat.as_ref();
let color:[f32; 4] = [self.color[0], self.color[1], self.color[2], 1.0];

self.init.queue.write_buffer(&self.vertex_uniform_buffer, 0, bytemuck::cast_slice(model_ref));
self.init.queue.write_buffer(&self.vertex_uniform_buffer, 64,
    bytemuck::cast_slice(view_projection_ref));
self.init.queue.write_buffer(&self.vertex_uniform_buffer, 128, bytemuck::cast_slice(normal_ref));
self.init.queue.write_buffer(&self.vertex_uniform_buffer, 192, bytemuck::cast_slice(&color));
}

pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format:wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
                view: &depth_view,
                depth_ops: Some(wgpu::Operations {
                    load: wgpu::LoadOp::Clear(1.0),
                    store: false,
                }),
            }),
        });
    }
}

```

```

        stencil_ops: None,
    },
}),
});

// draw surface
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);

// draw mesh
render_pass.set_pipeline(&self.pipeline2);
render_pass.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
render_pass.draw(0..self.num_vertices2, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
    let mut colormap_name = "jet";
    let mut color = "1.0,1.0,1.0";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        colormap_name = &args[1];
    }
    if args.len() > 2 {
        color = &args[2];
    }
    let clr = color.split(",").filter_map(|s| s.parse::<f32>().ok()).collect::<Vec<_>>();

    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(*format!("ch12_chart_pipelines"));

    let mut state = pollster::block_on(State::new(&window, colormap_name, clr));
    let render_start_time = std::time::Instant::now();

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                            ..
                        } => *control_flow = ControlFlow::Exit,
                    }
                }
            }
        }
    });
}
}

```

```

        WindowEvent::Resized(physical_size) => {
            state.resize(*physical_size);
        }
        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
            state.resize(**new_inner_size);
        }
        _ => {}
    }
}
Event::RedrawRequested(_) => {
    let now = std::time::Instant::now();
    let dt = now - render_start_time;
    state.update(dt);

    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{:?}", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

Unlike the preceding *multiple-pipelines* example, where we created two sets of uniform matrices to perform transformations on two objects separately, our current example uses a single uniform transformation on both the surface and wireframe because we want them to move together as a single surface chart.

Next, we create two pipelines, *pipeline* and *pipeline2*, for the surface and wireframe respectively. These two pipelines have very different shader programs, primitive types, and *VertexState* attributes. Finally, we render the surface and wireframe using the following code snippet:

```

// draw surface
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);

// draw mesh
render_pass.set_pipeline(&self.pipeline2);
render_pass.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
render_pass.draw(0..self.num_vertices2, 0..1);

```

Note how we set different pipelines when creating the surface and the wireframe. Following the same procedure, you can easily create various 3D surface charts by specifying different math functions.

## 12.5.4 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
```

## 328 | Practical GPU Graphics with wgpu and Rust

```
name = "ch12_chart_pipelines"  
path = "examples/ch12/chart_pipelines.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch12_chart_pipelines
```

This produces the results shown in Fig.12-5.

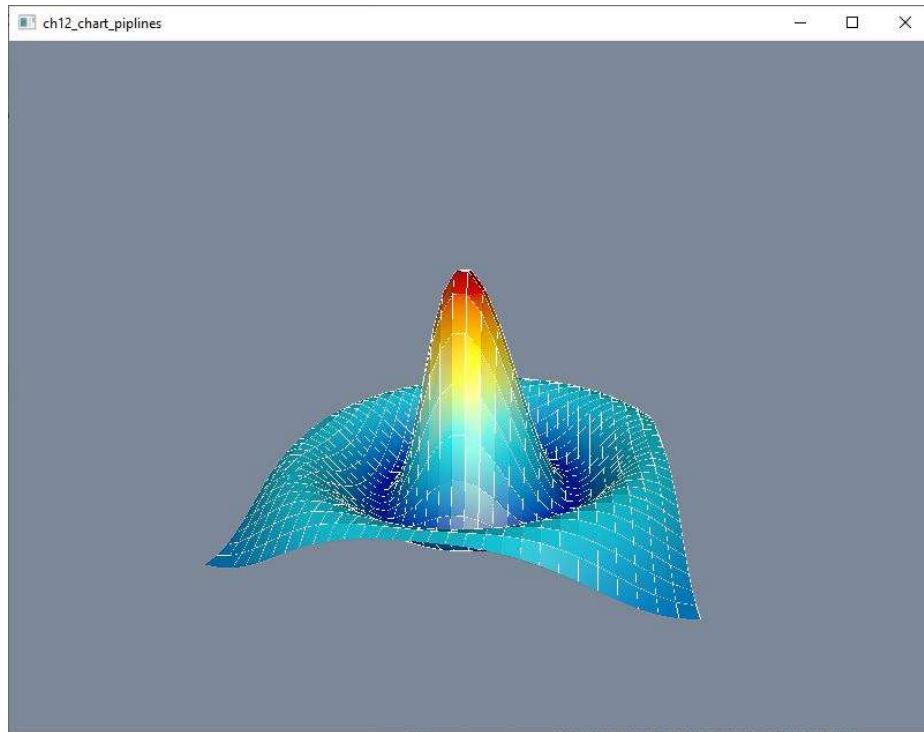
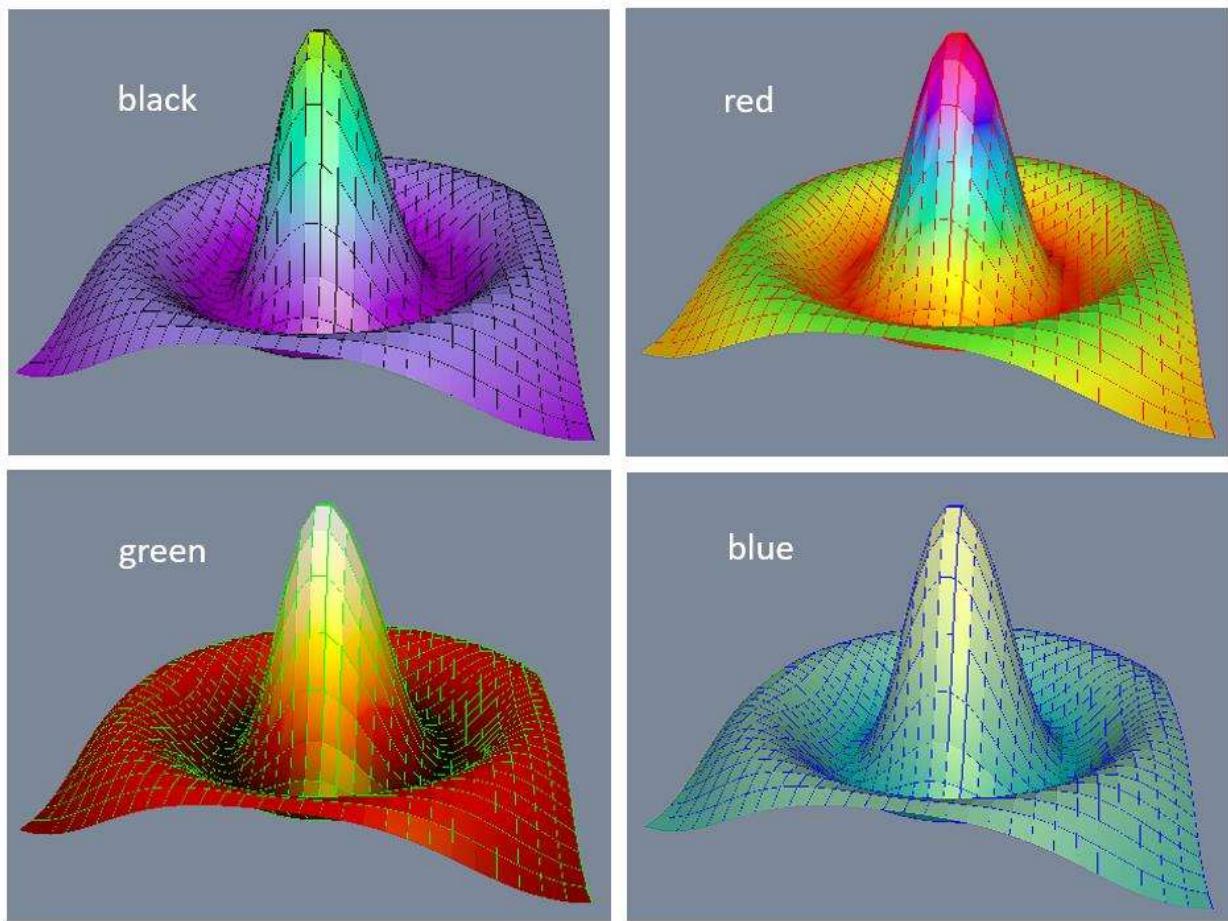


Fig.12-5. Surface chart with wireframe.

You can see that by default, our surface chart has a white wireframe. If you want, you can also change the colormap and wireframe color in the terminal window. Fig.12-6 shows the results of executing the following commands respectively:

```
cargo run --example ch12_chart_pipelines "cool" "0.0,0.0,0.0"  
cargo run --example ch12_chart_pipelines "hsv" "1.0,0.0,0.0"  
cargo run --example ch12_chart_pipelines "hot" "0.0,1.0,0.0"  
cargo run --example ch12_chart_pipelines "spring" "0.0,0.0,1.0"
```



*Fig. 12-6. 3D sinc charts with different wireframe colors.*

You can see from Fig. 12-6 that the wireframe has a pure color independent of the underlying surface colormap.

One issue associated with 3D surface charts created using the two-pipeline approach is that you cannot change the default thickness (1px) of the wireframe because the current *wgpu* API does not allow you to change the thickness of the line primitive.

## 12.6 Charts with Coordinate Axes

The coordinate axes are a basic element of any chart that we have yet to render in any of our example so far. However, we can easily add coordinate axes to our 3D charts using the multiple-pipeline approach presented in the preceding examples. In this section, you will learn how to add coordinate axes to the 3D surface chart with wireframe presented in the preceding sections.

### 12.6.1 Shaders for Coordinate Axes

We need to use new shader code to create the pipeline for coordinate axes. Add a new file called *axes.wgsl* to the *examples/ch12/* folder with the following code:

## 330 | Practical GPU Graphics with wgpu and Rust

```
[[block]] struct Uniforms {
    model_mat : mat4x4<f32>;
    view_project_mat : mat4x4<f32>;
};

[[binding(0), group(0)]] var<uniform> uniforms : Uniforms;

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_color : vec4<f32>;
};

[[stage(vertex)]]
fn vs_main([[location(0)]] pos: vec4<f32>, [[location(1)]] color: vec4<f32>) -> Output {
    var output:Output;
    output.position = uniforms.view_project_mat * uniforms.model_mat * pos;
    output.v_color = color;
    return output;
}

[[stage(fragment)]]
fn fs_main(in:Output) -> [[location(0)]] vec4<f32> {
    return in.v_color;
}
```

This code is very simple. It passes vertex and color data to the vertex shader and uses the vertex color in the fragment shader to render the coordinate axes.

### 12.6.2 Rust Code

The Rust code for this example is essentially the same as that used in the preceding example. Add a new Rust file called *chart\_axes.rs* to the *examples/ch12/* folder. Here I list only the code related to coordinate-axes. First, we need to define vertex data for the coordinate axes:

```
// for axes
#[repr(C)]
#[derive(Copy, Clone, Debug, Pod, Zeroable)]
pub struct Vertex3 {
    position: [f32; 4],
    color: [f32; 4],
}

pub fn vertex3(p:[f32;3], c:[f32;3]) -> Vertex3 {
    Vertex3 {
        position: [p[0], p[1], p[2], 1.0],
        color: [c[0], c[1], c[2], 1.0],
    }
}

pub fn create_vertices3(center:[f32;3], size:[f32; 3]) -> Vec<Vertex3> {
    let mut data:Vec<Vertex3> = Vec::with_capacity(6);
    data.push(vertex3([center[0], center[1], center[2]], [1.0, 0.0, 0.0]));
    data.push(vertex3([center[0] + size[0], center[1], center[2]], [1.0, 0.0, 0.0]));
    data.push(vertex3([center[0], center[1], center[2]], [0.0, 1.0, 0.0]));
    data.push(vertex3([center[0], center[1] + size[1], center[2]], [0.0, 1.0, 0.0]));
    data.push(vertex3([center[0], center[1], center[2]], [0.0, 0.0, 1.0]));
    data.push(vertex3([center[0], center[1], center[2] + size[2]], [0.0, 0.0, 1.0]));
    data.to_vec()
}
```

```
impl Vertex3 {
    const ATTRIBUTES: [wgpu::VertexAttribute; 2] = wgpu::vertex_attr_array![0=>Float32x4, 1=>Float32x4];
    fn desc<'a>() -> wgpu::VertexBufferLayout<'a> {
        wgpu::VertexBufferLayout {
            array_stride: mem::size_of::<Vertex3>() as wgpu::BufferAddress,
            step_mode: wgpu::VertexStepMode::Vertex,
            attributes: &Self::ATTRIBUTES,
        }
    }
}
```

Here, we define a *Vertex3* struct for our coordinate axes, which contains two fields: *position* and *color*. The *create\_vertices3* method is used to create vertex data for the coordinate axes. It accepts the *center* and *size* parameters as its input arguments. The *center* parameter lets you specify the center location of the axes, while the three components of the *size* parameter allow you to set the length of the coordinate axis along the *x*, *y*, and *z* directions. We define each axis with two points and set each one to a different color: red for the *x*-axis, green for the *y*-axis, and blue for the *z*-axis.

We then add three new fields to the *State* struct:

```
pub struct State {
    pub init: transforms::InitWgpu,
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,

    // for surface
    pipeline: wgpu::RenderPipeline,
    vertex_buffer: wgpu::Buffer,
    uniform_bind_group: wgpu::BindGroup,
    vertex_uniform_buffer: wgpu::Buffer,
    num_vertices: u32,

    // for mesh
    pipeline2: wgpu::RenderPipeline,
    vertex_buffer2: wgpu::Buffer,
    num_vertices2: u32,
    color: Vec<f32>,

    // for axes
    pipeline3: wgpu::RenderPipeline,
    vertex_buffer3: wgpu::Buffer,
    num_vertices3: u32,
}
```

Inside the *State::new* method, we generate the vertex data for our coordinate axes by calling the *create\_vertices3* method:

```
let data = create_vertices(colormap_name);
let data2 = create_vertices2();
let data3 = create_vertices3([0.0, -1.0, 0.0], [3.0, 3.0, 3.0]);
```

Here, we manually set the input arguments, *center* and *size*, for demonstration purpose, but you can also pass user-specified parameters instead. Next, we create the pipeline for our coordinate axes with the corresponding shader:

```
// for axes
let shader3 = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
```

## 332 | Practical GPU Graphics with wgpu and Rust

```
source: wgpu::ShaderSource::Wgsl(include_str!("axes.wgsl").into()),  
});  
  
let pipeline3 = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {  
    label: Some("Render Pipeline 3"),  
    layout: Some(&pipeline_layout),  
    vertex: wgpu::VertexState {  
        module: &shader3,  
        entry_point: "vs_main",  
        buffers: &[Vertex3::desc()],  
    },  
    fragment: Some(wgpu::FragmentState {  
        module: &shader3,  
        entry_point: "fs_main",  
        targets: &[wgpu::ColorTargetState {  
            format: init.config.format,  
            blend: Some(wgpu::BlendState {  
                color: wgpu::BlendComponent::REPLACE,  
                alpha: wgpu::BlendComponent::REPLACE,  
            }),  
            write_mask: wgpu::ColorWrites::ALL,  
        }],  
    }),  
    primitive: wgpu::PrimitiveState{  
        topology: wgpu::PrimitiveTopology::LineList,  
        strip_index_format: None,  
        ..Default::default()  
    },  
    depth_stencil: Some(wgpu::DepthStencilState {  
        format: wgpu::TextureFormat::Depth24Plus,  
        depth_write_enabled: true,  
        depth_compare: wgpu::CompareFunction::LessEqual,  
        stencil: wgpu::StencilState::default(),  
        bias: wgpu::DepthBiasState::default(),  
    }),  
    multisample: wgpu::MultisampleState::default(),  
});  
  
let vertex_buffer3 = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {  
    label: Some("Vertex Buffer 3"),  
    contents: cast_slice(&data3),  
    usage: wgpu::BufferUsages::VERTEX,  
});  
let num_vertices3 = data3.len() as u32;  
  
Self {  
    init,  
    view_mat,  
    project_mat,  
  
    // for surface  
    pipeline,  
    vertex_buffer,  
    uniform_bind_group,  
    vertex_uniform_buffer,  
    num_vertices,  
  
    // for mesh  
    pipeline2,  
    vertex_buffer2,
```

```

    num_vertices2,
    color,

    // for axes
    pipeline3,
    vertex_buffer3,
    num_vertices3,
}
}

```

Inside the render function, we render the surface, wireframe, and coordinate axes using the following code snippet:

```

// draw surface
render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);

// draw mesh
render_pass.set_pipeline(&self.pipeline2);
render_pass.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
render_pass.draw(0..self.num_vertices2, 0..1);

// draw axes
render_pass.set_pipeline(&self.pipeline3);
render_pass.set_vertex_buffer(0, self.vertex_buffer3.slice(..));
render_pass.draw(0..self.num_vertices3, 0..1);

```

Here, we create three pipelines: the first one is for the surface, the second one is for the wireframe, and the third one is for the coordinate axes. We also use a single uniform transformation on the surface, wireframe, and coordinate axes since we want them to move together as a single surface chart.

The code for the main function is identical to that used in the preceding example.

### 12.6.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch12_chart_axes"
path = "examples/ch12/chart_axes.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch12_chart_axes "cool" "0.0,0.0,0.0"
```

This produces the results shown in Fig.12-7.

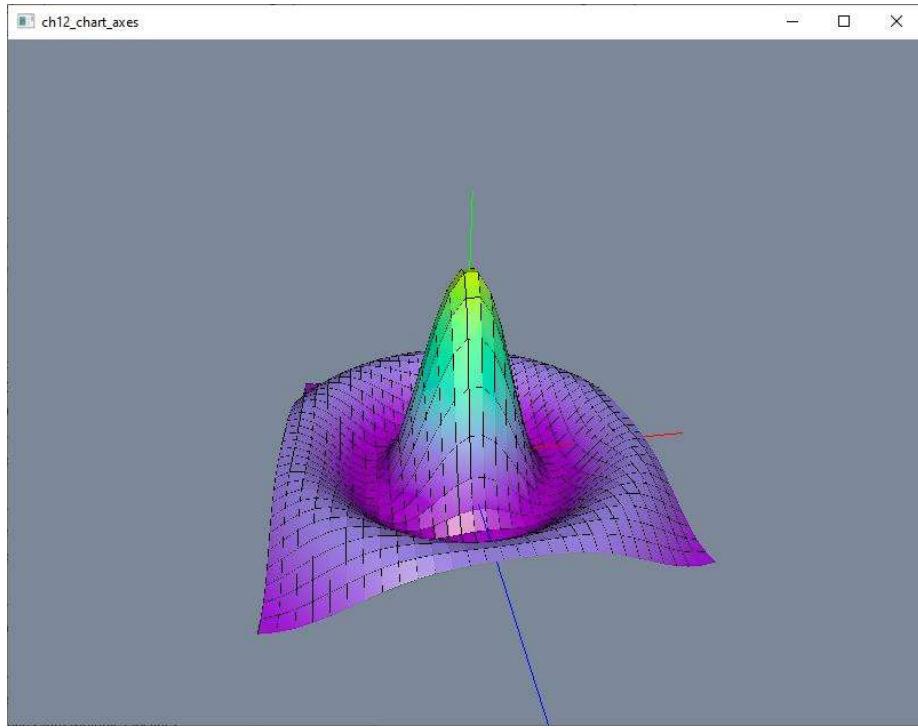


Fig.13-7. 3D surface chart with coordinate axes.

## 12.7 Creating 3D Charts with Multiple Render Passes

So far in our examples, we have used a single render pass to create multiple objects in a scene. In other words, we used a single command encoder to submit all of our draw calls to the GPU. But for more complex applications, we may need multiple render passes before presenting our images to the screen, using the results of one render pass in the next one. We may even need to render content off-screen for later use. For example, with multiple render passes, we can render a scene with multiple lights and shadows.

In this section, I will show you how to use two render passes to create a 3D *sinc* chart. As mentioned previously, in order to create a 3D surface, we need to specify the *depth\_stencil* attribute in the render pipeline and the *depth\_stencil\_attachment* in the render pass. In our *chart\_pipelines.rs* example, to make it possible to use a single render pass, we also added the *depth\_stencil* attribute to the pipeline for the wireframe. Previously, this is not necessary because we could set it to *None*, as we did in the *common.rs* file in the *examples/ch07/* folder. However, if we set this field to *None* in the *pipeline2*, we will get an “incompatible depth-stencil attachment” error. This means that all pipelines within a single render pass must have the same depth attachment. If we want to use pipelines with non-compatible attachments, we need to use multiple render passes.

### 12.7.1 Rust Code

The Rust code for this example is essentially the same as that used in the preceding *chart\_pipelines.rs* example. Add a new Rust file called *multiple\_render\_passes.rs* to the *examples/ch12/* folder. Here I will list only the changes to the code. First, we need to change the *depth\_stencil* attribute in *pipeline2* for the wireframe:

```
let pipeline2 = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline 2"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader2,
        entry_point: "vs_main",
        buffers: &[Vertex2::desc()],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader2,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::LineList,
        strip_index_format: None,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
```

Next, we need to make modifications to the *render* method with the following code:

```
pub fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let depth_texture = self.init.device.create_texture(&wgpu::TextureDescriptor {
        size: wgpu::Extent3d {
            width: self.init.config.width,
            height: self.init.config.height,
            depth_or_array_layers: 1,
        },
        mip_level_count: 1,
        sample_count: 1,
        dimension: wgpu::TextureDimension::D2,
        format: wgpu::TextureFormat::Depth24Plus,
        usage: wgpu::TextureUsages::RENDER_ATTACHMENT,
        label: None,
    });
    let depth_view = depth_texture.create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });
    {
```

```

// draw surface
let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
    label: Some("Render Pass 2"),
    color_attachments: &[wgpu::RenderPassColorAttachment {
        view: &view,
        resolve_target: None,
        ops: wgpu::Operations {
            load: wgpu::LoadOp::Clear(wgpu::Color {
                r: 0.2,
                g: 0.247,
                b: 0.314,
                a: 1.0,
            }),
            store: true,
        },
    }],
    depth_stencil_attachment: Some(wgpu::RenderPassDepthStencilAttachment {
        view: &depth_view,
        depth_ops: Some(wgpu::Operations {
            load: wgpu::LoadOp::Clear(1.0),
            store: false,
        }),
        stencil_ops: None,
    }),
});
```

```

render_pass.set_pipeline(&self.pipeline);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
render_pass.draw(0..self.num_vertices, 0..1);
}

{
    // draw mesh
    let mut render_pass2 = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Load,
                store: true,
            },
        }],
        depth_stencil_attachment: None,
    });

    render_pass2.set_pipeline(&self.pipeline2);
    render_pass2.set_vertex_buffer(0, self.vertex_buffer2.slice(..));
    render_pass2.set_bind_group(0, &self.uniform_bind_group, &[]);
    render_pass2.draw(0..self.num_vertices2, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}

```

Here, we use the `render_pass` method to draw the surface as before, but we use the `render_pass2` method without a `depth_stencil` to draw the wireframe. Note that when drawing the surface, we use

```
load: wgpu::LoadOp::Clear(wgpu::Color {
    r: 0.2,
    g: 0.247,
    b: 0.314,
    a: 1.0,
}),
```

inside the `color_attachment` field to clear the render target with a specified color before its content is loaded. But when drawing the wireframe, we use

```
load: wgpu::LoadOp::Load,
```

to load its content directly without clearing the render target. This means that we first draw our surface on the screen and then draw the wireframe on top of the surface.

## 12.7.2 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch12_multiple_render_passes"
path = "examples/ch12/multiple_render_passes.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch12_multiple_render_passes "hsv" "0.0,0.0,0.0"
```

This produces the results shown in Fig.12-8.

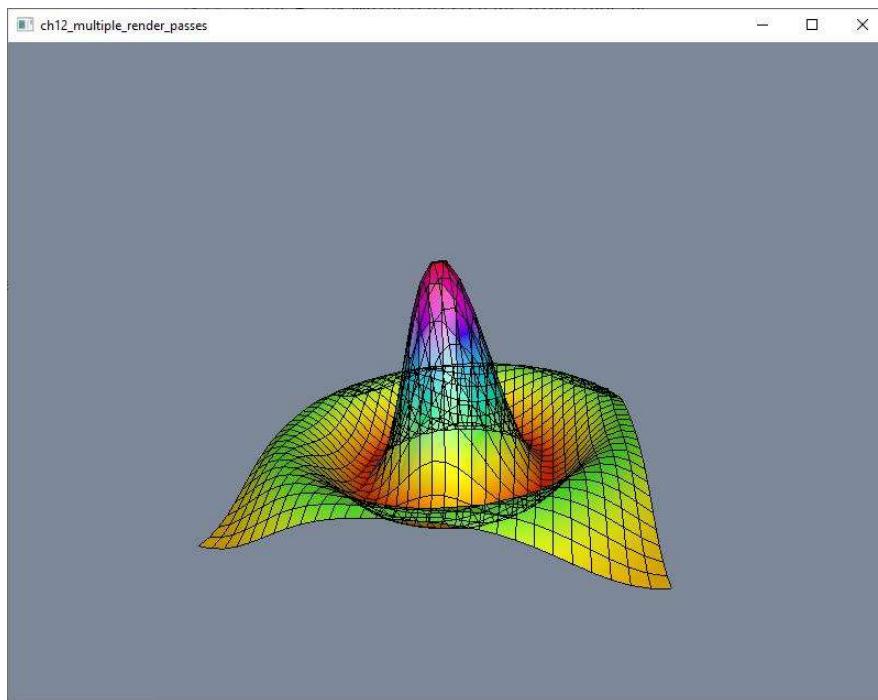


Fig.12-8. 3D sinc chart created with multiple render passes.



# 13 Compute Shaders and Particles

In the preceding chapters, we presented various approaches for rendering 3D objects graphically in a scene using both the vertex and fragment shaders. In fact, the *wgpu* API also exposes additional GPU capabilities not available in WebGL, such as compute shaders and storage buffers, which allow you to create powerful GPU compute applications to run on native devices and the web. In *wgpu*, the compute shader is separated from the rest of the graphics-rendering pipeline in its own stage of the GPU, where it can access the same resources as the graphics shaders while independently managing its own application flow. This is very useful for harnessing the power of the GPU for not just graphics computations but more general-purpose work as well.

This chapter introduces the compute shader and describes how to use it in a simple 2D rotation example. We then apply the compute shader to particle systems, which are collections of particles in which the particles follow a small set of predefined rules that determine values for their parameters based on time and other factors. The particles can move around, change size and color, appear, and disappear, and demonstrate all kinds of dynamic behaviors. By modifying the rules and the parameters that change, particle system can be used to create different visual effects like fireworks, rain, snow, flame, etc.

To simulate a particle system in CPU, we need to take all of the particles in every frame, and compute their parameters according to the rules. Then the CPU must send all of this updated information to the GPU, which renders the particles. This approach is simple and straightforward, but it becomes incredibly expensive for large numbers of particles. As the number of particles increases, both the workload on the CPU and the amount of data sent to the GPU every frame rises, until the computational cost is simply too high to be practical.

The other approach is to run the simulation directly on the GPU. The advantages are obvious: not only can we use the GPU's parallel processing architecture to calculate multiple particles simultaneously, but we also bypass the need to update data from the CPU every frame because the data is already available on the GPU. With a compute shader and *wgpu*, we can easily run a particle system simulation on GPU. This chapter will demonstrate three particle systems – one system mimics the flocking behavior of birds, one simulates the effect of gravity on particles, and the third one models particle kinematics.

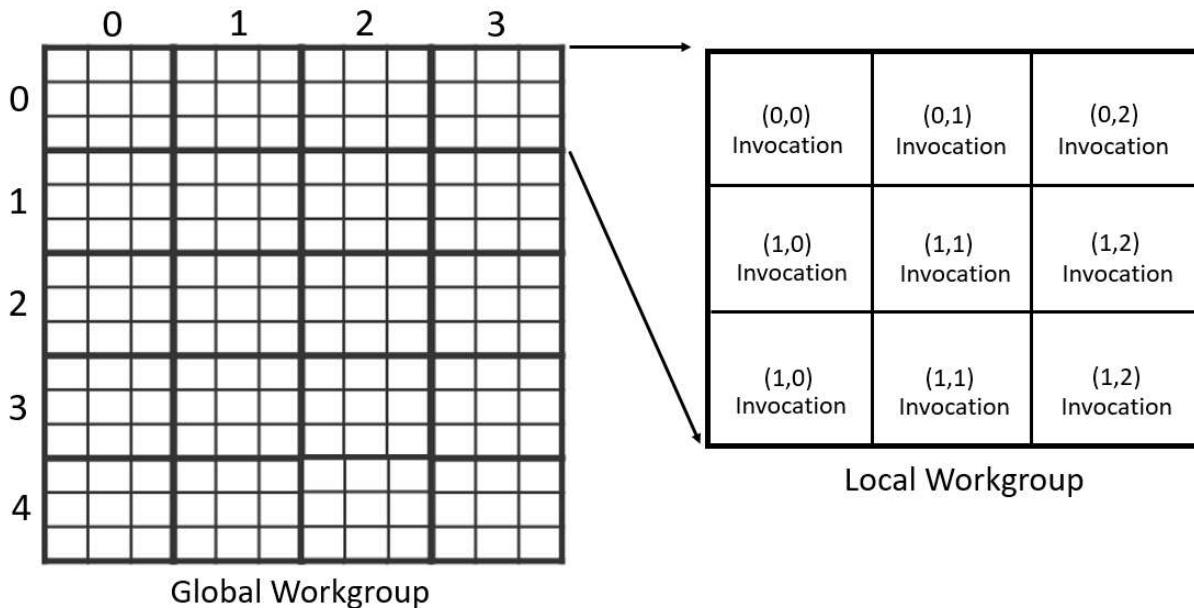
## 13.1 Compute Shader

Modern GPUs were developed to process enormous volumes of mathematical operations at the rapid rate needed to render complex 3D graphics in real-time for video games and similar uses. Today's high-end graphics processors are measured in teraflops, or trillions of calculations per second. This level of computational power would be useful for all sorts of applications, not just graphics processing. To allow

general-purpose applications to harness the power of the GPU, the *wgpu* API implements a special shader stage called the compute shader.

### 13.1.1 Compute Space and Workgroups

Like vertex and fragment shaders, which fit into the pipeline at specific points and operate on graphics-specific elements, compute shaders fit into the compute pipeline and operate on compute-specific elements. The GPU achieves its performance through parallelism, which in turn is obtained through work being launched in groups known as workgroups. Workgroups bundle together invocations into a set that concurrently executes and entry point function in a compute-shader-stage. Within a workgroup, the invocations share access to shader variables in the workgroup storage class. Groups of invocations form local neighborhoods known as local workgroups, which in turn are grouped into a global workgroup, as shown in Fig.13-1.



*Fig.13-1. Workgroup grid and invocations of a compute shader.*

You can see from the figure that the number of invocations of a compute shader is determined by the user-defined compute space. This space can be divided into a number of work groups. A global workgroup can then be divided into a number of local workgroups and each local workgroup contains a number of invocations. The compute space can be defined as a one-, two-, or three-dimensional space.

Since the invocations of the compute shader share variables and memory, they can communicate with each other and ensure their work remains coherent. Then, the workgroups, and the individual shader invocations they contain, can be executed in any order by the GPU.

In *wgpu*, the workgroup grid for a compute shader is the set of points with integer coordinates  $(i, j, k)$  with

```
0 <= i < workgroup_size_x, 0 <= j < workgroup_size_y, 0 <= k < workgroup_size_z,
```

Where  $(workgroup\_size_x, workgroup\_size_y, workgroup\_size_z)$  is the value specified for the *workgroup\_size* attribute of the entry point.

The `wgpu` API contains several built-in variables relating the workgroups and invocations of the compute shader:

- `local_invocation_id`: A `vec3<u32>` variable that indicates the current invocation's position in the workgroup grid, also known as its local invocation ID.
- `local_invocation_index`: A `u32` variable that represents a linearized index of the current local invocation.
- `global_invocation_id`: As `vec3<u32>` variable that indicates the current invocation's position in the compute shader grid, or compute space, also known as its global invocation ID.
- `workgroup_id`: A `vec3<u32>` variable that indicates the position of the current invocation's workgroup in the workgroup grid, also known as its workgroup ID.
- `workgroup_size`: A `vec3<u32>` variable that indicates the workgroup size of the current entry point.
- `num_workgroups`: A `vec<u32>` variable that indicates the dispatch size of the compute shader dispatched by the API.

With these built-in variables, you can easily access the workgroups and invocations of a compute shader in `wgpu`.

### 13.1.2 Write and Read Buffer

The `wgpu` API allows you to write data to memory for the GPU, a process that is not straightforward because of the sandboxing model used in modern web browsers. Let us look at the following code:

```
//create buffer to be mapped and an array for writing.
let numbers = [0.1, 1.0, 2.0];
let write_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Write Buffer"),
    contents: bytemuck::cast_slices(&numbers),
    usage: wgpu::BufferUsages::MAP_WRITE
});
```

This is the standard approach for creating a GPU buffer, which we have used extensively in preceding chapters. The only difference is the usage flag – here we use `wgpu::BufferUsages::MAP_WRITE`.

This example code demonstrates how to write 12 bytes to buffer memory. If you look at the source code of the `create_buffer_init` method closely, you will find that it creates a GPU buffer object mapped at creation. The associated raw binary data buffer can then be retrieved by calling the `get_mapped_range_mut` method.

A mapped buffer means that the buffer is owned by the CPU, and it is accessible to read/write from Rust code. To make it accessible to the GPU, it has to be unmapped by calling the `buffer.unmap` method. We need to specify whether a buffer is mapped or unmapped so that we can avoid race conditions where the CPU and GPU access memory at the same time.

Now that we have written a GPU buffer using the above code, we now want to copy the buffer to a second GPU buffer. This is done by simply replacing the original usage flag with the following code:

```
usage: wgpu::BufferUsages::MAP_WRITE | wgpu::BufferUsages::COPY_SRC
```

This way, we can use the buffer as the destination of the first GPU buffer and read it in Rust code once the GPU copy command has been executed. Here is the code list for this process:

```
//create a buffer to be mapped and an array for writing.
```

```

let numbers = [0.1, 1.0, 2.0];
let write_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Write Buffer"),
    contents: bytemuck::cast_slices(&numbers),
    usage: wgpu::BufferUsages::MAP_WRITE
});

//create a buffer for reading in an unmapped state.
let read_buffer = device.create_buffer(&wgpu::BufferDescriptor {
    label: None,
    size: result_buffer_size,
    usage: wgpu::BufferUsages::MAP_READ | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

```

Next, we use `device.create_command_encoder` to return the GPU command encoder, a Rust object containing buffered commands that will be submitted to the GPU as a batch. The methods on the GPU buffer are unbuffered, so they execute in sequence when called.

We then call `encoder.copy_buffer_to_buffer` on the GPU command encoder to add the buffered commands to the command queue so they can be executed later, as shown in the following code snippet:

```

//encode commands for copying buffer to buffer
let encoder = device.create_command_encoder();
encoder.copy_buffer_to_buffer(
    &write_buffer,           //source buffer
    0,                      //source offset
    &read_buffer,           //destination buffer
    0,                      //destination offset
    12                     //size
);

//submit commands
queue.submit(Some(encoder.finish()));

```

Now, we have to execute the GPU queue commands that we have sent. In order to read the second GPU buffer for execution, we first need to specify which portion of the buffer we are using for a given operation. Here, we will use the entire buffer with a total unbounded range. We then get the future representing when the buffer can be read by calling the `read_buffer_slice.map_async` method with `wgpu::MapMode::Read`:

```

let read_buffer_slice = read_buffer.slice(..);
let read_buffer_future = read_buffer_slice.map_async(wgpu::MapMode::Read);

```

This returns a promise that will resolve when the GPU buffer is mapped. We can then get the mapped range using the `read_buffer_slice.get_mapped_range` method, which contains the same values as the first GPU buffer once all of the queued GPU commands have been executed, as shown in the following code snippet:

```

//read buffer.
let data = read_buffer_slice.get_mapped_range();
let result = bytemuck::cast_slice(&data).to_vec();
drop(data);
read_buffer.unmap();

```

Note that we call the `drop(data)` method to make sure all mapped views are dropped before we unmap the buffer. You can see that in order to use GPU buffers in device queue submission, the buffers must be unmapped, while in order to write/read GPU buffers in Rust, they must be mapped.

## 13.2 2D Rotation in GPU

In this section, I will explain how to perform a simple 2D rotation directly in the GPU. The purpose of this example is not to examine the performance of parallel computations in the GPU, but to illustrate the procedure for using the compute shader in `wgpu` applications.

In matrix notation, a 2D rotation transform that rotates point  $(x_0, y_0)$  to  $(x_1, y_1)$  is given by

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} x_0 \cos \theta - y_0 \sin \theta \\ x_0 \sin \theta + y_0 \cos \theta \end{pmatrix}$$

Thus, the calculation we want to perform in the GPU is to find the final point  $(x_1, y_1)$  after rotating a given point  $(x_0, y_0)$  by a given rotation angle  $\theta$ .

### 13.2.1 Rust Code

Add a new sub-folder called `ch13` to the `examples/` folder, then add a new Rust file called `rotate2d.rs` to this newly created folder with the following code:

```
#![allow(dead_code)]
use wgpu::util::DeviceExt;

async fn run(point:Vec<f32>, angle:f32) -> Option<Vec<f32>> {
    let instance = wgpu::Instance::new(wgpu::Backends::all());
    let adapter = instance
        .request_adapter(&wgpu::RequestAdapterOptions::default())
        .await?;
    let (device, queue) = adapter
        .request_device(
            &wgpu::DeviceDescriptor {
                label: None,
                features: wgpu::Features::empty(),
                limits: wgpu::Limits::downlevel_defaults(),
            },
            None,
        )
        .await
        .unwrap();
    let shader = device.create_shader_module(&wgpu::ShaderModuleDescriptor {
        label: Some("Shader"),
        source: wgpu::ShaderSource::Wgsl(include_str!("rotate2d.wgsl").into()),
    });

    let point_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
        label: Some("Point Buffer"),
        contents: bytemuck::cast_slice(&[point[0], point[1]]),
        usage: wgpu::BufferUsages::STORAGE,
    });

    let angle_buffer = device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
        label: Some("Angle Buffer"),
        contents: bytemuck::cast_slice(&[angle]),
        usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::STORAGE,
    });

    let result_buffer_size:u64 = 8;
}
```

## 344 | Practical GPU Graphics with wgpu and Rust

```
let result_buffer = device.create_buffer(&wgpu::BufferDescriptor {
    label: None,
    size: result_buffer_size,
    usage: wgpu::BufferUsages::STORAGE | wgpu::BufferUsages::COPY_SRC,
    mapped_at_creation: false,
});

let read_buffer = device.create_buffer(&wgpu::BufferDescriptor {
    label: None,
    size: result_buffer_size,
    usage: wgpu::BufferUsages::MAP_READ | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

let bind_group_layout = device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
    entries: &[
        wgpu::BindGroupLayoutEntry {
            binding: 0,
            visibility: wgpu::ShaderStages::COMPUTE,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Storage {read_only: true},
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        },
        wgpu::BindGroupLayoutEntry {
            binding: 1,
            visibility: wgpu::ShaderStages::COMPUTE,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Uniform,
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        },
        wgpu::BindGroupLayoutEntry {
            binding: 2,
            visibility: wgpu::ShaderStages::COMPUTE,
            ty: wgpu::BindingType::Buffer {
                ty: wgpu::BufferBindingType::Storage {read_only: false},
                has_dynamic_offset: false,
                min_binding_size: None,
            },
            count: None,
        }
    ],
    label: Some("Bind Group Layout"),
});

let bind_group = device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: point_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: angle_buffer.as_entire_binding(),
        }
    ]
});
```

```

        },
        wgpu::BindGroupEntry {
            binding: 2,
            resource: result_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
);
};

let pipeline_layout = device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Pipeline Layout"),
    bind_group_layouts: &[&bind_group_layout],
    push_constant_ranges: &[],
});
;

let pipeline = device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
    label: Some("compute Pipeline"),
    layout: Some(&pipeline_layout),
    module: &shader,
    entry_point: "main",
});
;

let mut encoder = device.create_command_encoder(&wgpu::CommandEncoderDescriptor { label: None });
{
    let mut compute_pass = encoder.begin_compute_pass(&wgpu::ComputePassDescriptor { label: None });
    compute_pass.set_pipeline(&pipeline);
    compute_pass.set_bind_group(0, &bind_group, &[]);
    compute_pass.insert_debug_marker("compute collatz iterations");
    compute_pass.dispatch(2, 1, 1); //number of cells to run, the (x,y,z) size of item being processed
}
encoder.copy_buffer_to_buffer(&result_buffer, 0, &read_buffer, 0, result_buffer_size);
queue.submit(Some(encoder.finish()));

// read buffer
let read_buffer_slice = read_buffer.slice(..);
let read_buffer_future = read_buffer_slice.map_async(wgpu::MapMode::Read);
device.poll(wgpu::Maintain::Wait);
if let Ok(_) = read_buffer_future.await {
    // get buffer content
    let data = read_buffer_slice.get_mapped_range();
    let result = bytemuck::cast_slice(&data).to_vec();
    drop(data);
    read_buffer.unmap();
    println!("result = {:?}", result);
    Some(result)
} else {
    panic!("failed to run compute on gpu!")
}
}

fn main() {
    let mut point = "1.0,0.0";
    let mut angle = "45.0";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        point = &args[1];
    }
    if args.len() > 2 {
        angle = &args[2];
    }
}

```

```

let pt = point.split(",").filter_map(|s| s.parse::<f32>().ok()).collect::<Vec<_>>();
let agl = angle.parse::<f32>();

env_logger::init();
pollster::block_on(run(pt, agl.unwrap()));
}

```

In this example, we create three GPU buffers, *point\_buffer*, *angle\_buffer*, and *result\_buffer*. We store the initial point data in the *point\_buffer*, and store the rotation angle in the *angle\_buffer*, which is a uniform buffer. In practice, you can add any data to the *point\_buffer* and any parameters to the *angle\_buffer*. Note that our *result\_buffer* has the *STORAGE* and *COPY\_SRC* usage flags, meaning that we can write data to and read data from the buffer.

Next, we create the bind group and bind group layout, which are specific to *wgpu*. Remember that a bind group layout tells the shader which inputs correspond to which shader parameters, while a bind group stores the actual input and output data for the shader. In our example, the bind group contains the GPU buffers *point\_buffer*, *angle\_buffer*, and *result\_buffer*, which are assigned to the respective bindings 0, 1, 2 defined by the bind group layout.

With our three buffers setup by the bind group and bind group layout, we next create a compute pass encoder using the *encoder.begin\_compute\_pass* method. We set the encoder's bind group at index 0 with *compute\_pass.set\_bind\_group(0, &bind\_group)*, which corresponds to *group(0)* in the shader code.

In our example, the result is an array with two elements (*x1,y1*), so we call *compute\_pass.dispatch(2,1,1)* to encode the execution command. We call it “dispatching” when we encode a command to execute a compute pipeline on a set of data, analogous to “drawing” commands to execute a render pipeline. Here, the first argument *x* in the *dispatch* method is the first dimension, the second one *y* is the second dimension, and the latest one *z* is the third dimension. Both *y* and *z* are set to the default value of 1, while *x* will be the number of elements in our result array buffer.

Next, we create a GPU buffer called *read\_buffer* as a destination for copying the result buffer using the *copy\_buffer\_to\_buffer* method. Finally, we read the result by calling *read\_buffer\_slice.map\_async* with *wgpu::MapMode::Read* and waiting for it to indicate that the GPU buffer is mapped, at which point we can get mapped range with the *read\_buffer\_slice.get\_mapped\_range* method.

### 13.2.2 Shader Code

Add a new shader file called *rotate2d.wgsl* to the *examples/ch13/* folder with the following code:

```

[[block]] struct Data {
    numbers: [[stride(4)]] array<f32>;
};

[[block]] struct AngleData {
    angle: f32;
};

[[binding(0), group(0)]] var<storage, read> point_data : Data;
[[binding(1), group(0)]] var<uniform> angle_data: AngleData;
[[binding(2), group(0)]] var<storage, read_write> result: Data;

[[stage(compute), workgroup_size(1)]]
fn main([[builtin(global_invocation_id)]] global_id : vec3<u32>) {
    var index:u32 = global_id.x;
    var pt:vec2<f32> = normalize(vec2<f32>(point_data.numbers[0], point_data.numbers[1]));
    var p0:f32 = pt[0];

```

```

var p1:f32 = pt[1];
var theta:f32 = angle_data.angle*3.1415926/180.0;
var res:f32 = 0.0;
if(index == 0u) {
    res = p0 * cos(theta) - p1 * sin(theta);
} else {
    res = p0 * sin(theta) + p1 * cos(theta);
}
result.numbers[index] = res;
}

```

Note how we use the built in *global\_invocation\_id* variable to access the data in the buffers. Here, we first normalize the input point, meaning that our GPU code always returns a normalized point after rotation regardless if the input point is normalized or not. Following the procedure presented here, you can perform various computations directly in the GPU.

### 13.2.3 Run Application

Now, add the following code snippet to the *Cargo.toml* file:

```

[[example]]
name = "ch13_rotate2d"
path = "examples/ch13/rotate2d.rs"

```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch13_rotate2d
```

This will produce the following result in your terminal window:

```
result = [0.70710677, 0.7071068]
```

If we change the initial point to [1, 1] and rotation angle to 30 degrees:

```
cargo run --example ch13_rotate2d "1,1" "30"
```

we will get the following result:

```
result = [0.25881904, 0.96592575]
```

## 13.3 Compute Boids

Boids (short for “bird-oid objects”) is an artificial life program, originally developed by Craig Reynolds in 1986 that simulates the flocking behavior of birds. Like most artificial life simulations, Boids demonstrates emergent behavior, or, the complexity arising from the interaction of individual birds following a set of simple rules. The example presented in the following subsections is based on the WebGPU sample created by Austin Eng (please see his original sample at <https://github.com/austinEng/webgpu-samples/tree/main/src/sample/computeBoids>).

### 13.3.1 Rust Code

Add a new Rust file called *boids.rs* to the *examples/ch13/* folder and enter the following content into it:

```

#![allow(dead_code)]
use std:: {iter, mem };
use wgpu::util::DeviceExt;

```

## 348 | Practical GPU Graphics with wgpu and Rust

```
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
    window::WindowBuilder,
};

use rand::{
    distributions::{Distribution, Uniform},
    SeedableRng,
};
#[path = "../common/transforms.rs"]
mod transforms;

const NUM_PARTICLES: u32 = 5000;
const PARTICLES_PER_GROUP: u32 = 64;

struct State {
    init: transforms::InitWgpu,
    particle_bind_groups: Vec<wgpu::BindGroup>,
    particle_buffers: Vec<wgpu::Buffer>,
    vertices_buffer: wgpu::Buffer,
    compute_pipeline: wgpu::ComputePipeline,
    render_pipeline: wgpu::RenderPipeline,
    render_bind_group: wgpu::BindGroup,
    work_group_count: u32,
    frame_num: usize,
}

impl State {
    fn required_limits() -> wgpu::Limits {
        wgpu::Limits::downlevel_defaults()
    }

    fn required_downlevel_capabilities() -> wgpu::DownlevelCapabilities {
        wgpu::DownlevelCapabilities {
            flags: wgpu::DownlevelFlags::COMPUTE_SHADERS,
            ..Default::default()
        }
    }
}

async fn new(window: &Window, color_scale: f32) -> Self {
    let init = transforms::InitWgpu::init_wgpu(window).await;

    let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
        label: Some("Shader"),
        source: wgpu::ShaderSource::Wgsl(include_str!("boids.wgsl").into()),
    });

    let param_data = [
        0.04f32, // deltaT
        0.1, // rule1Distance
        0.025, // rule2Distance
        0.025, // rule3Distance
        0.02, // rule1Scale
        0.05, // rule2Scale
        0.005, // rule3Scale
    ]
    .to_vec();

    let color_scale_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
```

```

label: Some("Color Scale Buffer"),
contents: bytemuck::cast_slice(&[color_scale]),
usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let param_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Parameter Buffer"),
    contents: bytemuck::cast_slice(&param_data),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let compute_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: wgpu::BufferSize::new(
                        (param_data.len() * mem::size_of::<f32>()) as _,
                    ),
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: true },
                    has_dynamic_offset: false,
                    min_binding_size: wgpu::BufferSize::new((NUM_PARTICLES * 16) as _),
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 2,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: false },
                    has_dynamic_offset: false,
                    min_binding_size: wgpu::BufferSize::new((NUM_PARTICLES * 16) as _),
                },
                count: None,
            },
        ],
        label: None,
    });
}

let compute_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("compute"),
    bind_group_layouts: &[&compute_bind_group_layout],
    push_constant_ranges: &[],
});

let render_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {

```

## 350 | Practical GPU Graphics with wgpu and Rust

```
        binding: 0,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: wgpu::BufferSize::new(
                ([color_scale].len() * mem::size_of::<f32>()) as _,
            ),
        },
        count: None,
    },
],
label: None,
});

let render_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &render_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: color_scale_buffer.as_entire_binding(),
        },
    ],
    label: Some("Render Bind Group"),
});

let render_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("render"),
    bind_group_layouts: &[&render_bind_group_layout],
    push_constant_ranges: &[],
});

let render_pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: None,
    layout: Some(&render_pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[
            wgpu::VertexBufferLayout {
                array_stride: 4 * 4,
                step_mode: wgpu::VertexStepMode::Instance,
                attributes: &wgpu::vertex_attr_array![0 => Float32x2, 1 => Float32x2],
            },
            wgpu::VertexBufferLayout {
                array_stride: 2 * 4,
                step_mode: wgpu::VertexStepMode::Vertex,
                attributes: &wgpu::vertex_attr_array![2 => Float32x2],
            },
        ],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[init.config.format.into()],
    }),
    primitive: wgpu::PrimitiveState::default(),
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
});
```

```

let compute_pipeline = init.device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
    label: Some("Compute Pipeline"),
    layout: Some(&compute_pipeline_layout),
    module: &shader,
    entry_point: "cs_main",
});
};

let vertex_data = [-0.01f32, -0.02, 0.01, -0.02, 0.00, 0.02];
let vertices_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some("Vertex Buffer"),
    contents: bytemuck::bytes_of(&vertex_data),
    usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::COPY_DST,
});
};

let mut initial_particle_data = vec![0.0f32; (4 * NUM_PARTICLES) as usize];
let mut rng = rand::rngs::StdRng::seed_from_u64(42);
let unif = Uniform::new_inclusive(-1.0, 1.0);
for particle_instance_chunk in initial_particle_data.chunks_mut(4) {
    particle_instance_chunk[0] = unif.sample(&mut rng); // posx
    particle_instance_chunk[1] = unif.sample(&mut rng); // posy
    particle_instance_chunk[2] = unif.sample(&mut rng) * 0.1; // velx
    particle_instance_chunk[3] = unif.sample(&mut rng) * 0.1; // vely
}

let mut particle_buffers = Vec::<wgpu::Buffer>::new();
let mut particle_bind_groups = Vec::<wgpu::BindGroup>::new();
for i in 0..2 {
    particle_buffers.push(
        init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some(&format!("Particle Buffer {}", i)),
            contents: bytemuck::cast_slice(&initial_particle_data),
            usage: wgpu::BufferUsages::VERTEX
                | wgpu::BufferUsages::STORAGE
                | wgpu::BufferUsages::COPY_DST,
        }),
    );
}

for i in 0..2 {
    particle_bind_groups.push(init.device.create_bind_group(&wgpu::BindGroupDescriptor {
        layout: &compute_bind_group_layout,
        entries: &[
            wgpu::BindGroupEntry {
                binding: 0,
                resource: param_buffer.as_entire_binding(),
            },
            wgpu::BindGroupEntry {
                binding: 1,
                resource: particle_buffers[i%2].as_entire_binding(),
            },
            wgpu::BindGroupEntry {
                binding: 2,
                resource: particle_buffers[(i + 1) % 2].as_entire_binding(),
            },
        ],
        label: None,
    }));
}
let work_group_count = ((NUM_PARTICLES as f32) / (PARTICLES_PER_GROUP as f32)).ceil() as u32;

```

```

Self {
    init,
    particle_bind_groups,
    particle_buffers,
    vertices_buffer,
    compute_pipeline,
    render_pipeline,
    render_bind_group,
    work_group_count,
    frame_num: 0,
}
}

fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {
    // empty
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self.init.device.create_command_encoder(&wgpu::CommandEncoderDescriptor {
        label: Some("Render Encoder"),
    });

    {
        // compute pass
        let mut compute_pass = encoder.begin_compute_pass(
            &wgpu::ComputePassDescriptor { label: None });
        compute_pass.set_pipeline(&self.compute_pipeline);
        compute_pass.set_bind_group(0, &self.particle_bind_groups[self.frame_num % 2], &[]);
        compute_pass.dispatch(self.work_group_count, 1, 1);
    }
    {
        // render pass
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.5,

```

```

        g: 0.5,
        b: 0.5,
        a: 1.0,
    }),
    store: true,
},
],
depth_stencil_attachment: None,
});

render_pass.set_pipeline(&self.render_pipeline);
render_pass.set_vertex_buffer(0,
    self.particle_buffers[(self.frame_num + 1) % 2].slice(..));
render_pass.set_vertex_buffer(1, self.vertices_buffer.slice(..));
render_pass.set_bind_group(0, &self.render_bind_group, &[]);
render_pass.draw(0..3, 0..NUM_PARTICLES);
}
self.frame_num += 1;

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
let mut color_scale = "0.1";
let args: Vec<String> = std::env::args().collect();
if args.len() > 1 {
    color_scale = &args[1];
}
let clr = color_scale.parse::<f32>();

env_logger::init();
let event_loop = EventLoop::new();
let window = WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(*format!("ch13_boids"));

let mut state = pollster::block_on(State::new(&window, clr.unwrap()));

event_loop.run(move |event, _, control_flow| {
    match event {
        Event::WindowEvent {
            ref event,
            window_id,
        } if window_id == window.id() => {
            if !state.input(event) {
                match event {
                    WindowEvent::CloseRequested
                    | WindowEvent::KeyboardInput {
                        input:
                            KeyboardInput {
                                state: ElementState::Pressed,
                                virtual_keycode: Some(VirtualKeyCode::Escape),
                                ..
                            },
                    ..
                },
            } => *control_flow = ControlFlow::Exit,
            WindowEvent::Resized(physical_size) => {

```

```

        state.resize(*physical_size);
    }
    WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
        state.resize(**new_inner_size);
    }
    _ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!(":{}:", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

This code simulates each bird with a small triangle specified by the `vertex_data`. The `param_data` contains rules that describe the behavior of birds. You can see that we create two sets of particle buffers using the same initial particle data. In creating the bind group for the compute pipeline, we set `particle_buffers[i%2]` at binding 1 for reading and `particle_buffers[(i + 1)%2]` at binding 2 for writing, ensuring that the buffer used for writing is always different from that used for reading.

The `render_bind_group` contains a uniform buffer called `color_scale_buffer`, which is used to set the color for the triangles in the fragment shader. For the render pipeline, our vertex buffers contain two sets of data: one is the particle buffer and the other is the vertex data that describes each bird's shape (i.e., a triangle). This means that our rendering pipeline renders not only the shape for a single bird, but also all birds that are described by the positions and velocities included in the particle buffer.

### 13.3.2 Shader Code

The shader program in this example is more complex because it contains three types of shaders: vertex, fragment, and compute.

Add a new shader file called `boids.wgsl` to the `examples/ch13/` folder with the following code:

```

struct Input {
    [[location(0)]] a_particle_pos : vec2<f32>;
    [[location(1)]] a_particle_vel : vec2<f32>;
    [[location(2)]] a_pos : vec2<f32>;
};

struct Output {
    [[builtin(position)]] position : vec4<f32>;
    [[location(0)]] v_vel:vec2<f32>;
};

[[stage(vertex)]]

```

```

fn vs_main(input: Input) -> Output {
    var output: Output;
    var angle : f32 = -atan2(input.a_particle_vel.x, input.a_particle_vel.y);
    var pos : vec2<f32> = vec2<f32>(
        (input.a_pos.x * cos(angle)) - (input.a_pos.y * sin(angle)),
        (input.a_pos.x * sin(angle)) + (input.a_pos.y * cos(angle)));
    output.position = vec4<f32>(pos + input.a_particle_pos, 0.0, 1.0);
    output.v_vel = input.a_particle_vel;
    return output;
}

[[block]] struct Uniforms {
    colorScale: f32;
};

[[binding(0), group(0)]] var<uniform> param : Uniforms;

[[stage(fragment)]]
fn fs_main([[location(0)]] v_vel: vec2<f32>) -> [[location(0)]] vec4<f32> {
    let pi:f32 = 3.1415926;
    let c:f32 = param.colorScale;
    return vec4<f32>(c + (1.0-c)*sin(2.0*pi*v_vel.x), c + (1.0-c)*sin(2.0*pi*v_vel.y),
                    c + (1.0-c)*cos(pi*(v_vel.x - v_vel.y)), 1.0);
}

struct Particle {
    pos : vec2<f32>;
    vel : vec2<f32>;
};

[[block]] struct SimParams {
    deltaT : f32;
    rule1Distance : f32;
    rule2Distance : f32;
    rule3Distance : f32;
    rule1Scale : f32;
    rule2Scale : f32;
    rule3Scale : f32;
};

[[block]] struct Particles {
    particles : [[stride(16)]] array<Particle>;
};

[[binding(0), group(0)]] var<uniform> params : SimParams;
[[binding(1), group(0)]] var<storage, read> particlesA : Particles;
[[binding(2), group(0)]] var<storage, read_write> particlesB : Particles;

[[stage(compute), workgroup_size(64)]]
fn cs_main([[builtin(global_invocation_id)]] GlobalInvocationID : vec3<u32>) {
    let total = arrayLength(&particlesA.particles);
    var index : u32 = GlobalInvocationID.x;
    if (index >= total) {
        return;
    }
    var vPos : vec2<f32> = particlesA.particles[index].pos;
    var vVel : vec2<f32> = particlesA.particles[index].vel;
    var cMass : vec2<f32> = vec2<f32>(0.0, 0.0);
    var cVel : vec2<f32> = vec2<f32>(0.0, 0.0);
    var colVel : vec2<f32> = vec2<f32>(0.0, 0.0);
    var cMassCount : u32 = 0u;
    var cVelCount : u32 = 0u;
    var pos : vec2<f32>;
    var vel : vec2<f32>;
}

```

## 356 | Practical GPU Graphics with wgpu and Rust

```
for (var i : u32 = 0u; i < total; i = i + 1u) {
    if (i == index) {
        continue;
    }
    pos = particlesA.particles[i].pos.xy;
    vel = particlesA.particles[i].vel.xy;
    if (distance(pos, vPos) < params.rule1Distance) {
        cMass = cMass + pos;
        cMassCount = cMassCount + 1u;
    }
    if (distance(pos, vPos) < params.rule2Distance) {
        colVel = colVel - (pos - vPos);
    }
    if (distance(pos, vPos) < params.rule3Distance) {
        cVel = cVel + vel;
        cVelCount = cVelCount + 1u;
    }
}
if (cMassCount > 0u) {
    var temp : f32 = f32(cMassCount);
    cMass = (cMass / vec2<f32>(temp, temp)) - vPos;
}
if (cVelCount > 0u) {
    var temp : f32 = f32(cVelCount);
    cVel = cVel / vec2<f32>(temp, temp);
}
vVel = vVel + (cMass * params.rule1Scale) + (colVel * params.rule2Scale) + (
    cVel * params.rule3Scale);
vVel = normalize(vVel) * clamp(length(vVel), 0.0, 0.1);
// kinematic update
vPos = vPos + (vVel * params.deltaTime);
// Wrap around boundary
if (vPos.x < -1.0) {
    vPos.x = 1.0;
}
if (vPos.x > 1.0) {
    vPos.x = -1.0;
}
if (vPos.y < -1.0) {
    vPos.y = 1.0;
}
if (vPos.y > 1.0) {
    vPos.y = -1.0;
}
// Write back
particlesB.particles[index].pos = vPos;
particlesB.particles[index].vel = vVel;
}
```

Note that in the vertex shader, we add an output variable, `v_vel`, for velocity, which will be used for coloring the birds according to their velocities.

### 13.3.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch13_boids"
path = "examples/ch13/boids.rs"
```

We can then run our application by issuing the following *cargo run* command in the terminal window:

```
cargo run --example ch13_boids "0.5"
```

This will produce the results shown in Fig.13-2.

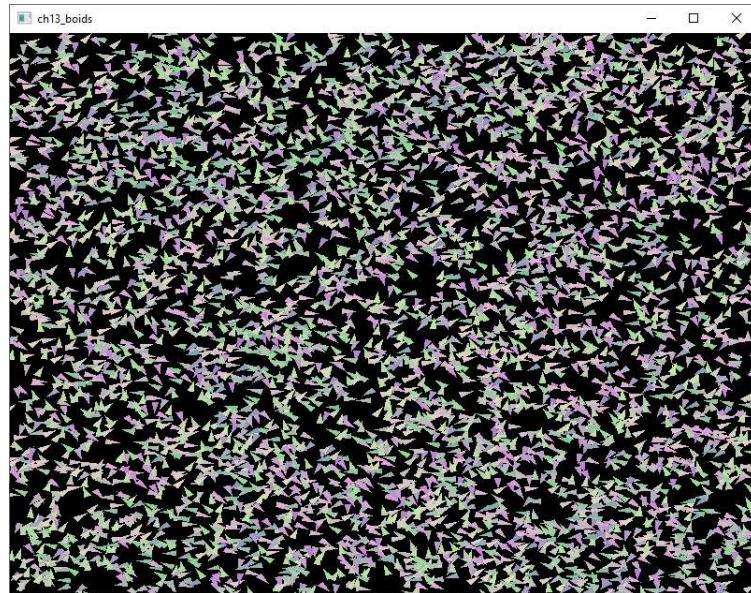


Fig.13-2. Birds distributed randomly with different colors at the beginning.

You can see that at the beginning, the birds are distributed on the scene randomly with different positions, velocities, and colors. However, they will gradually form larger clusters of similar velocities and colors. Eventually, they will evolve into several large clusters with the same velocity and color. The final color depends on the initial conditions, such as particle size and color control level. Fig.13-3 shows the results after several minutes of evolution from the initial results displayed in Fig.13-2.

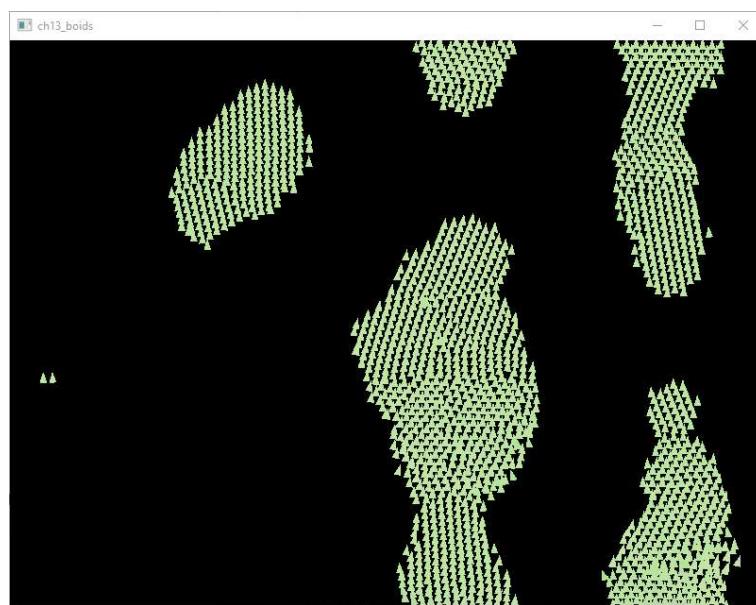


Fig.13-3. Birds patterns after several-minute evolution.

## 13.4 Particles under Gravity

In this section, I will demonstrate how to use a compute shader to simulate the effect of gravity on a particle system containing of 50,000 particles. For our simulation, we will define three mass centers (attractors), which can also be called black holes. They will be the only objects that affect our particles, and they will apply a force on each particle inversely proportional to the square of the distance between the particle and the mass centers.

At the beginning, the three mass centers are created at random locations, and the particles are distributed randomly with different velocities. A few minutes later, the particles will gradually be attracted toward the black holes by the force of gravity. This example is based on the WebGPU sample created by Tarek Sherif (please see his original sample at <https://github.com/tsherif/webgpu-examples/blob/gh-pages/particles.html>).

### 13.4.1 Rust Code

Add a new Rust file called *attractors.rs* to the *examples/ch13/* folder and enter the following content into it:

```
#![allow(dead_code)]
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
    window::WindowBuilder,
};
use rand::{
    distributions::{Distribution, Uniform}
};
#[path = "../common/transforms.rs"]
mod transforms;

const PARTICLES_PER_GROUP: u32 = 64;

struct State {
    init: transforms::InitWgpu,

    // compute
    position_buffers: Vec<wgpu::Buffer>,
    velocity_buffers: Vec<wgpu::Buffer>,
    color_buffer: wgpu::Buffer,
    mass_uniform_buffer: wgpu::Buffer,
    compute_bind_groups: Vec<wgpu::BindGroup>,
    compute_pipeline: wgpu::ComputePipeline,
    num_particles: u32,

    // render
    uniform_buffer: wgpu::Buffer,
    vertex_buffer: wgpu::Buffer,
    render_bind_group: wgpu::BindGroup,
    render_pipeline: wgpu::RenderPipeline,
    frame_num: usize,
}

impl State {
```

```

fn required_limits() -> wgpu::Limits {
    wgpu::Limits::downlevel_defaults()
}

fn required_downlevel_capabilities() -> wgpu::DownlevelCapabilities {
    wgpu::DownlevelCapabilities {
        flags: wgpu::DownlevelFlags::COMPUTE_SHADERS,
        ..Default::default()
    }
}

async fn new(window: &Window, num_particles: u32, particle_size: f32, color_opacity:f32,
            mass_factor:Vec<f32>) -> Self {
    let init = transforms::InitWgpu::init_wgpu(window).await;

    let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
        label: Some("Shader"),
        source: wgpu::ShaderSource::Wgsl(include_str!("attractors.wgsl").into()),
    });

    // compute ****
}

let mut position_data = vec![0.0f32; num_particles as usize * 4];
let mut velocity_data = vec![0.0f32; num_particles as usize * 4];
let mut color_data = vec![0.0f32; num_particles as usize * 4];
let mut rng = rand::thread_rng();
let unif_mp = Uniform::new_inclusive(-1.0, 1.0);
let unif_p = Uniform::new_inclusive(0.0, 1.0);

for position_chunck in position_data.chunks_mut(4){
    position_chunck[0] = unif_mp.sample(&mut rng);
    position_chunck[1] = unif_mp.sample(&mut rng);
    position_chunck[2] = 0.0;
    position_chunck[3] = 1.0;
}
for velocity_chunck in velocity_data.chunks_mut(4){
    velocity_chunck[0] = unif_mp.sample(&mut rng)*0.0001;
    velocity_chunck[1] = unif_mp.sample(&mut rng)*0.0001;
    velocity_chunck[2] = 0.0;
    velocity_chunck[3] = 1.0;
}
for color_chunck in color_data.chunks_mut(4){
    color_chunck[0] = unif_p.sample(&mut rng);
    color_chunck[1] = unif_p.sample(&mut rng);
    color_chunck[2] = unif_p.sample(&mut rng);
    color_chunck[3] = color_opacity;
}

let mut position_buffers = Vec::<wgpu::Buffer>::new();
let mut velocity_buffers = Vec::<wgpu::Buffer>::new();

for i in 0..2 {
    position_buffers.push(
        init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some(&format!("Position Buffer {}", i)),
            contents: bytemuck::cast_slice(&position_data),
            usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::STORAGE |
                wgpu::BufferUsages::COPY_DST,
        }),
    );
}

```

```

velocity_buffers.push(
    init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
        label: Some(&format!("Velocity Buffer {}", i)),
        contents: bytemuck::cast_slice(&velocity_data),
        usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::STORAGE | wgpu::BufferUsages::COPY_DST,
    })
);
}

let color_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Color Buffer")),
    contents: bytemuck::cast_slice(&color_data),
    usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::COPY_DST,
});

let mass_uniform_data = [
    unif_mp.sample(&mut rng), unif_mp.sample(&mut rng), 0.0, 1.0, // mass 1 position
    unif_mp.sample(&mut rng), unif_mp.sample(&mut rng), 0.0, 1.0, // mass 2 position
    unif_mp.sample(&mut rng), unif_mp.sample(&mut rng), 0.0, 1.0, // mass 3 position
    unif_p.sample(&mut rng) * mass_factor[0]/num_particles as f32, // mass 1 factor
    unif_p.sample(&mut rng) * mass_factor[1]/num_particles as f32, // mass 2 factor
    unif_p.sample(&mut rng) * mass_factor[2]/num_particles as f32, // mass 3 factor
].to_vec();

let mass_uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Mass Attractor Uniform Buffer")),
    contents: bytemuck::cast_slice(&mass_uniform_data),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let compute_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: true },
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: true },
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 2,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: false },
                    has_dynamic_offset: false,
                }
            }
        ]
    }
);

```

```

        min_binding_size: None,
    },
    count: None,
},
wgpu::BindGroupLayoutEntry {
    binding: 3,
    visibility: wgpu::ShaderStages::COMPUTE,
    ty: wgpu::BindingType::Buffer {
        ty: wgpu::BufferBindingType::Storage { read_only: false },
        has_dynamic_offset: false,
        min_binding_size: None,
    },
    count: None,
},
wgpu::BindGroupLayoutEntry {
    binding: 4,
    visibility: wgpu::ShaderStages::COMPUTE,
    ty: wgpu::BindingType::Buffer {
        ty: wgpu::BufferBindingType::Uniform,
        has_dynamic_offset: false,
        min_binding_size: None,
    },
    count: None,
},
],
label: None,
});

let mut compute_bind_groups = Vec::new();
for i in 0..2 {
    compute_bind_groups.push(
        init.device.create_bind_group(&wgpu::BindGroupDescriptor {
            layout: &compute_bind_group_layout,
            entries: &[
                wgpu::BindGroupEntry {
                    binding: 0,
                    resource: position_buffers[i % 2].as_entire_binding(),
                },
                wgpu::BindGroupEntry {
                    binding: 1,
                    resource: velocity_buffers[i % 2].as_entire_binding(),
                },
                wgpu::BindGroupEntry {
                    binding: 2,
                    resource: position_buffers[(i + 1) % 2].as_entire_binding(),
                },
                wgpu::BindGroupEntry {
                    binding: 3,
                    resource: velocity_buffers[(i + 1) % 2].as_entire_binding(),
                },
                wgpu::BindGroupEntry {
                    binding: 4,
                    resource: mass_uniform_buffer.as_entire_binding(),
                },
            ],
            label: None,
        })
    );
}
}

```

## 362 | Practical GPU Graphics with wgpu and Rust

```
let compute_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("compute"),
    bind_group_layouts: &[&compute_bind_group_layout],
    push_constant_ranges: &[],
});

let compute_pipeline = init.device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
    label: Some("Compute Pipeline"),
    layout: Some(&compute_pipeline_layout),
    module: &shader,
    entry_point: "cs_main",
});

// render ****

let uniform_data = vec![
    init.config.width as f32,
    init.config.height as f32,
    particle_size as f32,
];
let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Uniform Buffer")),
    contents: bytemuck::cast_slice(&uniform_data),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let vertex_data = vec![
    -1.0f32, -1.0,
    1.0, -1.0,
    -1.0, 1.0,
    1.0, 1.0,
].to_vec();

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Vertex Buffer")),
    contents: bytemuck::cast_slice(&vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});

let render_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
        ],
        label: None,
    });
let render_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &render_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
```

```

        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    },
],
label: Some("Render Bind Group"),
});

let render_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("render"),
    bind_group_layouts: &[&render_bind_group_layout],
    push_constant_ranges: &[],
});

let render_pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: None,
    layout: Some(&render_pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[
            wgpu::VertexBufferLayout {
                array_stride: 8,
                step_mode: wgpu::VertexStepMode::Vertex,
                attributes: &wgpu::vertex_attr_array![0 => Float32x2],
            },
            wgpu::VertexBufferLayout {
                array_stride: 16,
                step_mode: wgpu::VertexStepMode::Instance,
                attributes: &wgpu::vertex_attr_array![1 => Float32x4],
            },
            wgpu::VertexBufferLayout {
                array_stride: 16,
                step_mode: wgpu::VertexStepMode::Instance,
                attributes: &wgpu::vertex_attr_array![2 => Float32x4],
            },
        ],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[init.config.format.into()],
    }),
    primitive: wgpu::PrimitiveState{
        topology:wgpu::PrimitiveTopology::TriangleStrip,
        strip_index_format:Some(wgpu::IndexFormat::UInt32),
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
}

Self {
    init,
    // compute
    position_buffers,
    velocity_buffers,
    color_buffer,
    mass_uniform_buffer,
    compute_bind_groups,
}

```

```

        compute_pipeline,
        num_particles,

        // render
        uniform_buffer,
        vertex_buffer,
        render_pipeline,
        render_bind_group,
        frame_num: 0,
    }
}

fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {
    // empty
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self.init.device.create_command_encoder(&wgpu::CommandEncoderDescriptor {
        label: Some("Command Encoder"),
    });

    {
        // compute pass
        let mut compute_pass = encoder.begin_compute_pass(
            &wgpu::ComputePassDescriptor { label: None });
        compute_pass.set_pipeline(&self.compute_pipeline);
        compute_pass.set_bind_group(0, &self.compute_bind_groups[self.frame_num % 2], &[]);
        compute_pass.dispatch(self.num_particles, 1, 1);
    }

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.0,
                        g: 0.0,
                        b: 0.0,
                    })
                }
            }]
        });
    }
}

```

```

        a: 1.0,
    }),
    store: true,
},
],
depth_stencil_attachment: None,
));

render_pass.set_pipeline(&self.render_pipeline);
render_pass.set_bind_group(0, &self.render_bind_group, &[]);
render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
render_pass.set_vertex_buffer(1, self.color_buffer.slice(..));
render_pass.set_vertex_buffer(2, self.position_buffers[(self.frame_num + 1) % 2].slice(..));

render_pass.draw(0..4, 0..self.num_particles);
}

self.frame_num += 1;

self.init.queue.submit(std::iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
let mut num_particles = "20000";
let mut size = "2.0";
let opacity = 0.5;
let mass = vec![10.0, 10.0, 10.0];

let args: Vec<String> = std::env::args().collect();
if args.len() > 1 {
    num_particles = &args[1];
}
let np = num_particles.parse::<u32>().unwrap();
if args.len() > 2 {
    size = &args[2];
}
let sz = size.parse::<f32>().unwrap();

env_logger::init();
let event_loop = EventLoop::new();
let window = WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(*format!("ch13_attractors"));

let mut state = pollster::block_on(State::new(&window, np, sz,
opacity, mass));

event_loop.run(move |event, _, control_flow| {
match event {
Event::WindowEvent {
ref event,
window_id,
} if window_id == window.id() => {
if !state.input(event) {
match event {
WindowEvent::CloseRequested
| WindowEvent::KeyboardInput {

```

```

        input:
        KeyboardInput {
            state: ElementState::Pressed,
            virtual_keycode: Some(VirtualKeyCode::Escape),
            ..
        },
        ..
    } => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
}

match state.render() {
    Ok(_) => {}
    Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
    Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
    Err(e) => eprintln!("{}:", e),
}
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

In this example, each particle is simulated using a quad shape based on the *triangle-strip* primitive. By default, our particle system contains 50,000 particles. Note that different position buffers and velocity buffers are created for two groups of particles – one group is used for writing and the other group for reading. The STORAGE usage flag is assigned to all of these buffers, indicating that they will be used in the compute shader.

In addition, we create a uniform buffer for storing the mass parameters of the three mass centers, including their positions and mass factors. The initial positions of the mass centers are randomly generated, while the mass factor for each mass center can be specified, allowing you to examine the gravitational effect of each mass center with different masses.

Notice that, we created two bind groups for the compute pipeline, which will be used for different frames. For a specified frame, we write data to one buffer and read data from the other buffer; while for the next frame, we will switch the two buffers for writing and reading data.

### 13.4.2 Shader Code

The shader program in this example is also more complex since it contains three types of shaders: vertex, fragment, and compute.

Add a new shader file called *attractors.wgsl* to the *examples/ch13/* folder with the following code:

```

// vertex shader

[[block]] struct VertexUniforms {
    screenDimensions : vec2<f32>;
    particleSize : f32;
};

[[binding(0), group(0)]] var<uniform> uniforms : VertexUniforms;

struct Input {
    [[location(0)]] vertexPosition : vec2<f32>;
    [[location(1)]] color : vec4<f32>;
    [[location(2)]] position : vec4<f32>;
};

struct Output {
    [[builtin(position)]] Position : vec4<f32>;
    [[location(0)]] vColor : vec4<f32>;
};

[[stage(vertex)]]
fn vs_main(input: Input) -> Output {
    var output: Output;
    output.vColor = input.color;
    output.Position = vec4<f32>(
        input.vertexPosition * uniforms.particleSize / uniforms.screenDimensions + input.position.xy,
        input.position.z,
        1.0
    );
    return output;
}

// fragment shader
[[stage(fragment)]]
fn fs_main( [[location(0)]] vColor : vec4<f32>) -> [[location(0)]] vec4<f32> {
    return vec4<f32>(vColor.rgb * vColor.a, vColor.a);
}

// compute shader

[[block]] struct PositionVelocity {
    pv: [[stride(16)]] array<vec4<f32>>;
};

[[block]] struct Mass {
    mass1Position : vec4<f32>;
    mass2Position : vec4<f32>;
    mass3Position : vec4<f32>;
    mass1Factor : f32;
    mass2Factor : f32;
    mass3Factor : f32;
};

[[binding(0), group(0)]] var<storage, read> positionIn : PositionVelocity;
[[binding(1), group(0)]] var<storage, read> velocityIn : PositionVelocity;
[[binding(2), group(0)]] var<storage, read_write> positionOut : PositionVelocity;
[[binding(3), group(0)]] var<storage, read_write> velocityOut : PositionVelocity;
[[binding(4), group(0)]] var<uniform> mass : Mass;

[[stage(compute), workgroup_size(64)]]
fn cs_main([[builtin(global_invocation_id)]] GlobalInvocationID : vec3<u32>) {
    var index:u32 = GlobalInvocationID.x;
    var position:vec3<f32> = positionIn.pv[index].xyz;
}

```

```

var velocity:vec3<f32> = velocityIn.pv[index].xyz;
var massVec:vec3<f32> = mass.mass1Position.xyz-position;
var massDist2:f32 = max(0.01, dot(massVec, massVec));
var acceleration:vec3<f32> = mass.mass1Factor/massDist2 * normalize(massVec);
massVec = mass.mass2Position.xyz-position;
massDist2 = max(0.01, dot(massVec, massVec));
acceleration = acceleration + mass.mass2Factor/massDist2 * normalize(massVec);
massVec = mass.mass3Position.xyz-position;
massDist2 = max(0.01, dot(massVec, massVec));
acceleration = acceleration + mass.mass3Factor/massDist2 * normalize(massVec);
velocity = velocity + acceleration;
velocity = 0.995 * velocity;

//write back
positionOut.pv[index] = vec4<f32>(position + velocity, 1.0);
velocityOut.pv[index] = vec4<f32>(velocity, 1.0);
}

```

Here, we use Newton's law of gravity, force  $\sim mass/r^2$ , to calculate the acceleration of each particle. We only consider the interaction between particles and mass centers, and neglect the interaction between different particles and interaction between different mass centers.

Note that in the compute shader, we use the built-in variable `global_invocation_id` to access each particle's position and velocity. In our calculations, we do not use any loops to compute the acceleration for each particle, instead, we compute it in parallel for all particles. This is why doing the calculations in GPU is much faster than doing them in CPU, where we would need to use a for-loop for every particle.

### 13.4.3 Run Application

Now, add the following code snippet to the `Cargo.toml` file:

```

[[example]]
name = "ch13_attractors"
path = "examples/ch13/attractors.rs"

```

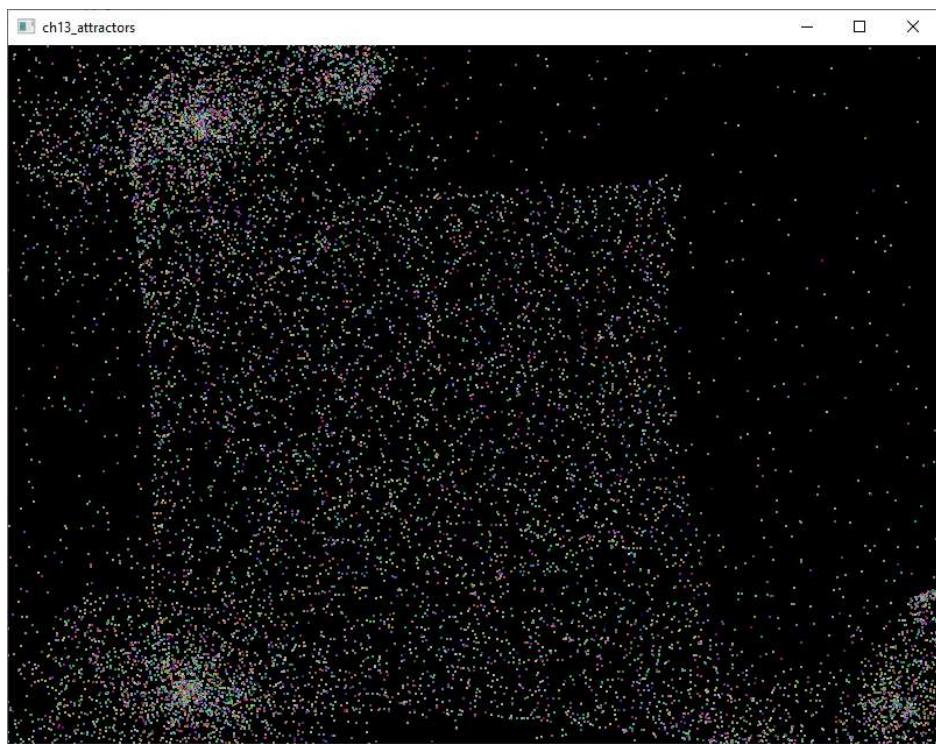
We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch13_attractors
```

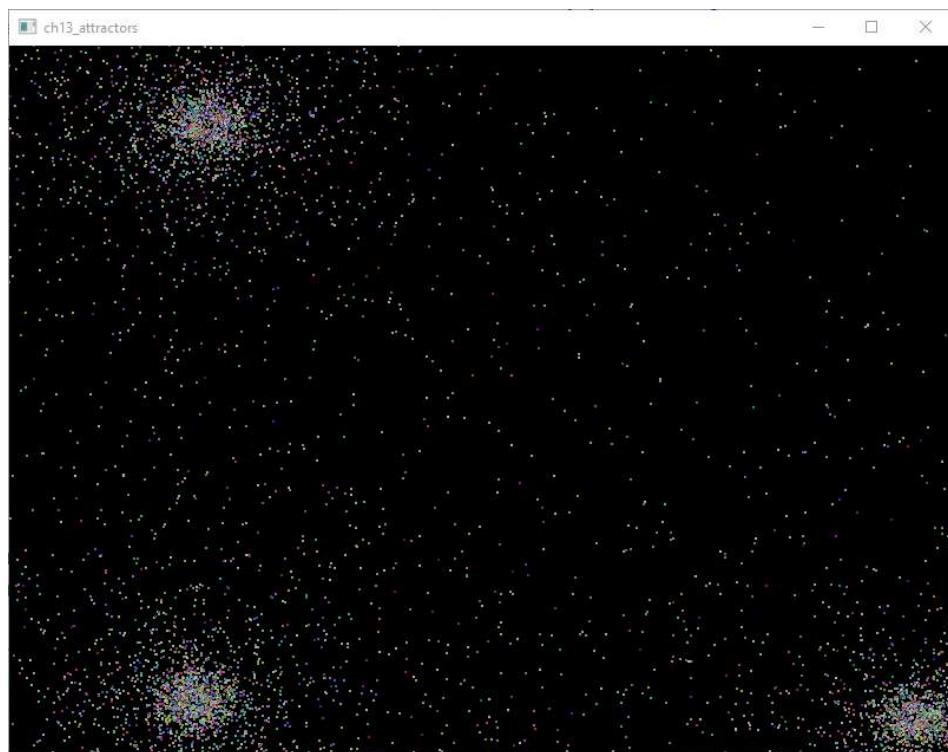
This will produce the results shown in Fig.13-4.

You can see that at the beginning, the particles are distributed randomly. However, after a few minutes, the particles are gradually attracted toward the three mass centers. Depending on the initial locations and mass factors of the mass centers, the way that the three mass centers attract particles may look very different: one mass center may attract more particles than the others may.

Fig.13-5 shows the results after several minutes of evolution. You can clearly see that the particles form three clusters around the mass centers.



*Fig.13.4. Particles distributed randomly at the beginning.*



*Fig.13-5. Particles attracted toward mass centers.*

## 13.5 Particle Collision

In this example, we will simulate a 2D particle system based on standard kinematics as expressed by the following equation:

$$\vec{P}(t) = \vec{P}_0 + \vec{v}_0 t + \frac{1}{2} \vec{a} t^2$$

This equation describes the position of a particle at time  $t$ . Here  $\vec{P}_0$  is the initial position,  $\vec{v}_0$  is the initial velocity, and  $\vec{a}$  is the acceleration.

In our 2D particle system, there are four walls the particles can bounce off of. We need to check, in every frame, whether the particles are bouncing off the edge of our canvas – if yes, we need to reverse the movements of the particles so that they move in the opposite direction and stay within the visible boundaries. This example is based on the WebGPU sample created by Georgi Nikoloff (please see his original sample at <https://gnikoloff.github.io/webgpu-dojo/examples/gpu-compute-matrix/>).

### 13.5.1 Rust Code

Add a new Rust file called *particles.rs* to the *examples/ch13/* folder and enter the following content into it:

```
#![allow(dead_code)]
use std::time::SystemTime;
use cgmath::Matrix4;
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    window::Window,
    event_loop::{ControlFlow, EventLoop},
    window::WindowBuilder,
};
use rand::{
    distributions::{Distribution, Uniform},
    SeedableRng,
};
#[path = "../common/transforms.rs"]
mod transforms;

const PARTICLES_PER_GROUP: u32 = 64;

struct State {
    init: transforms::InitWgpu,
    // compute
    particle_buffer: wgpu::Buffer,
    particle_uniform_data: Vec<f32>,
    particle_uniform_buffer: wgpu::Buffer,
    compute_bind_group: wgpu::BindGroup,
    compute_pipeline: wgpu::ComputePipeline,
    work_group_count: u32,
    num_particles: u32,
    // render
    view_mat: Matrix4<f32>,
    project_mat: Matrix4<f32>,
```

```

uniform_buffer: wgpu::Buffer,
vertex_buffer: wgpu::Buffer,
render_bind_group: wgpu::BindGroup,
render_pipeline: wgpu::RenderPipeline,

// parameters
unif_mp: Uniform<f32>,
rng: rand::rngs::StdRng,
start: SystemTime,
t0: f32,
t1: f32,
}

impl State {
    fn required_limits() -> wgpu::Limits {
        wgpu::Limits::downlevel_defaults()
    }

    fn required_downlevel_capabilities() -> wgpu::DownlevelCapabilities {
        wgpu::DownlevelCapabilities {
            flags: wgpu::DownlevelFlags::COMPUTE_SHADERS,
            ..Default::default()
        }
    }
}

async fn new(window: &Window, num_particles: u32, particle_size: f32) -> Self {
    let start = SystemTime::now();
    let init = transforms::InitWgpu::init_wgpu(window).await;

    let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
        label: Some("Shader"),
        source: wgpu::ShaderSource::Wgsl(include_str!("particles.wgsl").into()),
    });

    // compute ****
    let mut particle_data = vec![0.0f32; num_particles as usize * 8];
    let mut rng = rand::rngs::StdRng::seed_from_u64(42);
    let unif_mp = Uniform::new_inclusive(-1.0, 1.0);
    let unif_p = Uniform::new_inclusive(0.0, 1.0);
    for particle_chunck in particle_data.chunks_mut(8) {
        // position
        particle_chunck[0] = unif_p.sample(&mut rng)*init.config.width as f32 * 2.0;
        particle_chunck[1] = unif_p.sample(&mut rng)*init.config.height as f32 * 2.0;
        // velocity
        particle_chunck[2] = unif_mp.sample(&mut rng)*400.0;
        particle_chunck[3] = unif_mp.sample(&mut rng)*400.0;
        // scale factor for particle size
        particle_chunck[4] = unif_p.sample(&mut rng) + 1.0;
        // color rgb
        particle_chunck[5] = unif_p.sample(&mut rng);
        particle_chunck[6] = unif_p.sample(&mut rng);
        particle_chunck[7] = unif_p.sample(&mut rng);
    }

    let particle_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
        label: Some(&format!("Particle Buffer")),
        contents: bytemuck::cast_slice(&particle_data),
        usage: wgpu::BufferUsages::VERTEX | wgpu::BufferUsages::STORAGE,
    });
}

```

```

let particle_uniform_data = [
    init.config.width as f32,           // size
    init.config.height as f32,
    0.0,                                // delta_frame
    0.5,                                // bounce_factor
    unif_mp.sample(&mut rng)*240.0,      // acceleration left
    unif_mp.sample(&mut rng)*240.0,
    unif_mp.sample(&mut rng)*240.0,
    unif_mp.sample(&mut rng)*240.0,
    unif_mp.sample(&mut rng)*240.0,      // acceleration right
    unif_mp.sample(&mut rng)*240.0,
    unif_mp.sample(&mut rng)*240.0,
    unif_mp.sample(&mut rng)*240.0,
].to_vec();

let particle_uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Particle Uniform Buffer")),
    contents: bytemuck::cast_slice(&particle_uniform_data),
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
});

let compute_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Storage { read_only: false },
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
            wgpu::BindGroupLayoutEntry {
                binding: 1,
                visibility: wgpu::ShaderStages::COMPUTE,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
        ],
        label: None,
    });
}

let compute_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &compute_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: particle_buffer.as_entire_binding(),
        },
        wgpu::BindGroupEntry {
            binding: 1,
            resource: particle_uniform_buffer.as_entire_binding(),
        },
    ],
});

```

```

        ],
        label: Some("Compute Bind Group"),
    });

let compute_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("compute"),
    bind_group_layouts: &[&compute_bind_group_layout],
    push_constant_ranges: &[],
});

let compute_pipeline = init.device.create_compute_pipeline(&wgpu::ComputePipelineDescriptor {
    label: Some("Compute Pipeline"),
    layout: Some(&compute_pipeline_layout),
    module: &shader,
    entry_point: "cs_main",
});

// render ****
****

let camera_position = (0.0, 0.0, 2.0).into();
let look_direction = (0.0,0.0,0.0).into();
let up_direction = cgmath::Vector3::unit_y();
let right = init.config.width as f32;
let top = init.config.height as f32;

let (view_mat,project_mat, _vp) =
    transforms::create_view_projection_ortho(0.0, right, 0.0, top, -2.0, 3.0,
    camera_position, look_direction, up_direction);

let uniform_buffer = init.device.create_buffer(&wgpu::BufferDescriptor{
    label: Some("Uniform Buffer"),
    size: 2 * 16 * 4,
    usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
    mapped_at_creation: false,
});

let vertex_data = vec![
    -particle_size/2.0, -particle_size/2.0,
    particle_size/2.0, -particle_size/2.0,
    particle_size/2.0, particle_size/2.0,
    -particle_size/2.0, particle_size/2.0,
].to_vec();

let vertex_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
    label: Some(&format!("Vertex Buffer")),
    contents: bytemuck::cast_slice(&vertex_data),
    usage: wgpu::BufferUsages::VERTEX,
});

let render_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor {
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::VERTEX,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
            },
        ],
        label: Some("Compute Bind Group"),
    });

```

```

        count: None,
    },
],
label: None,
});

let render_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &render_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Render Bind Group"),
});

let render_pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("render"),
    bind_group_layouts: &[&render_bind_group_layout],
    push_constant_ranges: &[],
});

let render_pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: None,
    layout: Some(&render_pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[
            wgpu::VertexBufferLayout {
                array_stride: 2 * 4,
                step_mode: wgpu::VertexStepMode::Vertex,
                attributes: &wgpu::vertex_attr_array![0 => Float32x2,
                ],
            },
            wgpu::VertexBufferLayout {
                array_stride: 4 * 8,
                step_mode: wgpu::VertexStepMode::Instance,
                attributes: &wgpu::vertex_attr_array![1 => Float32x2, 2 => Float32x2,
                3=> Float32, 4=>Float32x3],
            },
        ],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[init.config.format.into()],
    }),
    primitive: wgpu::PrimitiveState{
        topology:wgpu::PrimitiveTopology::TriangleStrip,
        strip_index_format:Some(wgpu::IndexFormat::Uint32),
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
}

let work_group_count = ((num_particles as f32) / (PARTICLES_PER_GROUP as f32)).ceil() as u32;
Self {

```

```

    init,

    // compute
    particle_buffer,
    particle_uniform_data,
    particle_uniform_buffer,
    compute_bind_group,
    compute_pipeline,
    work_group_count,
    num_particles,

    // render
    view_mat,
    project_mat,
    uniform_buffer,
    vertex_buffer,
    render_pipeline,
    render_bind_group,

    // parameters
    unif_mp,
    rng,
    start,
    t0: 0.0,
    t1: 0.0,
}
}

fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {
    // empty
}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {
    let t = self.start.elapsed().unwrap().as_millis() as f32/1000.0;
    let dt0 = t - self.t0;
    if dt0 >= 1.5 {
        for i in 4..12 {
            self.particle_uniform_data[i] = self.unif_mp.sample(&mut self.rng)*240.0;
        }
        self.t0 = t;
    }
    let dt1 = t - self.t1;
    self.t1 = t;
    self.particle_uniform_data[2] = dt1;

    let output = self.init.surface.get_current_texture()?;
}

```

## 376 | Practical GPU Graphics with wgpu and Rust

```
let view = output
    .texture
    .create_view(&wgpu::TextureViewDescriptor::default());

let mut encoder = self.init.device.create_command_encoder(&wgpu::CommandEncoderDescriptor {
    label: Some("Render Encoder"),
});

{
    // compute pass
    let mut compute_pass = encoder.begin_compute_pass(
        &wgpu::ComputePassDescriptor { label: None });
    compute_pass.set_pipeline(&self.compute_pipeline);
    compute_pass.set_bind_group(0, &self.compute_bind_group, &[]);
    compute_pass.dispatch(self.work_group_count, 1, 1);
}

{
    // render pass
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {
                    r: 0.0,
                    g: 0.0,
                    b: 0.0,
                    a: 1.0,
                }),
                store: true,
            },
        }],
        depth_stencil_attachment: None,
    });
    let project_ref: &[f32;16] = self.project_mat.as_ref();
    let view_ref: &[f32;16] = self.view_mat.as_ref();
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(project_ref));
    self.init.queue.write_buffer(&self.uniform_buffer, 64, bytemuck::cast_slice(view_ref));
    self.init.queue.write_buffer(&self.particle_uniform_buffer, 0,
        bytemuck::cast_slice(&self.particle_uniform_data));

    render_pass.set_pipeline(&self.render_pipeline);
    render_pass.set_bind_group(0, &self.render_bind_group, &[]);
    render_pass.set_vertex_buffer(0, self.vertex_buffer.slice(..));
    render_pass.set_vertex_buffer(1, self.particle_buffer.slice(..));
    render_pass.draw(0..4, 0..self.num_particles);
}
self.init.queue.submit(std::iter::once(encoder.finish()));
output.present();

Ok(())
}

fn main() {
    let mut num_particles = "100000";
    let mut size = "2.0";
```

```

let args: Vec<String> = std::env::args().collect();
if args.len() > 1 {
    num_particles = &args[1];
}
let np = num_particles.parse::<u32>().unwrap();
if args.len() > 2 {
    size = &args[2];
}
let sz = size.parse::<f32>().unwrap();
env_logger::init();
let event_loop = EventLoop::new();
let window = WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&*format!("ch13_particles"));
let mut state = pollster::block_on(State::new(&window, np, sz));

event_loop.run(move |event, _, control_flow| {
    match event {
        Event::WindowEvent {
            ref event,
            window_id,
        } if window_id == window.id() => {
            if !state.input(event) {
                match event {
                    WindowEvent::CloseRequested
                    | WindowEvent::KeyboardInput {
                        input:
                            KeyboardInput {
                                state: ElementState::Pressed,
                                virtual_keycode: Some(VirtualKeyCode::Escape),
                                ..
                            },
                    ..
                    } => *control_flow = ControlFlow::Exit,
                    WindowEvent::Resized(physical_size) => {
                        state.resize(*physical_size);
                    }
                    WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                        state.resize(**new_inner_size);
                    }
                    _ => {}
                }
            }
        }
        Event::RedrawRequested(_) => {
            state.update();

            match state.render() {
                Ok(_) => {}
                Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
                Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
                Err(e) => eprintln!("{}: {}", e, e),
            }
        }
        Event::MainEventsCleared => {
            window.request_redraw();
        }
        _ => {}
    });
});
}

```

In this example, each particle is simulated using a quad shape based on the *triangle-strip* primitive. By default, our particle system contains 100,000 particles. Note that the *particle\_buffer* contains data for position, velocity, size scaling, and color, which are initialized with the random values. The STORAGE usage flag is assigned to this buffer, indicating that it will be used in the compute shader.

In addition, we create a *particle\_uniform\_buffer* to store the bounce factor, time interval, and accelerations for different directions. Here, we also initialize the accelerations with random numbers to increase the randomness of the particle movements.

Note that inside the *render* function, we update the accelerations every two seconds, while we update the time interval every frame.

This program also allows you to specify the number of particles and particle size.

### 13.5.2 Shader Code

As before, the shader program in this example is also more complex because it contains three types of shaders: vertex, fragment, and compute.

Add a new shader file called *particles.wgsl* to the *examples/ch13/* folder with the following code:

```
// vertex shader

[[block]] struct Transform {
    projectionMatrix: mat4x4<f32>;
    viewMatrix: mat4x4<f32>;
};

[[group(0), binding(0)]] var<uniform> transform: Transform;

struct Input {
    [[location(0)]] position: vec2<f32>;
    [[location(1)]] instancePosition: vec2<f32>;
    [[location(2)]] instanceVelocity: vec2<f32>;
    [[location(3)]] scaleFactor: f32;
    [[location(4)]] color: vec3<f32>;
};

struct Output {
    [[builtin(position)]] Position : vec4<f32>;
    [[location(0)]] color : vec4<f32>;
};

[[stage(vertex)]]
fn vs_main (input: Input) -> Output {
    var output: Output;
    let scaleMatrix = mat4x4<f32>(
        vec4<f32>(input.scaleFactor, 0.0, 0.0, 0.0),
        vec4<f32>(0.0, input.scaleFactor, 0.0, 0.0),
        vec4<f32>(0.0, 0.0, input.scaleFactor, 0.0),
        vec4<f32>(0.0, 0.0, 0.0, 1.0)
    );
    let pos = vec4<f32>(input.position, 0.0, 1.0);
    let ins_pos = vec4<f32>(input.instancePosition, 0.0, 1.0);
    let transformedPos = scaleMatrix * pos + ins_pos;
    output.Position = transform.projectionMatrix * transform.viewMatrix * transformedPos;
    output.color = vec4<f32>(input.color, 1.0);
    return output;
}
```

```

// fragment shader

[[stage(fragment)]]
fn fs_main (in: Output) -> [[location(0)]] vec4<f32> {
    return in.color;
}

// compute shader

struct ParticleData {
    position: vec2<f32>;
    velocity: vec2<f32>;
    radius: f32;
};

[[block]] struct ParticlesBuffer {
    particles: [[stride(32)]] array<ParticleData>;
};
[[group(0), binding(0)]] var<storage, read_write> particlesBuffer: ParticlesBuffer;

[[block]] struct Uniforms {
    canvasSize: vec2<f32>;
    deltaTime: f32;
    bounceFactor: f32;
    acceLeft: vec4<f32>;
    acceRight: vec4<f32>;
};
[[group(0), binding(1)]] var<uniform> uniforms : Uniforms;

[[stage(compute), workgroup_size(64, 1, 1)]]
fn cs_main([[builtin(global_invocation_id)]] GlobalInvocationID : vec3<u32>) {
    let index = GlobalInvocationID.x;
    let canvasWidth = uniforms.canvasSize.x * 2.0;
    let canvasHeight = uniforms.canvasSize.y * 2.0;

    let particleRadius = particlesBuffer.particles[index].radius;
    let vx = particlesBuffer.particles[index].velocity.x;
    let vy = particlesBuffer.particles[index].velocity.y;
    if (particlesBuffer.particles[index].position.x < canvasWidth * 0.5) {
        if (particlesBuffer.particles[index].position.y > canvasHeight * 0.5) {
            particlesBuffer.particles[index].velocity.x = vx + uniforms.acceLeft.x * uniforms.deltaTime;
            particlesBuffer.particles[index].velocity.y = vy + uniforms.acceLeft.y * uniforms.deltaTime;
        } else {
            particlesBuffer.particles[index].velocity.x = vx + uniforms.acceLeft.z * uniforms.deltaTime;
            particlesBuffer.particles[index].velocity.y = vy + uniforms.acceLeft.w * uniforms.deltaTime;
        }
    } else {
        if (particlesBuffer.particles[index].position.y > canvasHeight * 0.5) {
            particlesBuffer.particles[index].velocity.x = vx + uniforms.acceRight.x * uniforms.deltaTime;
            particlesBuffer.particles[index].velocity.y = vy + uniforms.acceRight.y * uniforms.deltaTime;
        } else {
            particlesBuffer.particles[index].velocity.x = vx + uniforms.acceRight.z * uniforms.deltaTime;
            particlesBuffer.particles[index].velocity.y = vy + uniforms.acceRight.w * uniforms.deltaTime;
        }
    }
    particlesBuffer.particles[index].position.x = particlesBuffer.particles[index].position.x
        + particlesBuffer.particles[index].velocity.x * uniforms.deltaTime;
    particlesBuffer.particles[index].position.y = particlesBuffer.particles[index].position.y
        + particlesBuffer.particles[index].velocity.y * uniforms.deltaTime;
}

```

```
// handle screen viewport
if (particlesBuffer.particles[index].position.x + particleRadius * 0.5 > canvasWidth) {
    particlesBuffer.particles[index].position.x = canvasWidth - particleRadius * 0.5;
    particlesBuffer.particles[index].velocity.x = vx * -uniforms.bounceFactor;
} elseif (particlesBuffer.particles[index].position.x - particleRadius * 0.5 < 0.0) {
    particlesBuffer.particles[index].position.x = particleRadius * 0.5;
    particlesBuffer.particles[index].velocity.x = vx * -uniforms.bounceFactor;
}
if (particlesBuffer.particles[index].position.y + particleRadius * 0.5 > canvasHeight) {
    particlesBuffer.particles[index].position.y = canvasHeight - particleRadius * 0.5;
    particlesBuffer.particles[index].velocity.y = vy * -uniforms.bounceFactor;
} elseif (particlesBuffer.particles[index].position.y - particleRadius * 0.5 < 0.0) {
    particlesBuffer.particles[index].position.y = particleRadius * 0.5;
    particlesBuffer.particles[index].velocity.y = vy * -uniforms.bounceFactor;
}
}
```

In the vertex shader, we construct a scaling matrix using the scaling factor, and then use it to scale the particle size. Since the scaling factor is generated with random numbers, the particles after this scaling transformation will have different sizes.

Note that in the compute shader, we use the built-in variable `global_invocation_id` to access each particle's position and velocity. We first update the particle velocities using the accelerations and then update the particle positions with the updated velocities. Next, we reverse the velocity directions according to the bouncing factor when particles hit the walls.

In our calculations, we do not use any loops to update the position and velocity for each particle, instead, we update them in parallel for all particles. This is why doing our calculations in GPU is much faster than it would be in CPU, where we would need to use a for-loop to perform calculations for every particle.

### 13.5.3 Run Application

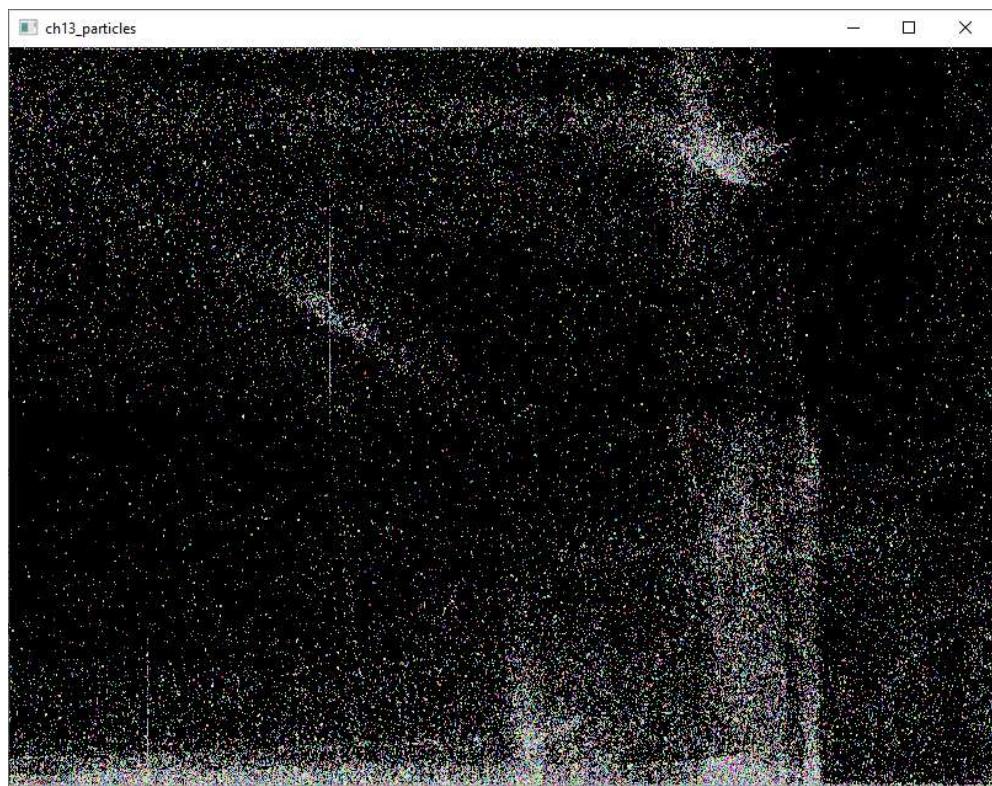
Now, add the following code snippet to the `Cargo.toml` file:

```
[[example]]
name = "ch13_particles"
path = "examples/ch13/particles.rs"
```

We can then run our application by issuing the following `cargo run` command in the terminal window:

```
cargo run --example ch13_particles
```

This will produce a result using the default parameters, as shown in Fig.13-6.



*Fig. 13-6. Simulated particle movement.*



# 14 Visualizing Complex Functions

Complex numbers, complex functions, and complex analysis are part of an important branch of mathematics. They find wide applications in solving real scientific and engineering problems. For any complex function, the values  $z$  and their function  $f(z)$  from the complex domain may be separated into real and imaginary parts:

$$\begin{aligned} z &= x + iy \\ f(z) &= f(x + iy) = u(x, y) + iv(x, y) \end{aligned}$$

Where  $x$ ,  $y$ ,  $u(x,y)$ , and  $v(x,y)$  are all real-valued.

In this chapter, you will learn how to visualize complex functions using the domain coloring approach. In addition, I will explain how to create fractal plots for the Mandelbrot and Julia sets, as well as 3D fractal images.

Normally, we would need a four-dimensional space to visualize complex functions of a single complex variable. But by using the domain coloring approach to encode the values in the complex plane, we can plot complex functions in two-dimensional space.

Generating domain coloring and fractal images using Rust code in `wgpu` is an inherently CPU-intensive process because the images need to be drawn pixel-by-pixel using bitmap rendering. We usually create a  $n \times m$  1-byte array image and display it as a texture mapped onto a rectangle canvas comprising two triangle primitives.

However, each point we compute for domain coloring or fractal images corresponds to a fragment in the color buffer, and its computation does not involve any surrounding points. Consequently, we can perform this computation in GPU – mapping complex points to screen pixels and coloring them with a fragment shader. Here, we will create our domain coloring and fractal images by directly using the fragment shader in the GPU.

## 14.1 Complex Functions in Shader

Unlike real functions, there are no built-in functions for complex numbers in the shader. Here, we will use the `vec2<f32>` data type to represent complex numbers. Add a new sub-folder called `ch14` to the `examples/` folder, and then add a common shader file called `cmath_func.wgsl` to this newly created folder. In the following sections, I will define some math operations and complex functions in this shader file, which will be used to generate our domain coloring and fractal images.

## 14.1.1 Math Operations

Here, I will define some math operations in our `cmath_func.wgsl` file. In WGSL, we use a *let*-declaration outside all functions to define a module-scope constant. Here, we will first introduce two module constants, *pi* and *e*:

```
let pi: f32 = 3.14159265359;
let e: f32 = 2.71828182845;
```

*Addition:* We define a `c_add` function that adds a real constant to a complex number:

```
fn c_add(a:vec2<f32>, s:f32) -> vec2<f32>{
    return vec2<f32>(a.x + s, a.y);
}
```

*Multiplication:* Next, we define multiplication for two complex variables:

```
fn c_mul(a:vec2<f32>, b:vec2<f32>) -> vec2<f32>{
    return vec2<f32>(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x);
}
```

*Division:* We also need to define the division operation for complex variables:

```
fn c_div(a:vec2<f32>, b:vec2<f32>) -> vec2<f32>{
    let d:f32 = dot(b,b);
    return vec2<f32>(dot(a,b)/d, (a.y*b.x-a.x*b.y)/d);
}
```

These math operations will be used in generating our domain coloring and fractal images. Following the procedure presented here, you can easily define your own math operations for complex variables in your shaders.

## 14.1.2 Commonly Used Functions

Here, I will define several commonly used complex functions in the `cmath_func.wgsl` file.

*Complex conjugate:* We can use the following function to calculate the complex conjugate:

```
fn c_conj(z:vec2<f32>) -> vec2<f32>{
    return vec2<f32>(z.x, -z.y);
}
```

This function simply reverses the sign of the imaginary part of a complex variable.

*Argument:* The argument angle of a complex variable is given by:

$$\tan \phi = \frac{z.y}{z.x}$$

The following function will return the argument angle:

```
fn c_arg(z: vec2<f32>) -> f32{
    var f:f32 = atan2(z.y, z.x);
    if(f < 0.0){
        f = f + 2.0*pi;
    }
    return f/(2.0*pi);
}
```

*Square root:* The square-root function for a complex variable can be calculated using the following function:

```
fn c_sqrt(z:vec2<f32>) -> vec2<f32>{
    let m:f32 = length(z);
    let s = sqrt(0.5*vec2<f32>(m + z.x, m - z.x));
    return s*vec2<f32>(1.0, sign(z.y));
}
```

*Inverse:* Given a complex variable  $z$ , the following function calculates its inverse  $1/z$ :

$$\frac{1}{z} = \frac{1}{x + iy} = \frac{x - iy}{x^2 + y^2} = \frac{z.x - z.y}{dot(z, z)}$$

```
fn c_inv(z:vec2<f32>) -> vec2<f32>{
    return vec2<f32>(z.x/dot(z, z), -z.y/dot(z, z));
}
```

*Exponential:* We now consider the exponential function for a complex variable  $z$ :

$$e^z = e^{x+iy} = e^x(\cos y + i \sin y)$$

Using this relationship, we can easily implement the complex exponential function:

```
fn c_exp(z:vec2<f32>) -> vec2<f32>{
    return vec2<f32>(exp(z.x)*cos(z.y), exp(z.x)*sin(z.y));
}
```

*Power function:* Here, we want to calculate the power function  $z^n$ , where  $n$  is a *f32* data type:

$$z^n = (r e^{i\phi})^n = r^n [\cos(n\phi) + i \sin(n\phi)]$$

We can use this relationship to create the complex power function:

```
fn c_pow(z:vec2<f32>, n:f32) -> vec2<f32>{
    let r:f32 = length(z);
    let a:f32 = atan2(z.y, z.x);
    return pow(r, n) * vec2<f32>(cos(a*n), sin(a*n));
}
```

*Logarithm:* The complex log function has a simple relationship in polar coordinates:

$$\log(z) = \log(re^{i\phi}) = \log(r) + i\phi$$

The corresponding function for computing the complex log function is given below:

```
fn c_log(z:vec2<f32>) -> vec2<f32>{
    return vec2<f32>(log(length(z)), atan2(z.y, z.x));
}
```

*Sine function:* A complex sine function can be calculated using complex exponential functions:

$$\begin{aligned} \sin z &= \frac{1}{2i} (e^{iz} - e^{-iz}) = \frac{1}{2i} [e^x(\cos y + i \sin y) - e^{-x}(\cos y - i \sin y)] \\ &= \cosh y \sin x + i \sinh y \cos x \end{aligned}$$

We can easily implement the complex sine function in the fragment shader:

```
fn c_sin(z:vec2<f32>) -> vec2<f32>{
    let a = pow(e, z.y);
    let b = pow(e, -z.y);
```

```
    return vec2<f32>((z.x)*(a+b)*0.5, (z.x)*(a-b)*0.5);
}
```

*Cosine function:* Similar to the case of the complex sine function, a complex cosine function can also be calculated using complex exponential functions:

$$\begin{aligned}\cos z &= \frac{1}{2}(e^{iz} + e^{-iz}) = \frac{1}{2}[e^x(\cos y + i \sin y) + e^{-x}(\cos y - i \sin y)] \\ &= \cosh y \cos x - i \sinh y \sin x\end{aligned}$$

Using this relationship, we can create the complex cosine function in the fragment shader:

```
fn c_cos(z:vec2<f32>) ->vec2<f32>{
    let a = pow(e, z.y);
    let b = pow(e, -z.y);
    return vec2<f32>((z.x)*(a+b)*0.5, -(z.x)*(a-b)*0.5);
}
```

*Tangent function:* A complex tangent function is computed using complex sine and cosine functions:

$$\tan z = \frac{\sin z}{\cos z}$$

Add the following function to the shader code:

```
fn c_tan(z:vec2<f32>) ->vec2<f32>{
    return c_div(c_sin(z), c_cos(z));
}
```

This function can also be implemented without using complex sine and cosine functions:

```
fn c_tan1(z:vec2<f32>) ->vec2<f32>{
    let a = pow(e, z.y);
    let b = pow(e, -z.y);
    let cx = cos(z.x);
    let ab = (a - b)*0.5;
    return vec2<f32>((z.x)*cx, ab*(a+b)*0.5)/(cx*cx+ab*ab);
}
```

*Inverse hyperbolic sine function:* The complex inverse sinh function can be expressed in the following form:

$$\operatorname{asinh}(z) = \log(z + \sqrt{1 + z^2})$$

Using this relationship, add the following function to the shader code:

```
fn c_asinh(z:vec2<f32>) -> vec2<f32>{
    let a = z + c_sqrt(c_mul(z,z) + vec2<f32>(1.0,0.0));
    return c_log(a);
}
```

You can easily add your own custom complex functions to the shader code by following the procedure presented here.

## 14.2 Color Functions

While the RGB color model is the most common way to mix and create colors in computer graphics, the HSV (hue, saturation, value) model is closer to how humans consciously think about color. The RGB

describes colors in terms of red, green, and blue primary colors, while HSV describes colors in terms of their tint (hue), shade (saturation), and brightness (value).

Domain coloring represents complex functions by assigning a color to each point on the complex plane. We usually use the HSV color model to associate hue values with the argument (or phase) of a complex variable  $z$ , defined in the region  $(0, 2\pi)$ . Then, we convert the HSV color model into the RGB model that the fragment shader uses. In this section, we will implement various HSV to RGB color conversion functions that will be used to create domain coloring for complex functions.

### 14.2.1 Color Conversion in Shader

The standard domain coloring approach uses the phase and magnitude of a complex function to determine the hue and the contour lines respectively. The contours of the magnitude occur around the integer value of  $\log_2(|z|)$ . We usually color these contour lines with a light color, while we color the grid lines of the real and imaginary parts of the complex function with a dark color.

Add a new `hsv_to_rgb` function to our `cmath_func.wgsl` shader file with the following code:

```
fn hsv_to_rgb(z:vec2<f32>) -> vec3<f32>{
    let len = length(z);
    let h = c_arg(z);
    var fx = 2.0*(fract(z.x) - 0.5);
    var fy = 2.0*(fract(z.y) - 0.5);
    fx = fx*fx;
    fy = fy*fy;
    var g = 1.0 - (1.0 - fx)*(1.0 - fy);
    g = pow(abs(g), 10.0);
    var c = 2.0*(fract(log2(len)) - 0.5);
    c = 0.7*pow(abs(c), 10.0);
    var v = 1.0 - 0.5*g;
    let f = abs((h*6.0 + vec3<f32>(0.0,4.0,2.0))%6.0 - 3.0) - 1.0;
    var rgb = clamp(f, vec3<f32>(0.0,0.0,0.0), vec3<f32>(1.0,1.0,1.0));
    rgb = rgb*rgb*(3.0 - 2.0*rgb);
    rgb = (1.0-c)*v*mix(vec3<f32>(1.0, 1.0, 1.0), rgb, 1.0);
    return rgb + c*vec3<f32>(1.0,1.0,1.0);
}
```

Here, we first calculate the H, S, and V values using the magnitude and argument of the complex function, and then convert them into RGB color. In this calculation, we use several built-in WGSL functions, including `fract`, `clamp`, and `mix`. The `fract( $e$ )` function returns the fractional bits of  $e$ ; `clamp( $e1, e2, e3$ )` returns  $\min(\max(e1, e2), e3)$ ; and `mix( $e1, e2, e3$ )` returns the component-wise linear blend of  $e1$  and  $e2$  using the scalar blending factor  $e3$  for each component.

I should point out that various versions of HSV to RGB color conversion have been implemented for the purpose of domain coloring for complex functions. The version used in this book allows you to easily discern the behavior of the complex function.

Fig.14-1 (a) shows a complex function  $f(z) = z$  in the complex plane created using the `hsv_to_rgb` color-conversion function. The white circles indicate the magnitude contours and the black gridlines indicate the real and imaginary parts. The white magnitude contours bunch up, while the black gridlines do not, around  $z = 0$ , indicating that there is a zero at  $z = 0$ .

Fig.14-1 (b) shows another complex function  $f(z) = 1/z$  where the pole at  $z = 0$  is indicated by the way the black gridlines bunch up as the white magnitude contours get closer and closer to the pole. From the

results displayed in Fig. 14-1, you can clearly tell the difference between the zeros and poles in the complex plane.

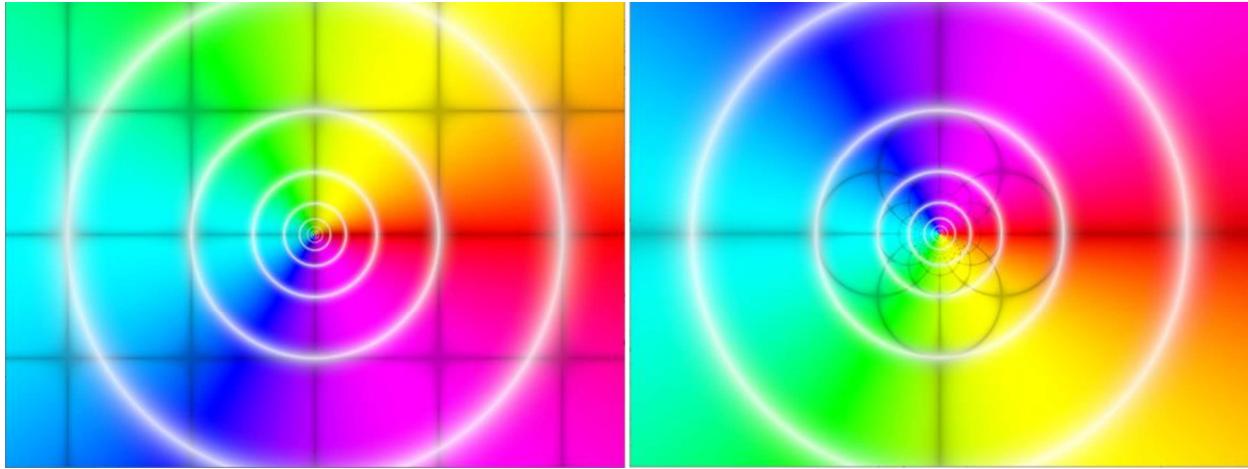


Fig.14-1. (a)  $f(z) = z$  (left) and (b)  $f(z) = 1/z$  (right) plotted using the `hsv_to_rgb` function.

## 14.2.2 Colormaps in Shader

In chapter 9, we discussed implementing colormaps in Rust code. Here, I will explain how to implement colormaps in the shader program, which we can then use to generate domain coloring for complex functions. I now simply provide several commonly used colormaps using various  $m \times 3$  colormap arrays. Add a new function called `colormap` to our `cmath_func.wgsl` shader file with the following code:

```
// colormap
fn colormap(colormap_id:i32) -> array<vec3<f32>, 11>{
    var colors:array<vec3<f32>, 11>;
    switch (colormap_id) {
        case 1: { // jet
            colors = array<vec3<f32>, 11>(
                vec3<f32>(0.0,0.0,0.51),
                vec3<f32>(0.0,0.24,0.67),
                vec3<f32>(0.01,0.49,0.78),
                vec3<f32>(0.01,0.75,0.89),
                vec3<f32>(0.02,1.0,1.0),
                vec3<f32>(0.51,1.0,0.5),
                vec3<f32>(1.0,1.0,0.0),
                vec3<f32>(0.99,0.67,0.0),
                vec3<f32>(0.99,0.33,0.0),
                vec3<f32>(0.98,0.0,0.0),
                vec3<f32>(0.5,0.0,0.0));
        }
        case 2 : { // hsv
            colors = array<vec3<f32>, 11>(
                vec3<f32>(1.0,0.0,0.0),
                vec3<f32>(1.0,0.5,0.0),
                vec3<f32>(0.97,1.0,0.01),
                vec3<f32>(0.0,0.99,0.04),
                vec3<f32>(0.0,0.98,0.52),
                vec3<f32>(0.0,0.98,1.0),
                vec3<f32>(0.01,0.49,1.0),
                vec3<f32>(0.0,0.0,0.0),
                vec3<f32>(0.0,0.0,0.0),
                vec3<f32>(0.0,0.0,0.0),
                vec3<f32>(0.0,0.0,0.0));
        }
    }
}
```

```

        vec3<f32>(0.03,0.0,0.99),
        vec3<f32>(1.0,0.0,0.96),
        vec3<f32>(1.0,0.0,0.49),
        vec3<f32>(1.0,0.0,0.02));
    break;
}
case 3: { // hot
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.0,0.0),
        vec3<f32>(0.3,0.0,0.0),
        vec3<f32>(0.6,0.0,0.0),
        vec3<f32>(0.9,0.0,0.0),
        vec3<f32>(0.93,0.0,0.0),
        vec3<f32>(0.97,0.55,0.0),
        vec3<f32>(1.0,0.82,0.0),
        vec3<f32>(1.0,0.87,0.25),
        vec3<f32>(1.0,0.91,0.5),
        vec3<f32>(1.0,0.96,0.75),
        vec3<f32>(1.0,1.0,1.0));
    break;
}
case 4: { // cool
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.49,0.0,0.7),
        vec3<f32>(0.45,0.0,0.85),
        vec3<f32>(0.42,0.15,0.89),
        vec3<f32>(0.38,0.29,0.93),
        vec3<f32>(0.27,0.57,0.910),
        vec3<f32>(0.0,0.8,0.77),
        vec3<f32>(0.0,0.97,0.57),
        vec3<f32>(0.0,0.98,0.46),
        vec3<f32>(0.0,1.0,0.35),
        vec3<f32>(0.16,1.0,0.03),
        vec3<f32>(0.58,1.0,0.0));
    break;
}
case 5: { // spring
    colors = array<vec3<f32>, 11>(
        vec3<f32>(1.0,0.0,1.0),
        vec3<f32>(1.0,0.1, 0.9),
        vec3<f32>(1.0,0.2,0.8),
        vec3<f32>(1.0,0.3,0.7),
        vec3<f32>(1.0,0.4,0.6),
        vec3<f32>(1.0,0.5,0.5),
        vec3<f32>(1.0,0.6,0.4),
        vec3<f32>(1.0,0.7,0.3),
        vec3<f32>(1.0,0.8,0.2),
        vec3<f32>(1.0,0.9,0.1),
        vec3<f32>(1.0,1.0,0.0));
    break;
}
case 6: { // summer
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.5,0.4),
        vec3<f32>(0.1,0.55,0.4),
        vec3<f32>(0.2,0.6,0.4),
        vec3<f32>(0.3,0.65,0.4),
        vec3<f32>(0.4,0.7,0.4),
        vec3<f32>(0.5,0.75,0.4),

```

```

        vec3<f32>(0.6,0.8,0.4),
        vec3<f32>(0.7,0.85,0.4),
        vec3<f32>(0.8,0.9,0.4),
        vec3<f32>(0.9,0.95,0.4),
        vec3<f32>(1.0,1.0,0.4));
    break;
}
case 7: { // autumn
    colors = array<vec3<f32>, 11>(
        vec3<f32>(1.0,0.0,0.0),
        vec3<f32>(1.0,0.1,0.0),
        vec3<f32>(1.0,0.2,0.0),
        vec3<f32>(1.0,0.3,0.0),
        vec3<f32>(1.0,0.4,0.0),
        vec3<f32>(1.0,0.5,0.0),
        vec3<f32>(1.0,0.6,0.0),
        vec3<f32>(1.0,0.7,0.0),
        vec3<f32>(1.0,0.8,0.0),
        vec3<f32>(1.0,0.9,0.0),
        vec3<f32>(1.0,1.0,0.0));
    break;
}
case 8: { // winter
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.0,1.0),
        vec3<f32>(0.0,0.1,0.95),
        vec3<f32>(0.0,0.2,0.9),
        vec3<f32>(0.0,0.3,0.85),
        vec3<f32>(0.0,0.4,0.8),
        vec3<f32>(0.0,0.5,0.75),
        vec3<f32>(0.0,0.6,0.7),
        vec3<f32>(0.0,0.7,0.65),
        vec3<f32>(0.0,0.8,0.6),
        vec3<f32>(0.0,0.9,0.55),
        vec3<f32>(0.0,1.0,0.5));
    break;
}
case 9: { // bone
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.0,0.0),
        vec3<f32>(0.08,0.08,0.11),
        vec3<f32>(0.16,0.16,0.23),
        vec3<f32>(0.25,0.25,0.34),
        vec3<f32>(0.33,0.33,0.45),
        vec3<f32>(0.41,0.44,0.54),
        vec3<f32>(0.5,0.56,0.62),
        vec3<f32>(0.58,0.67,0.7),
        vec3<f32>(0.66,0.78,0.78),
        vec3<f32>(0.83,0.89,0.89),
        vec3<f32>(1.0,1.0,1.0));
    break;
}
case 10: { // cooper
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.0,0.0),
        vec3<f32>(0.13,0.08,0.05),
        vec3<f32>(0.25,0.16,0.1),
        vec3<f32>(0.38,0.24,0.15),
        vec3<f32>(0.5,0.31,0.2),

```

```

        vec3<f32>(0.62,0.39,0.25),
        vec3<f32>(0.75,0.47,0.3),
        vec3<f32>(0.87,0.55,0.35),
        vec3<f32>(1.0,0.63,0.4),
        vec3<f32>(1.0,0.71,0.45),
        vec3<f32>(1.0,0.78,0.5));
    break;
}
case 11: { // greys
    colors = array<vec3<f32>, 11>(
        vec3<f32>(0.0,0.0,0.0),
        vec3<f32>(0.1,0.1,0.1),
        vec3<f32>(0.2,0.2,0.2),
        vec3<f32>(0.3,0.3,0.3),
        vec3<f32>(0.4,0.4,0.4),
        vec3<f32>(0.5,0.5,0.5),
        vec3<f32>(0.6,0.6,0.6),
        vec3<f32>(0.7,0.7,0.7),
        vec3<f32>(0.8,0.8,0.8),
        vec3<f32>(0.9,0.9,0.9),
        vec3<f32>(1.0,1.0,1.0));
    break;
}
return colors;
}

```

Inside the `colormap` function, each colormap contains 11 RGB color arrays and has a standard colormap name, and can be selected using the `colormap_id`.

Now, we can use this colormap data to convert the HSV color model to the RGB color model. Add a new function called `colormap_to_rgb` to our `cmath_func.wgsl` shader file with the following code:

```

fn colormap_to_rgb(z:vec2<f32>, colormap_id:i32) -> vec3<f32>{
    var c = colormap(colormap_id);
    let len = length(z);
    var h = atan2(z.y, z.x);
    if(h < 0.0) { h = h + 2.0*pi; }
    if(h >= 2.0*pi) { h = h - 2.0*pi; }
    var s:f32 = 0.0;
    var v = vec3<f32>(0.0, 0.0, 0.0);

    for(var i:i32 = 0; i < 11; i = i+1){
        if(h >= 0.2*pi*f32(i) && h < 0.2*pi*(f32(i)+1.0)){
            s = (h - f32(i)*0.2*pi)/(0.2*pi);
            v = s*c[i+1] + (1.0-s)*c[i];
        }
    }
    let b = fract(log2(len));
    return vec3<f32>(v[0]*b, v[1]*b, v[2]*b);
}

```

This function first retrieves the colormap data with the given `colormap_id`, then uses the complex variable `z` and the colormap data to create the HSV color model, and finally converts it into the RGB color model.

Fig.14-2 shows the complex functions  $f(z) = z$  (with a zero at  $z = 0$ ) and  $f(z) = 1/z$  (with a pole at  $z = 0$ ) in the complex plane created using the `jet` colormap. With this colormap model, we can still tell the difference between zeros and poles in the complex plane.

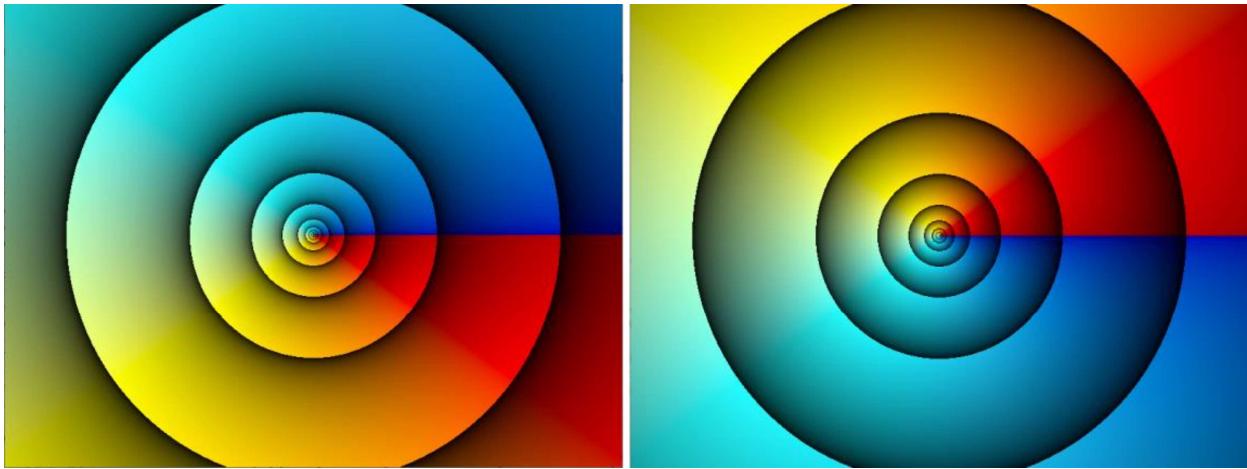


Fig.14-2. (a)  $f(z) = z$  (left) and (b)  $f(z) = 1/z$  (right) plotted using jet colormap.

In the following section, I will explain how to create domain coloring for complex functions using the color models implemented here.

## 14.3 Domain Coloring for Complex Functions

In this section, I will discuss how to generate domain coloring for several complex functions. Although domain coloring can be used to plot any complex functions, we will only consider analytic complex functions here, because analytic functions have very interesting properties that are clearly visible in domain coloring images. These properties include behavior near zeros and poles, branch cuts, singularities, etc.

### 14.3.1 Rust Code

The Rust code for our current example is very simple – it does not require any model-view-projection transformations because domain coloring produces what is basically a 2D image. Add a new Rust file named `domain_color.rs` to the `examples/ch14/` folder with the following code:

```
use std:: {iter, time::SystemTime};
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable };

#[path = "../common/transforms.rs"]
mod transforms;

#[repr(C)]
#[derive(Clone, Copy, Pod, Zeroable)]
struct Vertex {
    pos: [f32; 3],
}
```

```

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    uniform_bind_group: wgpu::BindGroup,
    uniform_buffer: wgpu::Buffer,
    max_iter: i32,
    start: SystemTime,
    t0: f32,
    select: f32,
    select_color: f32,
}

impl State {
    async fn new(window: &Window, select: f32, select_color: f32) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;
        let start = SystemTime::now();

        let s1 = include_str!("cmath_func.wgsl");
        let s2 = include_str!("domain_color.wgsl");
        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl([s1,s2].join("\n").into()),
        });

        // uniform data
        let param_data = vec![0.1, init.config.width as f32, init.config.height as f32,
            select, select_color];
        let frag_uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Fragment Uniform Buffer"),
            contents: bytemuck::cast_slice(&param_data),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });

        let uniform_bind_group_layout = init.device.create_bind_group_layout(
            &wgpu::BindGroupLayoutDescriptor{
                entries: &[
                    wgpu::BindGroupLayoutEntry {
                        binding: 0,
                        visibility: wgpu::ShaderStages::FRAGMENT,
                        ty: wgpu::BindingType::Buffer {
                            ty: wgpu::BufferBindingType::Uniform,
                            has_dynamic_offset: false,
                            min_binding_size: None,
                        },
                        count: None,
                    },
                ],
                label: Some("Uniform Bind Group Layout"),
            });
    }

    let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
        layout: &uniform_bind_group_layout,
        entries: &[
            wgpu::BindGroupEntry {
                binding: 0,
                resource: frag_uniform_buffer.as_entire_binding(),
            },
        ],
        label: Some("Uniform Bind Group"),
    });
}

```

```

});
```

```

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
```

```

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleStrip,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
```

```

    Self {
        init,
        pipeline,
        uniform_bind_group,
        uniform_buffer: frag_uniform_buffer,
        max_iter: 2,
        start,
        t0: 0.0,
        select,
        select_color,
    }
}
```

```

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        let param_data = vec![0.1, self.init.config.width as f32, self.init.config.height as f32,
            self.select, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));
    }
}

```

```

        }
    }

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {

    let t = self.start.elapsed().unwrap().as_millis() as f32;
    let dt = t - self.t0;
    if dt >= 10.0 {
        let a = 100;
        let m = (self.max_iter - a)%(4*a);
        let m_iter = (m as f32 - 2.0*a as f32).abs();

        let param_data = vec![m_iter as f32/100.0, self.init.config.width as f32,
            self.init.config.height as f32, self.select, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));

        self.max_iter += 1;
        self.t0 = t;
    }
    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: None,
        });

        render_pass.set_pipeline(&self.pipeline);
        render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
        render_pass.draw(0..4, 0..1);
    }
}

```

## 396 | Practical GPU Graphics with wgpu and Rust

```
        }

        self.init.queue.submit(iter::once(encoder.finish()));
        output.present();

        Ok(())
    }
}

fn main() {
    let mut select = "0";
    let mut select_color = "0";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        select = &args[1];
    }
    if args.len() > 2 {
        select_color = &args[2];
    }
    let select_id = select.parse::<f32>();
    let color_id = select_color.parse::<f32>();

    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}", "ch14-domain-coloring"));
    let mut state = pollster::block_on(State::new(&window, select_id.unwrap(), color_id.unwrap()));

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput {
                            input:
                                KeyboardInput {
                                    state: ElementState::Pressed,
                                    virtual_keycode: Some(VirtualKeyCode::Escape),
                                    ..
                                },
                            ..
                        } => *control_flow = ControlFlow::Exit,
                        WindowEvent::Resized(physical_size) => {
                            state.resize(*physical_size);
                        }
                        WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                            state.resize(**new_inner_size);
                        }
                        _ => {}
                    }
                }
            }
            Event::RedrawRequested(_) => {
                state.update();
                match state.render() {
```

```

Ok(_) => {}
Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
Err(e) => eprintln!(":{}?", e),
}
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
);
}

```

You can see that this code does not contain vertex data or a vertex buffer. For our implementation of domain coloring in *wgpu*, we will consider a rectangular region of pixels on the screen, created directly in the vertex shader, which will act as a discretized domain for our complex function. Every pixel within the domain is identified with a complex value where the complex function is evaluated. We then calculate the phase of the function and the corresponding color, and finally assign the resulting color to the pixel.

The code includes a uniform buffer for the fragment shader that contains parameters for animated function simulation, window size, function selection, and colormap selection. Inside the *render* function, we update the uniform parameters every 10 milliseconds, producing an animation for our function. Note that the animated simulation parameter will change within the region [0, 2].

Please note that our shader module is defined using two separate shader files: one is *cmath\_func.wgsl*, which contains the commonly used math operations and functions, and the other is *domain\_color.wgsl*, which is specific for our current application and will be coded in the following section. We use the following code snippet to create our shader module with these two shader files:

```

let s1 = include_str!("cmath_func.wgsl");
let s2 = include_str!("domain_color.wgsl");
let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
    label: Some("Shader"),
    source: wgpu::ShaderSource::Wgsl([s1,s2].join("\n").into()),
});

```

### 14.3.2 Shader Code

The shader code in this example is more complicated because we are now performing all of the calculations in the shader program. Add a new *domain\_color.wgsl* file to the *examples/ch14/* folder with the following code:

```

// define complex functions
fn c_func(z:vec2<f32>, a:f32, select_id:i32) -> vec2<f32>{
    var fz:vec2<f32> = z;
    if (select_id == 0) {
        let f1 = z - vec2<f32>(a, 0.0);
        let f2 = c_mul(z,z) + z + vec2<f32>(a, 0.0);
        fz = c_div(f1, f2);
    } elseif (select_id == 1) {
        fz = c_sqrt(c_div(c_log(vec2<f32>(-z.y - 3.0*a, z.x)), c_log(vec2<f32>(-z.y + a, z.x))));
    } elseif (select_id == 2){
        fz = a*c_sin(a*z);
    } elseif(select_id == 3){

```

## 398 | Practical GPU Graphics with wgpu and Rust

```

        fz = (a+0.5)*c_tan(c_tan((a+0.5)*z));
    } elseif(select_id == 4){
        fz = a*c_tan(c_sin((a+0.5)*z));
    } elseif (select_id == 5){
        fz = c_sqrt(vec2<f32>(a + z.x, z.y)) + c_sqrt(vec2<f32>(a - z.x, -z.y));
    } elseif (select_id == 6){
        fz = c_div(c_tan(c_exp2((0.5+a)*z)), z);
    } elseif (select_id == 7){
        fz = c_div(c_sin(c_cos(c_sin((a+0.5)*z))), c_mul(z,z) - a);
    } elseif (select_id == 8){
        fz = a*c_inv(vec2<f32>(1.0, 0.0) + c_mul((a+0.5)*z, c_exp2(c_exp2((a+0.5)*z))));
    } elseif (select_id == 9){
        fz = c_div(c_sin((a+0.5)*z), c_mul(c_cos(c_exp2((a+0.5)*z)), c_mul(z,z)- vec2<f32>(a*a,0.0)));
    } elseif (select_id == 10) {
        fz = c_inv(z + vec2<f32>(a, 0.0)) + c_inv(z - vec2<f32>(a, 0.0));
    }
    return fz;
}

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>(+1.0, -1.0),
        vec2<f32>(-1.0, +1.0),
        vec2<f32>(+1.0, +1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    a: f32;
    width: f32;
    height: f32;
    select: f32;
    select_color: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms: FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    let a = f_uniforms.a;
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;
    let select_id = i32(f_uniforms.select);
    let color_id = i32(f_uniforms.select_color);
    let scale = 5.0;
    let z = vec2<f32>(scale*(coord_in.x - 0.5*w)/w, -scale*(h/w)*(coord_in.y - 0.5*h)/h);

    let fz = c_func(c, a, select_id);

    if (color_id > 0 && color_id < 12) { // colormaps
        return vec4<f32>(colormap_to_rgb(fz, color_id), 1.0);
    } else { // default hsv to rgb
        return vec4<f32>(hsv_to_rgb(fz), 1.0);
    }
}

```

The code begins with a *c\_func* function that contains ten different complex functions with the default function  $f(z) = z$ . The complex function can be selected by specifying the *select\_id* parameter.

In the vertex shader, we create a four-point array that is used to construct a rectangle based on the primitive *TriangleStrip* specified in the render pipeline in the Rust code. This rectangle is the discretized domain for the complex function as described above.

In the fragment shader, we first pass the animation parameter, size of the window, function selection, and color selection as uniform variables. Inside the *fs\_main* function, each point we compute for the complex function corresponds to a fragment in the color buffer, and its computation does not involve any surrounding points. Consequently, we can compute these values by providing the shader with the location of the fragment. We define the complex variable  $z$  in the fragment shader with the following code snippet:

```
let z = vec2<f32>(scale*(coord_in.x - 0.5*w)/w, -scale*(h/w)*(coord_in.y - 0.5*h)/h);
```

Here, *coord\_in* corresponds to the built-in *position* variable, which is available to the fragment shader and provides the location of the fragment in window coordinates. We construct the complex variable  $z$  using the *vec2<f32>* data type in the shader as mentioned previously. The rasterizer then generates a fragment for each pixel in the color buffer, regardless of whether or not we have sent any colors to the shaders. Using *coord\_in.x* and *coord\_in.y*, we can generate colors for each fragment.

Next, we call the *c\_func* function with the *select\_id* parameter to retrieve the corresponding complex function, and using the default *hsv\_to\_rgb* or *colormap\_to\_rgb* function, we finally create the domain coloring for our complex function  $f_z$ .

By putting the computation-intensive code in the fragment shader, we free up the CPU and take advantage of the speed at which the GPU can perform math operations in parallel. Thus, rather than having to compute the color one pixel at a time as we would have to do using CPU-based Rust code, now we can use the multiple GPU cores to compute up to millions of fragment colors concurrently.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch14_domain_color"
path = "examples/ch14/domain_color.rs"
```

In the following subsections, we will discuss the results of domain coloring for the different complex functions included in the *c\_func* function.

### 14.3.3 Complex function with id = 0

The first complex function with *select\_id* = 0 is

$$f(z) = \frac{z - a}{z^2 + z + a}$$

This function has a zero at  $z_0 = a$  and two poles at

$$z_1 = \frac{-1 + i\sqrt{4a - 1}}{2}, \quad z_2 = \frac{-1 - i\sqrt{4a - 1}}{2}$$

We can run our application to create domain coloring for this complex function by issuing the following *cargo run* commands in the terminal window:

```
cargo run --example ch14_domain_color "0" "0"
cargo run --example ch14_domain_color "0" "1"
cargo run --example ch14_domain_color "0" "2"
cargo run --example ch14_domain_color "0" "4"
```

Here, we use two parameters to select the complex function and the colormap respectively. This will generate domain coloring for this selected complex function using different colormaps, as shown with  $a = 1.0$  in Fig.14-3. Note the behavior of the magnitude contours around the zero (right side) and two poles (left side).



Fig.14-3. Domain coloring for  $(z - 1)/(z^2 + z + 1)$ : (a) default `hsv_to_rgb` color, (b) `jet` colormap, (c) `hsv` colormap, and (d) `cool` colormap.

#### 14.3.4 Complex Function with $\text{id} = 1$

The next complex function, with `select_id = 1`, is expressed as

$$f(z) = \sqrt{\frac{\log(i z - 3a)}{\log(i z + a)}}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "1" "0"
cargo run --example ch14_domain_color "1" "5"
```

This will generate domain coloring for this selected complex function using different colormaps, as shown with  $a = 1.0$  in Fig.14-4. You can see that the default *hsv\_to\_rgb* color provides a more detailed description about the behavior of this complex function than the *spring* colormap does.

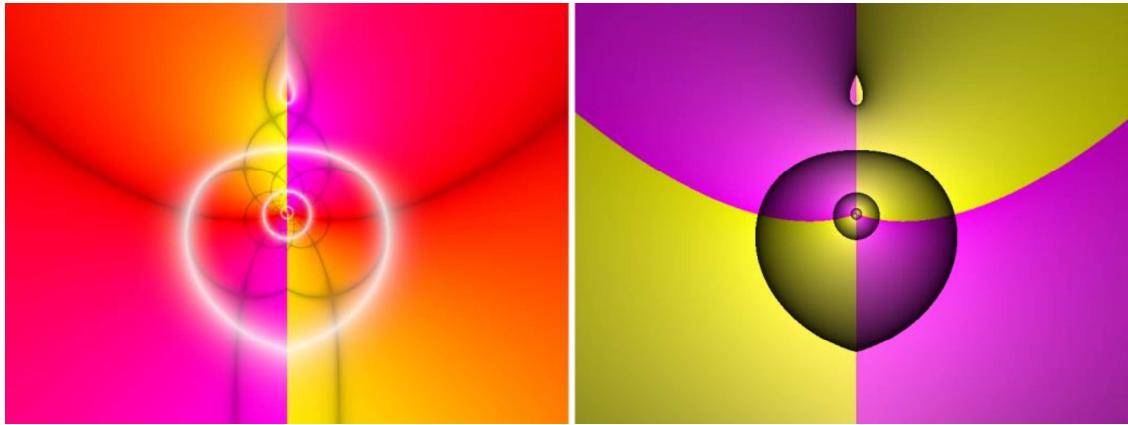


Fig.14-4. Domain coloring for  $\sqrt{\log(iz - 3)/\log(iz + 1)}$  with default (left) and spring (right) colormaps.

### 14.3.5 Complex Function with id = 2

The complex function with *select\_id* = 2 is a straightforward sine function of the complex variable  $z$ :

$$f(z) = a \sin(az)$$

The complex sine function with  $a = 1$  has simple zeros at  $n\pi$  and critical points at  $n\pi + \pi/2$ , where  $n$  is an integer. Along the real axis, we have a real sine function oscillating between  $-1$  and  $+1$ , but the complex sine function is no longer bounded within those limits. In fact, it continues exponentially in the vertical directions:

$$|\sin z| = \sqrt{\sin^2 x + \sinh^2 y}$$

We can run the following commands to create domain coloring for this sine function:

```
cargo run --example ch14_domain_color "2" "0"
cargo run --example ch14_domain_color "2" "2"
```

This will generate domain coloring for this sine function using different colormaps, as shown with  $a = 2.0$  in Fig.14-5.

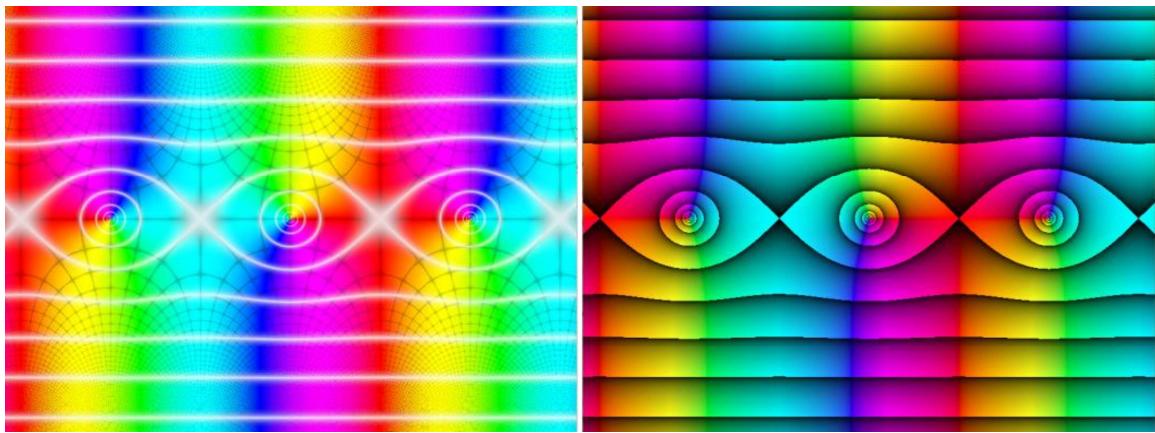


Fig.14.5. Domain coloring for  $2\sin(2z)$  with default (left) and hsv (right) colormaps.

### 14.3.6 Complex Function with id = 3

The next complex function, with `select_id` = 3, is given by:

$$f(z) = a \tan[\tan(az)]$$

This is a tangent of the tangent function of a complex variable  $z$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "3" "0"
cargo run --example ch14_domain_color "3" "6"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-6.

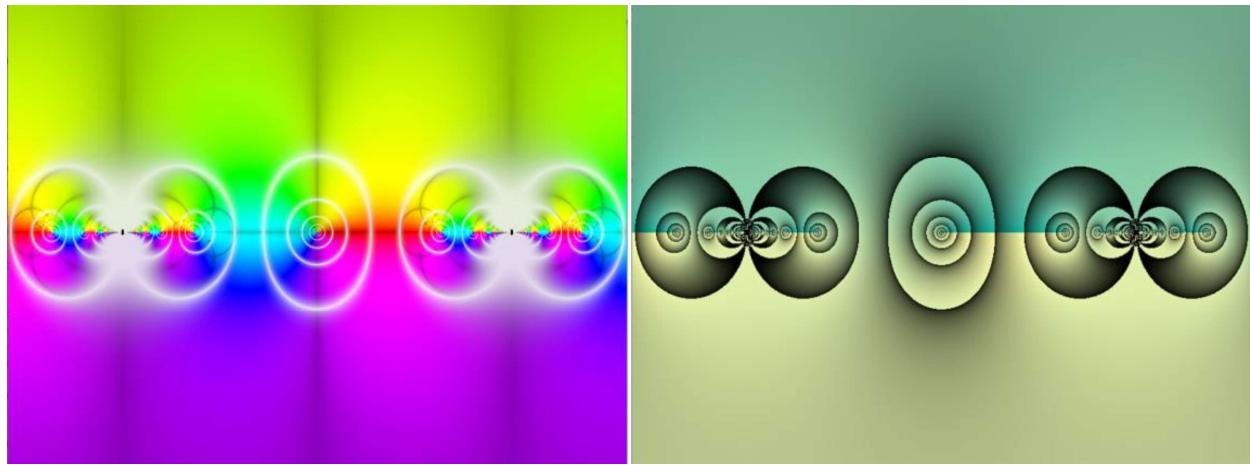


Fig.14-6. Domain coloring for  $\tan(\tan(z))$  with default (left) and summer (right) colormaps.

### 14.3.7 Complex Function with id = 4

The next complex function, with `select_id` = 4, is given by:

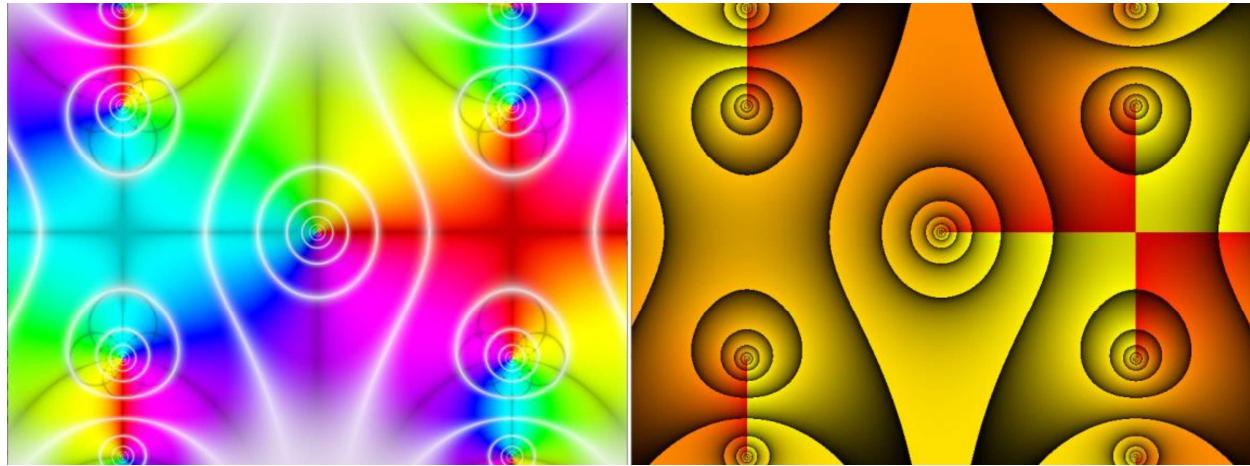
$$f(z) = a \tan[\sin(az)]$$

This is a tangent of the sine function of a complex variable  $z$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "4" "0"
cargo run --example ch14_domain_color "4" "7"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-7.



*Fig.14-7. Domain coloring for  $\tan(\sin(z))$  with default (left) and autumn (right) colormaps.*

### 14.3.8 Complex Function with $id = 5$

The next complex function, with  $select\_id = 5$ , is given by:

$$f(z) = \sqrt{a + z} + \sqrt{a - z}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "5" "0"
cargo run --example ch14_domain_color "5" "2"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-8.

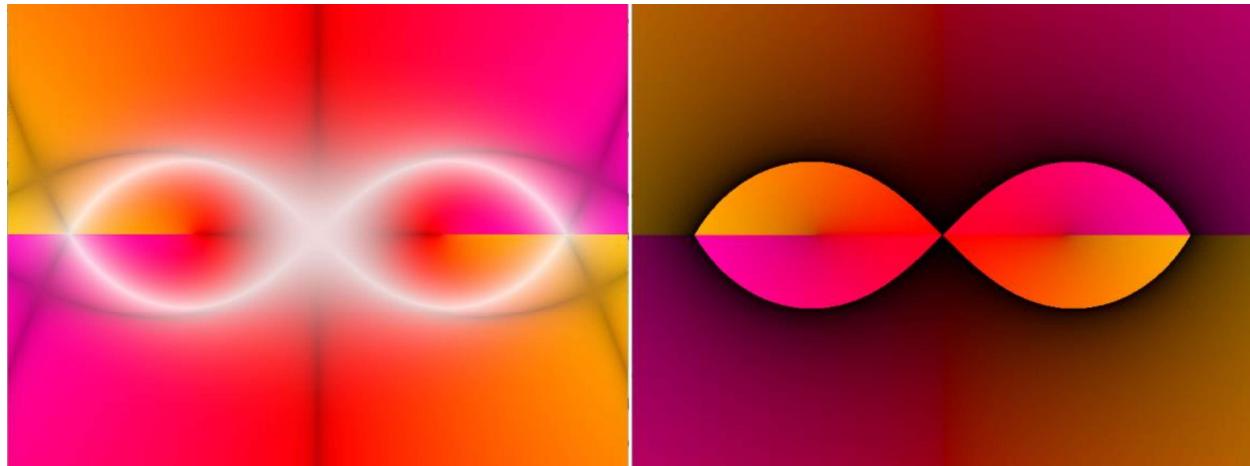


Fig.14-8. Domain coloring for  $\sqrt{1+z} + \sqrt{1-z}$  with default (left) and hsv (right) colormaps.

### 14.3.9 Complex Function with id = 6

The next complex function, with `select_id` = 6, is given by:

$$f(z) = \frac{\tan(az)^2}{z}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "6" "0"
cargo run --example ch14_domain_color "6" "1"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-9.

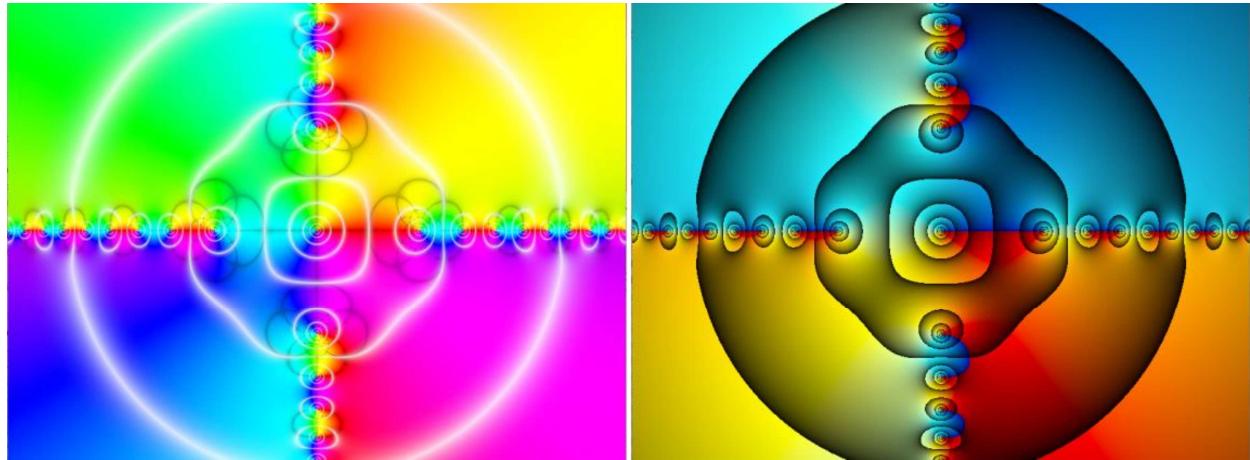


Fig.14-9. Domain coloring for  $\tan(z^2)/z$  with default (left) and jet (right) colormaps.

### 14.3.10 Complex Function with id = 7

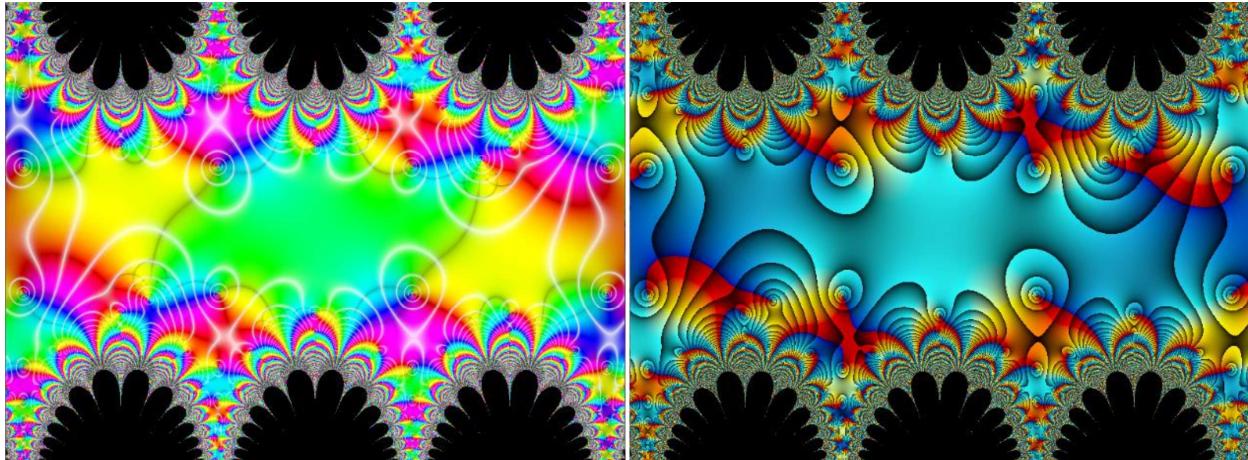
The next complex function, with `select_id` = 7, is given by:

$$f(z) = \frac{\sin(\cos(\sin az))}{z^2 - a}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "7" "0"
cargo run --example ch14_domain_color "7" "1"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-10.



*Fig.14-10. Domain coloring for  $\sin(\cos(\sin(2z)))/(z^2 - 2)$  with default (left) and jet (right) colormaps.*

### 14.3.11 Complex Function with id = 8

The next complex function, with `select_id` = 8, is given by:

$$f(z) = \frac{a}{a^5 z^5 + 1}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "8" "0"
cargo run --example ch14_domain_color "8" "1"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-11.

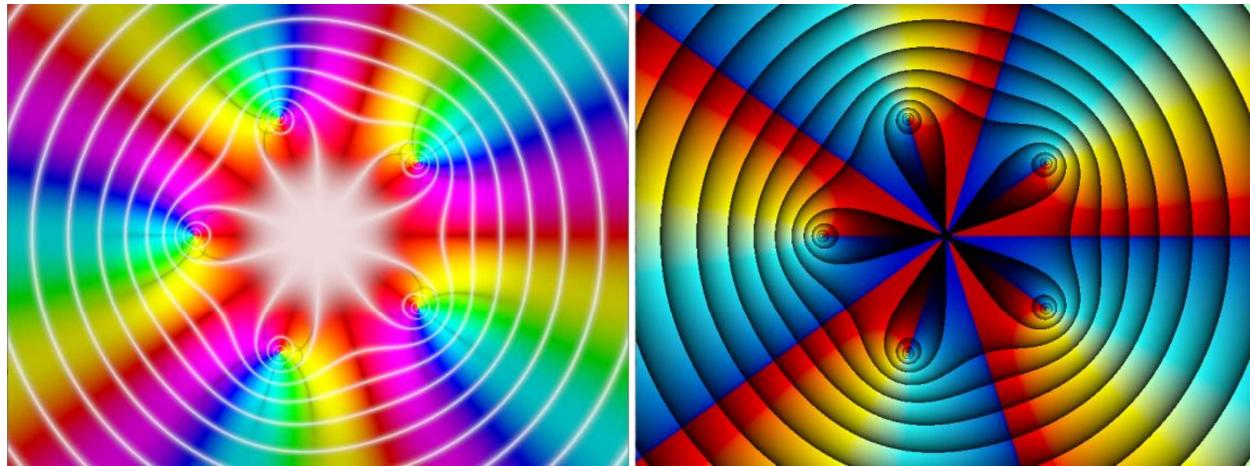


Fig.14-11. Domain coloring for  $1/(z^5 + 1)$  with default (left) and jet (right) colormaps.

### 14.3.12 Complex Function with id = 9

The next complex function, with `select_id` = 9, is given by:

$$f(z) = \frac{\sin(az)}{(z^2 - a^2) \cos(az)^2}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "9" "0"
cargo run --example ch14_domain_color "9" "1"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-12.

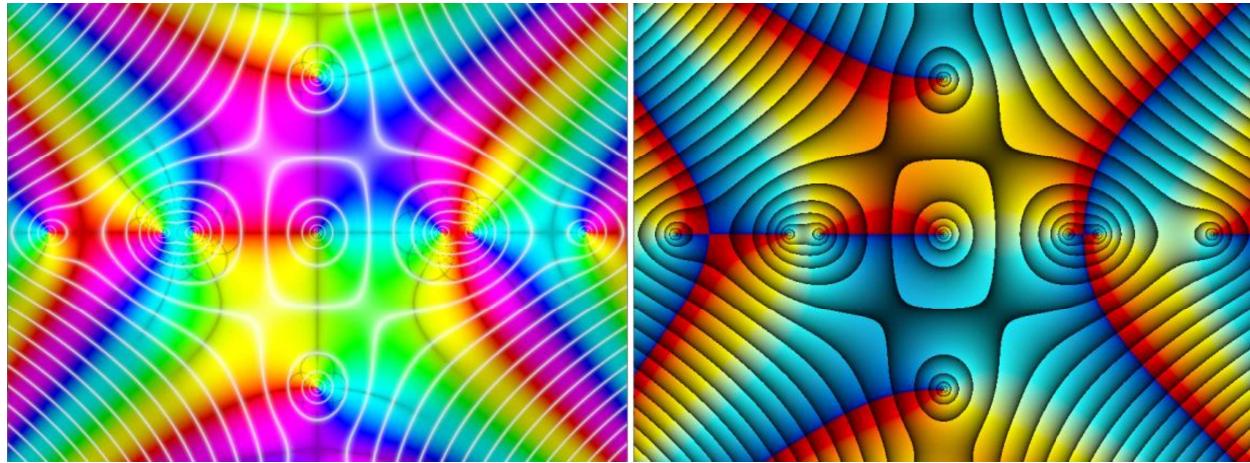


Fig.14-12. Domain coloring for  $\sin z / [(z^2 - 1) \cos z^2]$  with default (left) and jet (right) colormaps.

### 14.3.13 Complex Function with id = 10

The final complex function, with `select_id` = 10, is given by:

$$f(z) = \frac{1}{z+a} + \frac{1}{z-a}$$

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_domain_color "10" "0"
cargo run --example ch14_domain_color "10" "1"
```

This will generate domain coloring for this function using different colormaps, as shown with  $a = 1.0$  in Fig.14-13.

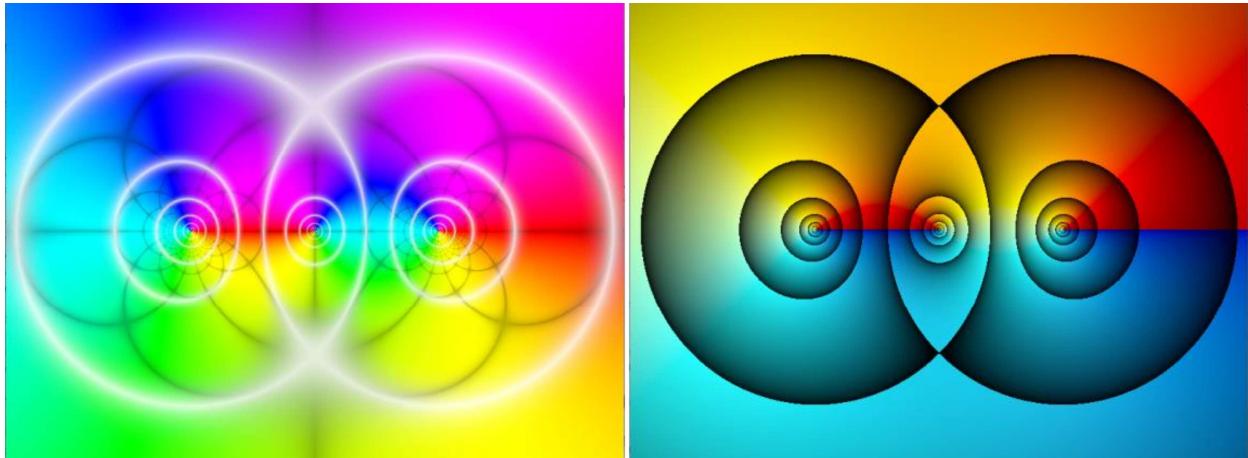


Fig.14-13. Domain coloring for  $\frac{1}{z+1} + \frac{1}{z-1}$  with default (left) and jet (right) colormaps.

## 14.4 Domain Coloring for Iterated Functions

One interesting type of complex function can be obtained by iterating analytic complex functions. We will use the notation  $f^{(n)}(z)$  to denote the  $n$ -th iteration of  $f$ , that is

$$f^{(1)}(z) = f(z), \quad f^{(2)}(z) = f(f(z)), \quad f^{(3)}(z) = f(f(f(z))), \dots \dots$$

In the following, I will show you how some of these iterated functions can produce very beautiful domain coloring images.

To start with you can copy the `domain_color.rs` Rust file used in the preceding example and rename it to `iterate_func.rs` in the `examples/ch14/` folder. Although the Rust code is identical, we will be using different shader code.

### 14.4.1 Shader Code

The shader code in this example is also complicated because all of the calculations are performed in the shader program. Add a new `iterate_func.wgsl` file to the `examples/ch14/` folder with the following code:

```
// define iterated complex functions
fn c_func(z:vec2<f32>, a:f32, select_id:i32) -> vec2<f32>{
    var fz:vec2<f32> = z;

    if (select_id == 0) {
```

## 408 | Practical GPU Graphics with wgpu and Rust

```

        fz = c_mul(vec2<f32>(a, a), c_log(c_mul(z,z)));
    } elseif (select_id == 1){
        fz = c_div(c_log(c_mul(z,z)-vec2<f32>(0.0, a)), c_exp(c_mul(z,z))-vec2<f32>(a, 0.0));
    } elseif (select_id == 2){
        fz = c_div(c_cos(z), c_sin(c_mul(z,z) - vec2<f32>(0.5*a, 0.0)));
    } elseif (select_id == 3){
        let f1 = c_inv(c_pow(z, 4.0) + vec2<f32>(0.0, 0.1*a));
        fz = c_asinh(c_sin(f1));
    } elseif (select_id == 4){
        let f1 = c_inv(c_pow(z, 6.0) + vec2<f32>(0.0, 0.5*a));
        fz = c_log(c_sin(f1));
    } elseif (select_id == 5){
        let f1 = c_mul(vec2<f32>(0.0,1.0), c_cos(z));
        let f2 = c_sin(c_mul(z,z) - vec2<f32>(a, 0.0));
        fz = c_div(f1, f2);
    } elseif (select_id == 6){
        let f1 = c_cos(c_mul(vec2<f32>(0.0,1.0), z));
        let f2 = c_sin(c_mul(z,z) - vec2<f32>(a, 0.0));
        fz = c_div(f1, f2);
    } elseif (select_id == 7){
        let f1 = c_tan(z);
        let f2 = c_sin(c_pow(z,8.0) - vec2<f32>(0.5*a, 0.0));
        fz = c_div(f1, f2);
    } elseif (select_id == 8){
        fz = c_inv(z) + c_div(c_mul(z,z), c_sin(c_pow(z,2.0) - vec2<f32>(a, 0.0)));
    } elseif (select_id == 9){
        fz = c_conj(z) + c_div(c_mul(z,z), c_sin(c_pow(z,2.0) - vec2<f32>(2.0*a, 0.0)));
    } elseif (select_id == 10){
        fz = c_sqrt(c_mul(vec2<f32>(0.0,1.0), z)) + c_div(c_mul(z,z),
            c_sin(c_pow(z,2.0) - vec2<f32>(2.0*a, 0.0)));
    } else {
        fz = c_mul(vec2<f32>(a, a), c_log(c_mul(z,z)));
    }
    return fz;
}

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>, 4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>( 1.0, -1.0),
        vec2<f32>(-1.0, 1.0),
        vec2<f32>( 1.0, 1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    a: f32;
    width: f32;
    height: f32;
    select: f32;
    select_color: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms : FragUniforms;

[[stage(fragment)]]

```

```

fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    let a = f_uniforms.a;
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;
    let aspect = w/h;
    var select_id:i32 = i32(f_uniforms.select);
    let color_id = i32(f_uniforms.select_color);
    let scale:f32 = 4.0;

    var z:vec2<f32> = vec2<f32>(scale*aspect*(coord_in.x - 0.5*w)/w, -scale*(coord_in.y - 0.5*h)/h);
    var i:i32 = 0;
    var iters:array<i32,11> = array<i32,11>(4,3,4,2,2,5,4,10,6,9,4);
    if(select_id > 10) {select_id = 0;}

    loop {
        if(i >= iters[select_id]) {break;}
        z = c_func(z, a, select_id);
        i = i + 1;
    }

    if (color_id > 0 && color_id < 12) { // colormaps
        return vec4<f32>(colormap_to_rgb(z, color_id), 1.0);
    } else { // default hsv to rgb
        return vec4<f32>(hsv_to_rgb(z), 1.0);
    }
}

```

This code begins with a *c\_func* function that contains eleven different iterated complex functions. The function can be selected by specifying the *select\_id* parameter.

The vertex shader in this example is the same as that used in the preceding example. Like before, we create a four-point array to construct the rectangle that acts as a discretized domain for the complex function.

In the fragment shader, we first pass the animation parameter, size of the window, function selection, and color selection as uniform variables. Inside the *fs\_main* function, we define an array of *i32* type with each element specifying the iteration order for each function included in the *c\_func* function:

```
var iters:array<i32,11> = array<i32,11>(4,3,4,2,2,5,4,10,6,9,4);
```

Next, we use a loop to iterate the selected complex function. Computing such iterations of functions is very computation intensive, but by performing the computations inside the fragment shader, we free up the CPU and take advantage of the speed at which the GPU can carry out math operations in parallel. Thus, rather than computing the color one pixel at a time as we would have to do with CPU-based Rust code, now we can use the multiple GPU cores to compute up to millions of fragment colors concurrently.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch14_iterate_func"
path = "examples/ch14/iterate_func.rs"
```

In the following subsections, we will discuss the results of domain coloring for the different iterated functions included in the *c\_func* function.

### 14.4.2 Iterated Function with id = 0

The first complex function, with `select_id` = 0, is

$$f_0(z) = a(1 + i)\log z^2$$

We will iterate it up to the 4<sup>th</sup> order:  $f_0^{(4)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "0" "0"
cargo run --example ch14_iterate_func "0" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 0.5 in Fig.14-14.

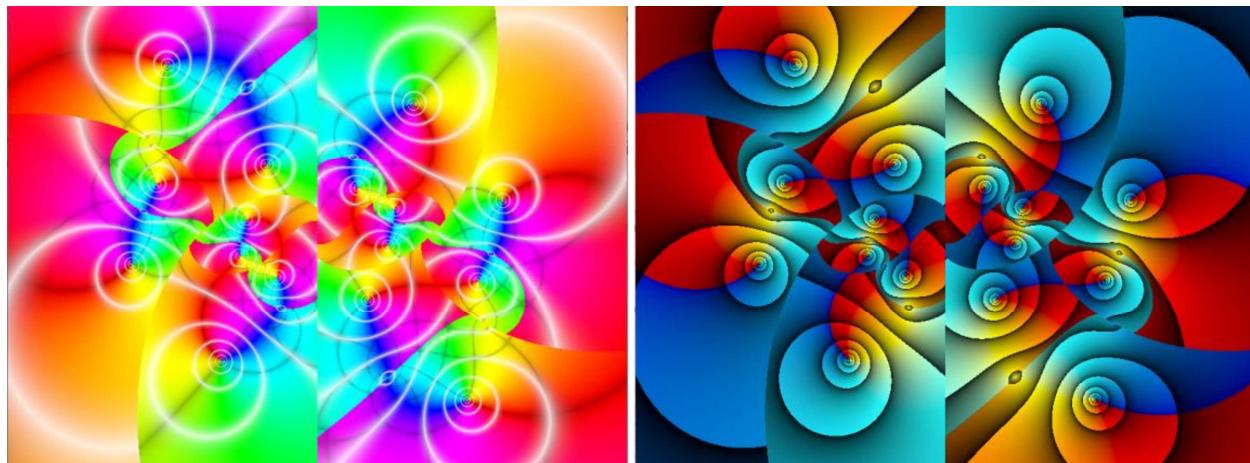


Fig.14-14. Domain coloring for  $f_0^{(4)}(z)$  with default (left) and jet (right) colormaps.

### 14.4.3 Iterated Function with id = 1

The complex function, with `select_id` = 1, is

$$f_1(z) = \frac{\log(z^2 - ia)}{e^{z^2} - a}$$

We will iterate it up to the 3<sup>rd</sup> order:  $f_1^{(3)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "1" "0"
cargo run --example ch14_iterate_func "1" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-15.

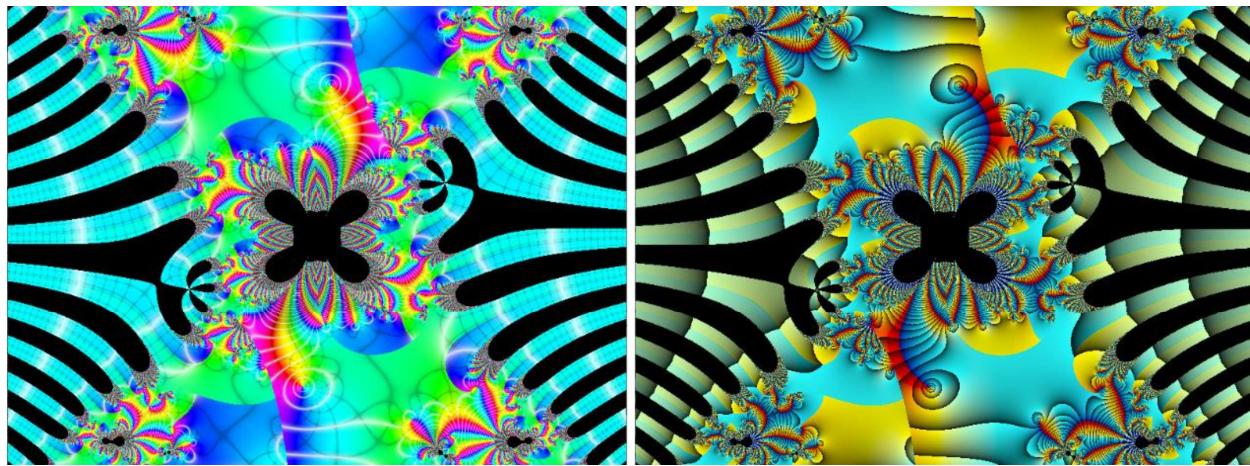


Fig.14-15. Domain coloring for  $f_1^{(3)}(z)$  with default (left) and jet (right) colormaps.

#### 14.4.4 Iterated Function with id = 2

The complex function, with `select_id = 2`, is

$$f_2(z) = \frac{\cos z}{\sin(z^2 - a)}$$

We will iterate it up to the 4<sup>th</sup> order:  $f_2^{(4)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "2" "0"
cargo run --example ch14_iterate_func "2" "2"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a = 1$  in Fig.14-16.

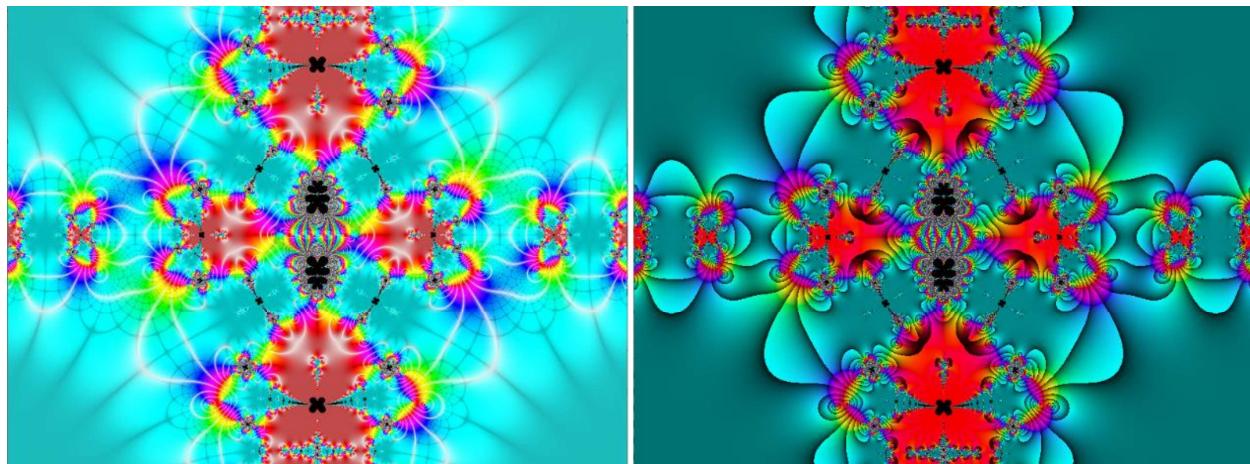


Fig.14-16. Domain coloring for  $f_2^{(4)}(z)$  with default (left) and hsv (right) colormaps.

### 14.4.5 Iterated Function with id = 3

The complex function with, `select_id` = 3, is

$$f_3(z) = \frac{1}{z^4 + ia}$$

We will iterate it up to the 2<sup>nd</sup> order:  $f_3^{(2)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "3" "0"
cargo run --example ch14_iterate_func "3" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 0.1 in Fig.14-17.

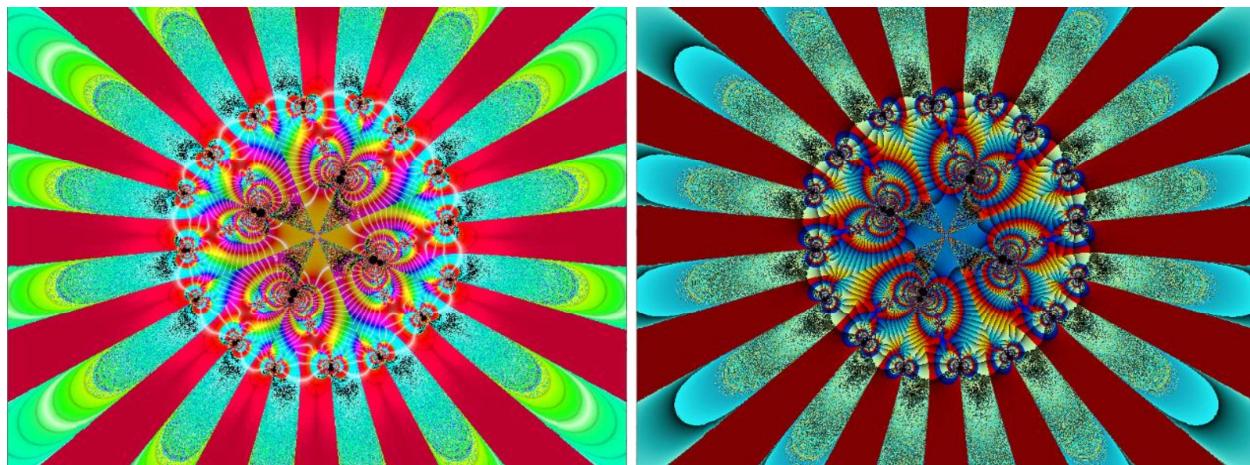


Fig.14-17. Domain coloring for  $f_3^{(2)}(z)$  with default (left) and jet (right) colormaps.

### 14.4.6 Iterated Function with id = 4

The complex function with, `select_id` = 4, is

$$f_4(z) = \frac{1}{z^6 + ia}$$

We will iterate it up to the 2<sup>nd</sup> order:  $f_4^{(2)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "4" "0"
cargo run --example ch14_iterate_func "4" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 0.5 in Fig.14-18.

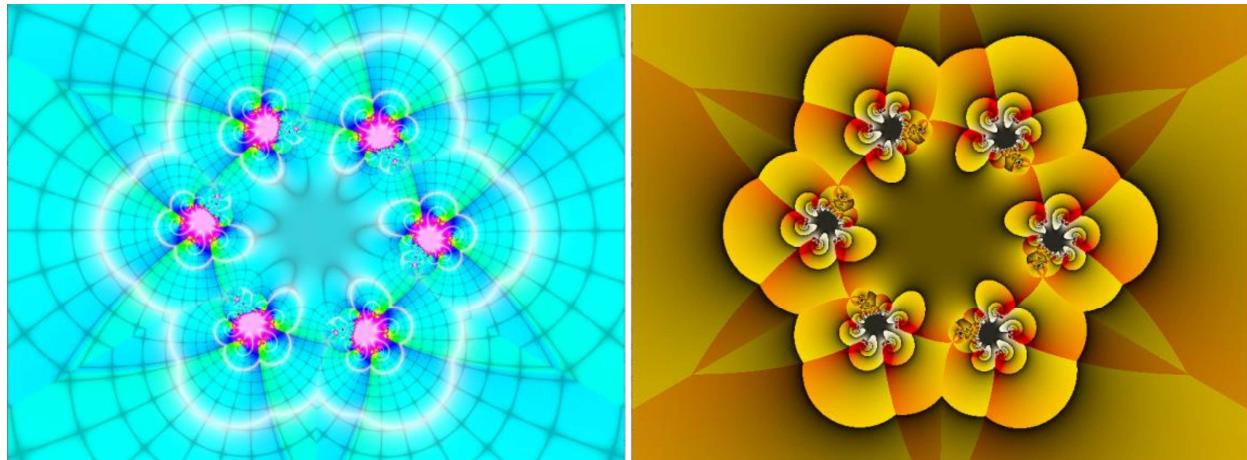


Fig.14-18. Domain coloring for  $f_4^{(2)}(z)$  with default (left) and hot (right) colormaps.

#### 14.4.7 Iterated Function with id = 5

The complex function, with `select_id = 5`, is

$$f_5(z) = \frac{i \cos z}{\sin(z^2 - a)}$$

We will iterate it up to the 5<sup>th</sup> order:  $f_5^{(5)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "5" "0"
cargo run --example ch14_iterate_func "5" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a = 1$  in Fig.14-19.

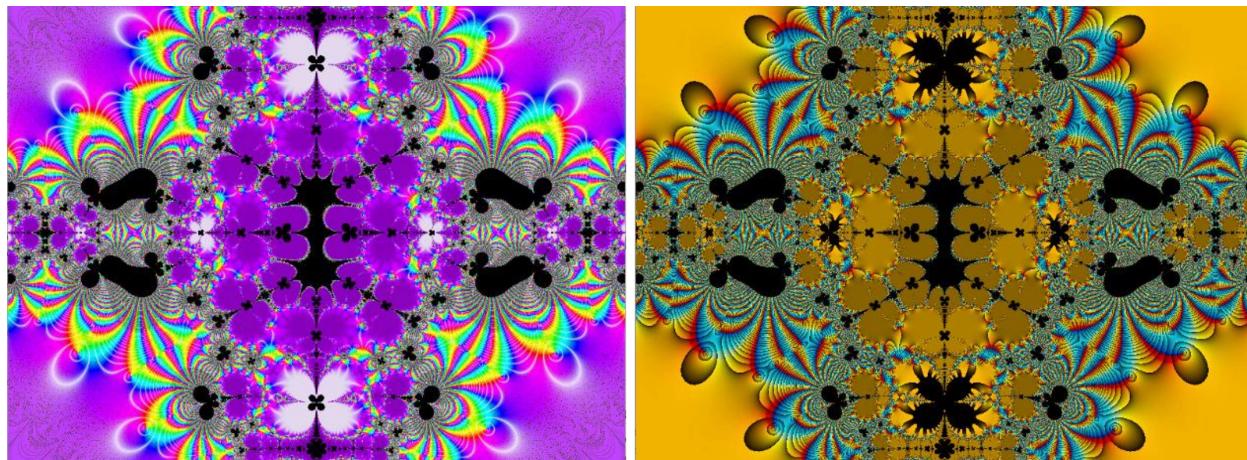


Fig.14-19. Domain coloring for  $f_5^{(5)}(z)$  with default (left) and jet (right) colormaps.

### 14.4.8 Iterated Function with id = 6

The complex function, with `select_id` = 6, is

$$f_6(z) = \frac{\cos(iz)}{\sin(z^2 - a)}$$

We will iterate it up to the 4<sup>th</sup> order:  $f_6^{(4)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "6" "0"
cargo run --example ch14_iterate_func "6" "4"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-20.

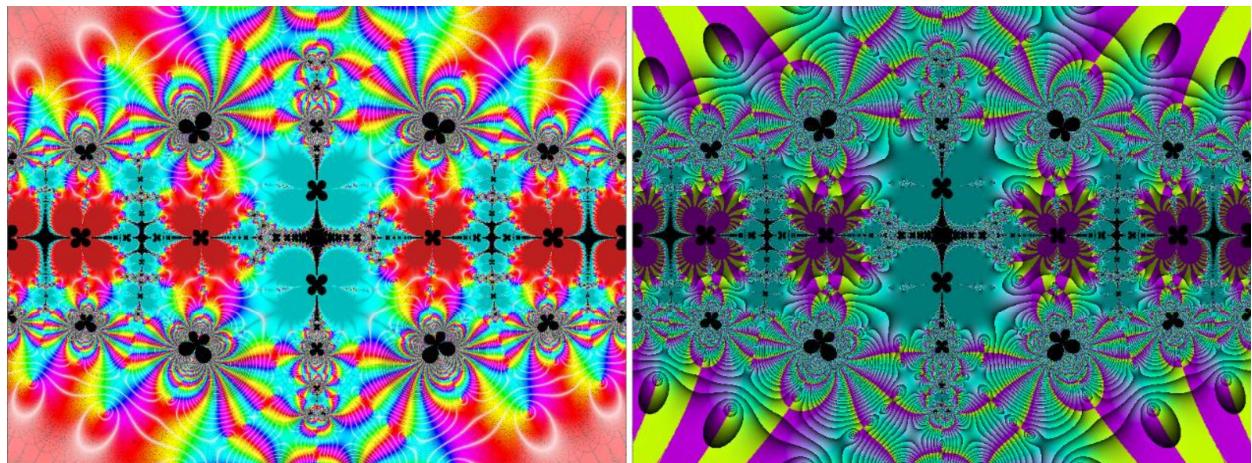


Fig.14-20. Domain coloring for  $f_6^{(4)}(z)$  with default (left) and cool (right) colormaps.

### 14.4.9 Iterated Function with id = 7

The complex function, with `select_id` = 7, is

$$f_7(z) = \frac{\tan(z)}{\sin(z^8 - a)}$$

We will iterate it up to the 10<sup>th</sup> order:  $f_7^{(10)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "7" "0"
cargo run --example ch14_iterate_func "7" "3"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-21.

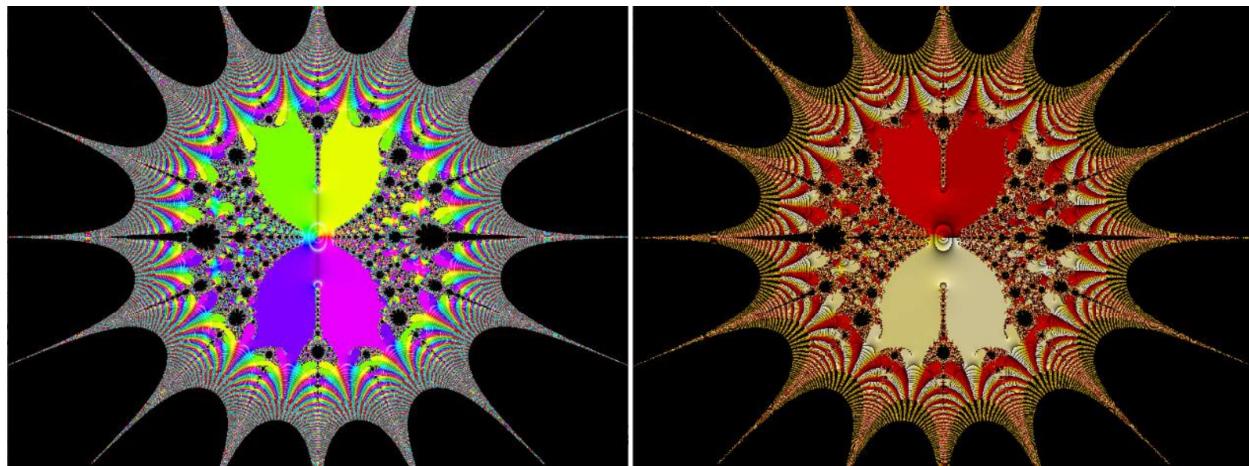


Fig.14-21. Domain coloring for  $f_7^{(10)}(z)$  with default (left) and hot (right) colormaps.

#### 14.4.10 Iterated Function with id = 8

The complex function, with `select_id` = 8, is

$$f_8(z) = \frac{1}{z} + \frac{z^2}{\sin(z^2 - a)}$$

We will iterate it up to the 6<sup>th</sup> order:  $f_8^{(6)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "8" "0"
cargo run --example ch14_iterate_func "8" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-22.

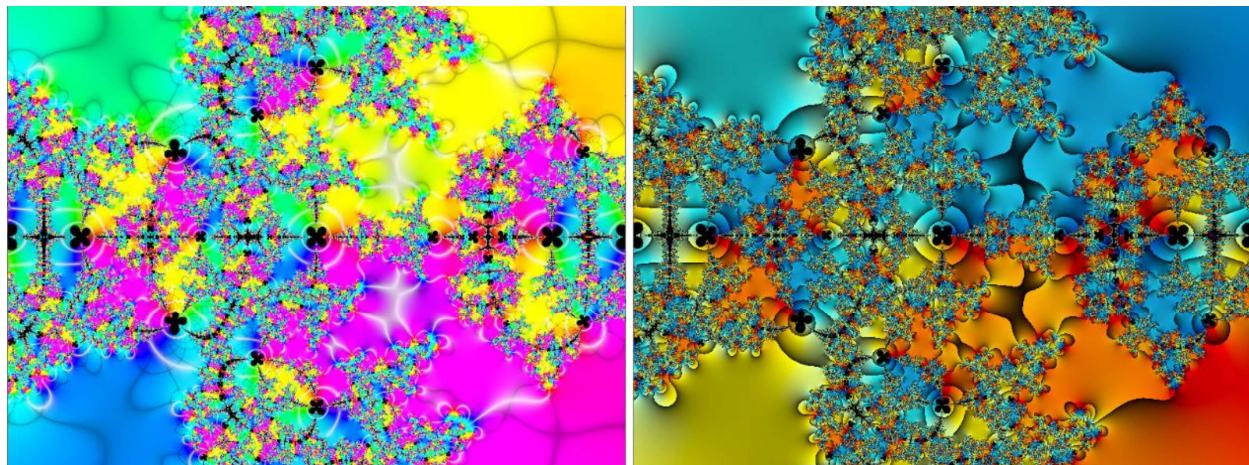


Fig.14-22. Domain coloring for  $f_8^{(6)}(z)$  with default (left) and jet (right) colormaps.

### 14.4.11 Iterated Function with id = 9

The complex function, with `select_id` = 9, is

$$f_9(z) = (x - iy) + \frac{z^2}{\sin(z^2 - a)}$$

We will iterate it up to the 9<sup>th</sup> order:  $f_9^{(9)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "9" "0"
cargo run --example ch14_iterate_func "9" "3"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-23.

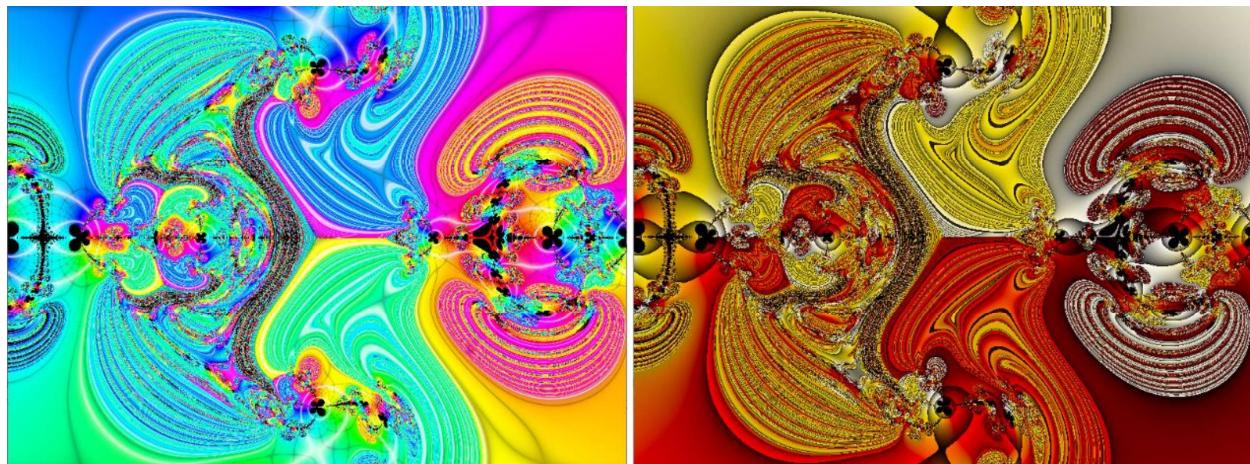


Fig.14-23. Domain coloring for  $f_9^{(9)}(z)$  with default (left) and hot (right) colormaps.

### 14.4.12 Iterated Function with id = 10

The final complex function, with `select_id` = 10, is

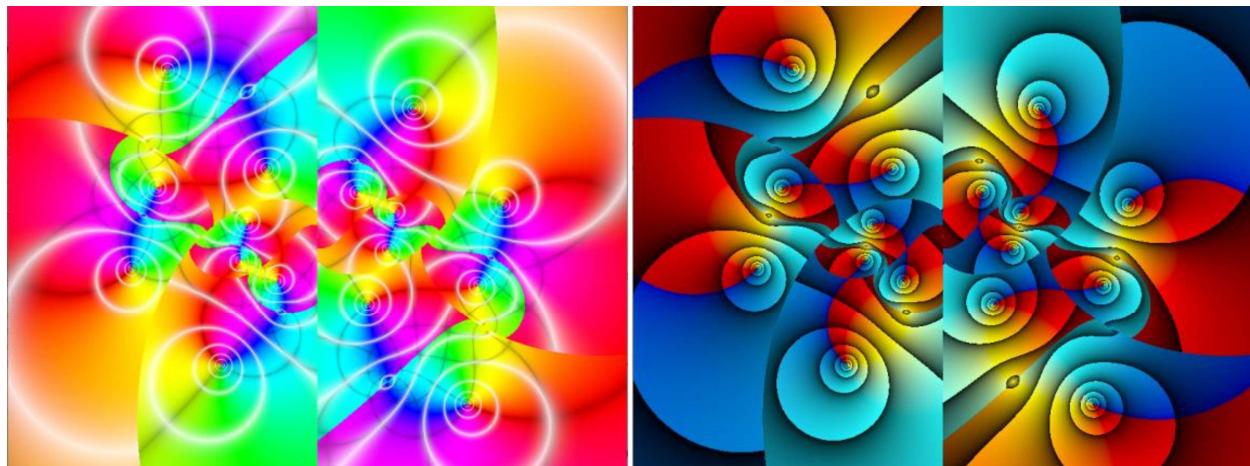
$$f_{10}(z) = \sqrt{iz} + \frac{z^2}{\sin(z^2 - a)}$$

We will iterate it up to the 4<sup>th</sup> order:  $f_{10}^{(4)}(z)$ .

We can run the following commands to create domain coloring for this function:

```
cargo run --example ch14_iterate_func "10" "0"
cargo run --example ch14_iterate_func "10" "1"
```

This will generate domain coloring for this iterated function using different colormaps, as shown with  $a$  = 1 in Fig.14-24.



*Fig. 14-24. Domain coloring for  $f_{10}^{(4)}(z)$  with default (left) and jet (right) colormaps.*

## 14.5 Fractal: Mandelbrot Set

A fractal is a very unusual object with fractional dimension. Fractals are currently a hot topic in mathematical theory and physics. You have probably seen fractals at some point or another: they are often as colorful figures in math and chaos physics books, frequently with zoomed-in view on a region of the picture to demonstrate an interesting property of fractals called self-similarity. That is, no matter how much you zoom in on a part of the fractal, you will still see the same pattern repeating over and over again.

In this section, you will learn how to create a popular fractal, the Mandelbrot set. The Mandelbrot set is probably the most widely recognized fractal. This is not just because it looks beautiful, but also because its elaborate beauty and complexity arises from a simple definition.

### 14.5.1 Mandelbrot Set Formula

The Mandelbrot set is a set of points in the complex plane that forms a fractal, based on a very simple equation:

$$z_{n+1} = z_n^2 + c$$

Where  $z$  and  $c$  are complex numbers. Now, we just need to iterate this function and see which values remain bounded in

$$|z_{n+1}| \leq 2$$

As we did when creating domain coloring for complex functions, we will use the fragment shader to do all of the computation for our Mandelbrot set. In this case, each point we compute for the Mandelbrot set corresponds to a fragment in the color buffer, and its calculation does not involve any surrounding points. Thus, we can compute these values in the fragment shader by providing the shader with the location of the fragment.

## 14.5.2 Rust Code

Add a new Rust file called *mandelbrot.rs* to the *examples/ch14/* folder and enter the following content into it:

```
use std:: {iter, time::SystemTime};
use wgpu::util::DeviceExt;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::{Window, WindowBuilder},
};
use bytemuck:: {Pod, Zeroable };

#[path = "../common/transforms.rs"]
mod transforms;

#[repr(C)]
#[derive(Clone, Copy, Pod, Zeroable)]
struct Vertex {
    pos: [f32; 3],
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    uniform_bind_group: wgpu::BindGroup,
    uniform_buffer: wgpu::Buffer,
    cx: f32,
    cy: f32,
    max_iter: i32,
    start: SystemTime,
    t0: f32,
    select_color: f32,
}

impl State {
    async fn new(window: &Window, select_color: f32, cx: f32, cy: f32) -> Self {
        let start = SystemTime::now();
        let init = transforms::InitWgpu::init_wgpu(window).await;

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: Some("Shader"),
            source: wgpu::ShaderSource::Wgsl(include_str!("mandelbrot.wgsl").into()),
        });

        // uniform data
        let param_data = vec![10.0, cx, cy, init.config.width as f32, init.config.height as f32, select_color];
        let frag_uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Fragment Uniform Buffer"),
            contents: bytemuck::cast_slice(&param_data),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });

        let uniform_bind_group_layout = init.device.create_bind_group_layout(
            &wgpu::BindGroupLayoutDescriptor{
                entries: &[
                    wgpu::BindGroupLayoutEntry {
```

```

        binding: 0,
        visibility: wgpu::ShaderStages::FRAGMENT,
        ty: wgpu::BindingType::Buffer {
            ty: wgpu::BufferBindingType::Uniform,
            has_dynamic_offset: false,
            min_binding_size: None,
        },
        count: None,
    },
],
label: Some("Uniform Bind Group Layout"),
});

let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: frag_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});
}

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
}

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleStrip,
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
});
}

Self {

```

```

        init,
        pipeline,
        uniform_bind_group,
        uniform_buffer: frag_uniform_buffer,
        cx,
        cy,
        max_iter: 2,
        start,
        t0: 0.0,
        select_color,
    }
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        let param_data = vec![10.0, self.cx, self.cy, new_size.width as f32,
            new_size.height as f32, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {

    let t = self.start.elapsed().unwrap().as_millis() as f32;
    let dt = t - self.t0;
    if dt >= 20.0 {
        let a = 50;
        let m = (self.max_iter - a)%(4*a);
        let m_iter = (m as f32 - 2.0*a as f32).abs();
        let param_data = vec![m_iter, self.cx, self.cy, self.init.config.width as f32,
            self.init.config.height as f32, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));

        self.max_iter += 1;
        self.t0 = t;
    }

    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });
}

```

```

    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {
                        r: 0.2,
                        g: 0.247,
                        b: 0.314,
                        a: 1.0,
                    }),
                    store: true,
                },
            }],
            depth_stencil_attachment: None,
        });
        render_pass.set_pipeline(&self.pipeline);
        render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
        render_pass.draw(0..4, 0..1);
    }

    self.init.queue.submit(iter::once(encoder.finish()));
    output.present();

    Ok(())
}
}

fn main() {
    let mut select_color = "0";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        select_color = &args[1];
    }
    let color_id = select_color.parse::<f32>();

    let cx = -0.5f32;
    let cy = 0.0f32;
    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&format!("{}", "ch14-mandelbrot"));
    let mut state = pollster::block_on(State::new(&window, color_id.unwrap(), cx, cy));

    event_loop.run(move |event, _, control_flow| {
        match event {
            Event::WindowEvent {
                ref event,
                window_id,
            } if window_id == window.id() => {
                if !state.input(event) {
                    match event {
                        WindowEvent::CloseRequested
                        | WindowEvent::KeyboardInput { input:

```

```

        KeyboardInput {
            state: ElementState::Pressed,
            virtual_keycode: Some(VirtualKeyCode::Escape),
            ..
        },
        ..
    } => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
    state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
    state.resize(**new_inner_size);
}
_ => {}
}
}
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!(":{}:", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});
}
}

```

This code is similar to that used in the preceding examples, except for the uniform parameters. In the current example, the uniform parameters include the maximum iterations, the initial complex value for the Mandelbrot set, the window size, and the color selection. Inside the `render` function, we will animate the maximum iterations continuously from 2 to 200 and then from 200 to 2.

### 14.5.3 Shader Code

Add a new shader file named `mandelbrot.wgsl` to the `examples/ch14/` folder with the following code:

```

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>, 4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>(+1.0, -1.0),
        vec2<f32>(-1.0, +1.0),
        vec2<f32>(+1.0, +1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    max_iter: f32;
}

```

```

cx: f32;
cy: f32;
width: f32;
height: f32;
select_color: f32;
};

[[binding(0), group(0)]] var<uniform> f_uniforms : FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    let pi:f32 = 3.1415926;
    let max_iter = i32(f_uniforms.max_iter);
    let cx = f_uniforms.cx;
    let cy = f_uniforms.cy;
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;
    let color_id = i32(f_uniforms.select_color);
    let aspect = w/h;
    let scale: f32 = 2.2;

    var x: f32 = scale*aspect*(coord_in.x - 0.5*w)/w + cx;
    var y: f32 = -scale*(coord_in.y - 0.5*h)/h - cy;

    var z: vec2<f32> = vec2<f32>(0.0, 0.0);
    var i:i32 = 0;

    loop {
        if(i >= max_iter) {break;}
        z = vec2<f32>(z.x*z.x - z.y*z.y + x, 2.0*z.x*z.y + y);
        i = i + 1;
        if(z.x*z.x + z.y*z.y > 4.0) {break;}
    }

    var v: f32;
    if(color_id == 1){
        v = fract(log2(f32(i)));
        v = clamp(0.0, v, 1.0);
        return vec4<f32>(v, v*v, v*v*v, 1.0);
    } else {
        v = f32(i)/f_uniforms.max_iter;
        return vec4<f32>(v, 0.5*(sin(5.0*pi*v)+1.0), 0.5*(cos(5.0*pi*v)+1.0), 1.0);
    }
}
}

```

The vertex shader in this example is the same as that used in the preceding example. As before, we create a four-point array to construct the rectangle that acts as a discretized domain for the complex function.

In the fragment shader, we first pass the number of maximum iterations, which also acts as the animation parameter, followed by the initial complex value for the Mandelbrot set, size of the window, and color selection as uniform variables. Inside the `fs_main` function, we start with  $z = \text{vec2}(0.0, 0.0)$  and iterate with a loop. Each iteration puts the value of the next iteration in  $z$  and examines whether it is diverging by checking if  $|z| > 2$ . What we have left at the end of the loop is the number of iterations.

If the initial complex number  $x + iy$  is within the Mandelbrot set, then the function will not diverge, the loop will go to the end, and the number of iterations will reach the number of maximum iterations. Otherwise, if  $x$  and  $y$  are not within the set, the number of iterations will be between 0 and the number of maximum iterations. The closer  $x$  and  $y$  are to the set, the higher the number of iterations will be. Therefore, we can use the number of iterations to color our Mandelbrot set.

Here, we create two sets of color schemes using the number of iterations, and can select between the sets by specifying the `color_id` parameter.

Now, add the following code snippet to the `Cargo.toml` file:

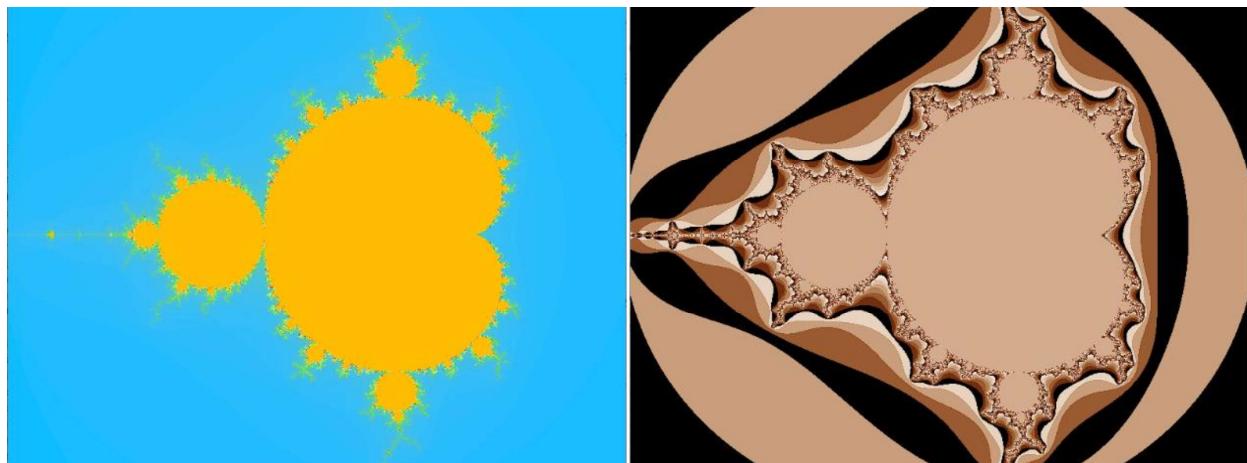
```
[[example]]
name = "ch14_mandelbrot"
path = "examples/ch14/mandelbrot.rs"
```

#### 14.5.4 Run Application

We can now run the following commands to create the Mandelbrot set:

```
cargo run --example ch14_mandelbrot "0"
cargo run --example ch14_mandelbrot "1"
```

This will generate results using two different color schemes, as show with maximum iteration = 200 in Fig.14-25.



*Fig.14-25. Mandelbrot set created using two different color schemes.*

### 14.6 Fractal: Julia Set

Another fractal, the Julia set, is very similar to the Mandelbrot set. In fact, the Mandelbrot set can be considered as a map of all Julia sets – the Mandelbrot set forms a kind of index to the Julia sets. The Julia set uses the same recursion formula as the Mandelbrot set, but instead of  $c$  varying for each point in the complex plane, we set  $c$  to a chosen complex value that is used as a seed. A Julia set is connected if  $c$  chosen from within the Mandelbrot set, while it is disconnected if  $c$  is outside of the Mandelbrot set.

#### 14.6.1 Rust Code

Add a new Rust file called `Julia_set.rs` to the `examples/ch14/` folder and enter the following content into it:

```
use std:: {iter, time::SystemTime};
use wgpu::util::DeviceExt;
use winit:: {
```

```

event::*,  

event_loop::{ControlFlow, EventLoop},  

window::{Window, WindowBuilder},  

};  

use bytemuck:: {Pod, Zeroable };  
  

#[path="../common/transforms.rs"]  

mod transforms;  
  

#[repr(C)]  

#[derive(Clone, Copy, Pod, Zeroable)]  

struct Vertex {  

    pos: [f32; 3],  

}  
  

struct State {  

    init: transforms::InitWgpu,  

    pipeline: wgpu::RenderPipeline,  

    uniform_bind_group: wgpu::BindGroup,  

    uniform_buffer: wgpu::Buffer,  

    cxy: Vec<Vec<f32>>,  

    max_iter: i32,  

    cxy_num: usize,  

    start: SystemTime,  

    t0: f32,  

    select_color: f32,  

}  
  

impl State {  

    async fn new(window: &Window, select_color: f32) -> Self {  

        let init = transforms::InitWgpu::init_wgpu(window).await;  

        let start = SystemTime::now();  

        let cxy = vec![  

            vec![0.3f32, 0.5],  

            vec![0.3, 0.45],  

            vec![0.3, 0.46],  

            vec![0.3, 0.47],  

            vec![0.3, 0.48],  

            vec![0.3, 0.49],  

            vec![0.24, 0.55],  

            vec![0.234, 0.545],  

            vec![0.234, 0.54],  

            vec![0.235, 0.53],  

        ];  
  

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {  

            label: Some("Shader"),  

            source: wgpu::ShaderSource::Wgsl(include_str!("julia_set.wgsl").into()),  

        });  
  

        // uniform data  

        let param_data = vec![2.0, cxy[0][0], cxy[0][1], init.config.width as f32,  

            init.config.height as f32, select_color];  

        let frag_uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {  

            label: Some("Fragment Uniform Buffer"),  

            contents: bytemuck::cast_slice(&param_data),  

            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,  

        });  

    }
}

```

```

let uniform_bind_group_layout = init.device.create_bind_group_layout(
    &wgpu::BindGroupLayoutDescriptor{
        entries: &[
            wgpu::BindGroupLayoutEntry {
                binding: 0,
                visibility: wgpu::ShaderStages::FRAGMENT,
                ty: wgpu::BindingType::Buffer {
                    ty: wgpu::BufferBindingType::Uniform,
                    has_dynamic_offset: false,
                    min_binding_size: None,
                },
                count: None,
            },
        ],
        label: Some("Uniform Bind Group Layout"),
    });
let uniform_bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
    layout: &uniform_bind_group_layout,
    entries: &[
        wgpu::BindGroupEntry {
            binding: 0,
            resource: frag_uniform_buffer.as_entire_binding(),
        },
    ],
    label: Some("Uniform Bind Group"),
});
let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&uniform_bind_group_layout],
    push_constant_ranges: &[],
});
let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],
    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleStrip,
        ..Default::default()
    },
},

```

```

        depth_stencil: None,
        multisample: wgpu::MultisampleState::default(),
    });

Self {
    init,
    pipeline,
    uniform_bind_group,
    uniform_buffer: frag_uniform_buffer,
    max_iter: 2,
    cxy,
    cxy_num: 0,
    start,
    t0: 0.0,
    select_color,
}
}

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);

        let param_data = vec![self.max_iter as f32, self.cxy[self.cxy_num][0],
            self.cxy[self.cxy_num][1], self.init.config.width as f32,
            self.init.config.height as f32, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &WindowEvent) -> bool {
    false
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {

    let t = self.start.elapsed().unwrap().as_millis() as f32;
    let dt = t - self.t0;
    if dt >= 20.0 {
        let param_data = vec![self.max_iter as f32, self.cxy[self.cxy_num][0],
            self.cxy[self.cxy_num][1], self.init.config.width as f32,
            self.init.config.height as f32, self.select_color];
        self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));

        self.max_iter += 1;
        if self.max_iter > 200 {
            self.max_iter = 2;
            self.cxy_num += 1;
        }
        if self.cxy_num >= self.cxy.len() {
            self.cxy_num = 0;
        }
        self.t0 = t;
    }
}
}

```

```

let output = self.init.surface.get_current_texture()?;
let view = output
    .texture
    .create_view(&wgpu::TextureViewDescriptor::default());

let mut encoder = self
    .init.device
    .create_command_encoder(&wgpu::CommandEncoderDescriptor {
        label: Some("Render Encoder"),
    });

{
    let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
        label: Some("Render Pass"),
        color_attachments: &[wgpu::RenderPassColorAttachment {
            view: &view,
            resolve_target: None,
            ops: wgpu::Operations {
                load: wgpu::LoadOp::Clear(wgpu::Color {
                    r: 0.2,
                    g: 0.247,
                    b: 0.314,
                    a: 1.0,
                }),
                store: true,
            },
        }],
        depth_stencil_attachment: None,
    });

    render_pass.set_pipeline(&self.pipeline);
    render_pass.set_bind_group(0, &self.uniform_bind_group, &[]);
    render_pass.draw(0..4, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

fn main() {
    let mut select_color = "0";
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        select_color = &args[1];
    }
    let color_id = select_color.parse::<f32>();

    env_logger::init();
    let event_loop = EventLoop::new();
    let window = WindowBuilder::new().build(&event_loop).unwrap();
    window.set_title(&*format!("{}", "ch14-julia-set"));
    let mut state = pollster::block_on(State::new(&window, color_id.unwrap()));

    event_loop.run(move |event, _, control_flow| {
        match event {

```

```

Event::WindowEvent {
    ref event,
    window_id,
} if window_id == window.id() => {
    if !state.input(event) {
        match event {
            WindowEvent::CloseRequested
            | WindowEvent::KeyboardInput {
                input:
                    KeyboardInput {
                        state: ElementState::Pressed,
                        virtual_keycode: Some(VirtualKeyCode::Escape),
                        ..
                    },
                    ..
            } => *control_flow = ControlFlow::Exit,
            WindowEvent::Resized(physical_size) => {
                state.resize(*physical_size);
            }
            WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
                state.resize(**new_inner_size);
            }
            _ => {}
        }
    }
}
Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {}", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});;
}

```

This code is similar to that used in the preceding example, except that the initial complex values are now represented by *cxy*, a *Vec<Vec<f32>>* data type consisting of ten complex values which will be used to generate different Julia sets.

Inside the *render* function, we will animate the number of maximum iterations continuously from 2 to 200.

## 14.6.2 Shader Code

Add a new shader file called *julia\_set.wgsl* to the *examples/ch14/* folder with the following code:

```

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>, 4>(

```

## 430 | Practical GPU Graphics with wgpu and Rust

```

        vec2<f32>(-1.0, -1.0),
        vec2<f32>(+1.0, -1.0),
        vec2<f32>(-1.0, +1.0),
        vec2<f32>(+1.0, +1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    max_iter: f32;
    cx: f32;
    cy: f32;
    width: f32;
    height: f32;
    select_color: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms : FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    let pi:f32 = 3.1415926;
    let max_iter = i32(f_uniforms.max_iter);
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;
    let color_id = i32(f_uniforms.select_color);
    let aspect = w/h;
    let scale: f32 = 2.2;
    let c = vec2<f32>(f_uniforms.cx, f_uniforms.cy);
    var z: vec2<f32>;
    z.y = scale*aspect*(coord_in.x - 0.5*w)/w;
    z.x = -scale*(coord_in.y - 0.5*h)/h;
    var i:i32 = 0;

    loop {
        if(i >= max_iter) {break;}
        z = vec2<f32>(z.x*z.x - z.y*z.y + c.x, 2.0*z.x*z.y + c.y);
        i = i + 1;
        if(z.x*z.x + z.y*z.y > 4.0) {break;}
    }

    var v: f32;
    if(color_id == 1){
        v = fract(log2(f32(i)));
        v = clamp(0.0, v, 1.0);
        return vec4<f32>(v, v*v, v*v*v, 1.0);
    } else {
        v = f32(i)/f_uniforms.max_iter;
        return vec4<f32>(v, 0.5*(sin(5.0*pi*v*v)+1.0), 0.5*(cos(5.0*pi*v)+1.0), 1.0);
    }
}

```

The vertex shader in this example is the same as that used in the preceding example. The fragment shader is also largely the same, but now we initialize the complex values for the set using a chosen value from *cxy*.

Inside the *fs\_main* function, we start by getting *z* from the current complex value and loop over it until it grows beyond  $|z| > 2$ . This will guarantee that either *z* will go to infinity, or the number of maximum

iterations will be reached. Finally, we create two sets of color schemes using this number of iterations, which we can choose between the sets by specifying the `color_id` parameter.

Now, add the following code snippet to the `Cargo.toml` file:

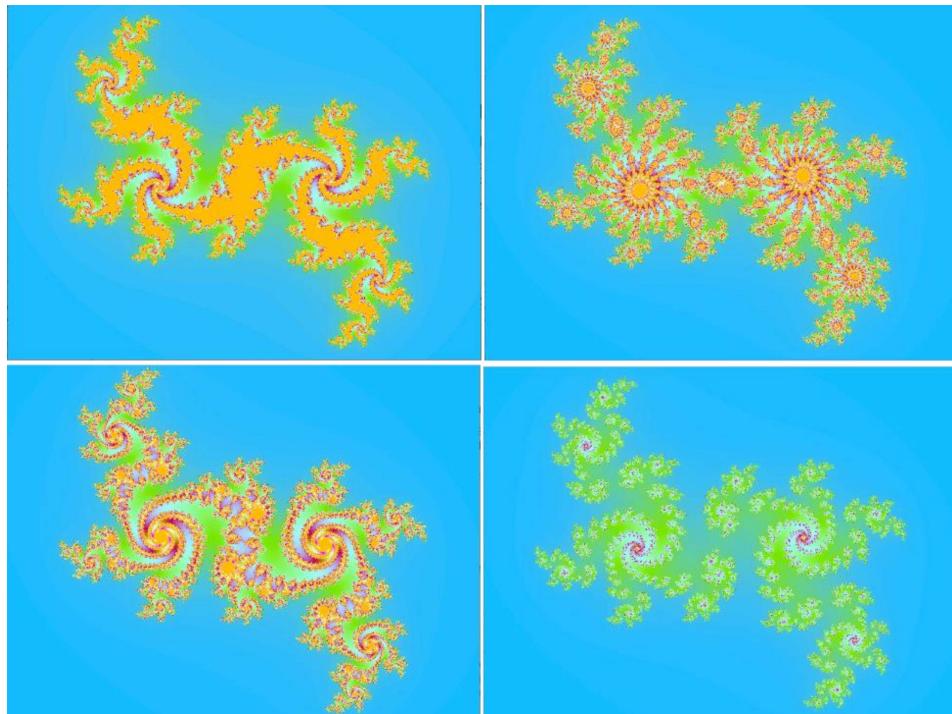
```
[[example]]
name = "ch14_julia_set"
path = "examples/ch14/julia_set.rs"
```

### 14.6.3 Run Application

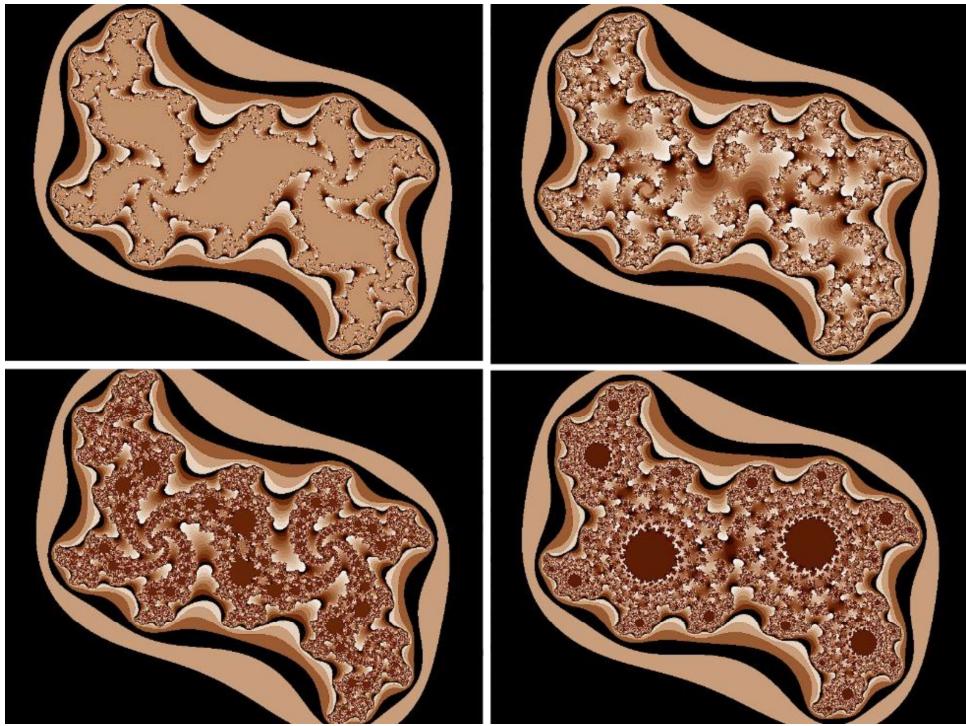
We can now run the following commands to create the Julia set:

```
cargo run --example ch14_julia_set "0"
cargo run --example ch14_julia_set "1"
```

Fig.14-26 and Fig.14-27 show the results for four different initial  $c$  values using the color schemes 0 and 1, respectively.



*Fig.14-26. Julia sets created using the color scheme 0.*



*Fig. 14-27. Julia sets created using the color scheme 1.*

## 14.7 3D Fractals

3D fractals like Mandelbulb and Mandelbox are a projection of the corresponding 2D fractals using 3D spherical coordinate system. Real-time rendering for 3D fractals has a very high computational cost. Here, I will convert three examples of 3D fractals, originally developed for WebGL, into *wgpu* applications. The purpose of these examples is to demonstrate the power and potential of *wgpu* and WGSL.

### 14.7.1 Common Code

To avoid code duplication, we will implement a common file called *fractal3d.rs* in the *examples/ch14/* folder, which we can reuse to create a variety of different 3D fractal images. Enter the following content into the *fractal3d.rs* file:

```
use std:: {iter, time::SystemTime};
use wgpu::util::DeviceExt;
use wgpu::ShaderSource;
use winit::{
    event::*,
    event_loop::{ControlFlow, EventLoop},
    window::Window,
};
use bytemuck:: {Pod, Zeroable };

#[path = "../common/transforms.rs"]
mod transforms;
```

```

#[repr(C)]
#[derive(Clone, Copy, Pod, Zeroable)]
struct Vertex {
    pos: [f32; 3],
}

struct State {
    init: transforms::InitWgpu,
    pipeline: wgpu::RenderPipeline,
    bind_group: wgpu::BindGroup,
    uniform_buffer: wgpu::Buffer,
    start: SystemTime,
    time: f32,
    max_iter: f32,
    mousex: f32,
    mousey: f32,
    mouse_press: bool,
    scale:f32,
    mouse_control: bool,
}

impl State {
    async fn new(window: &Window, shader_source: ShaderSource<'_>, max_iter:f32, scale:f32,
    mouse_control:bool) -> Self {
        let init = transforms::InitWgpu::init_wgpu(window).await;
        let start = SystemTime::now();

        let shader = init.device.create_shader_module(&wgpu::ShaderModuleDescriptor {
            label: None,
            source: shader_source,
        });

        // uniform data
        let param_data = vec![0.0, max_iter, 0.0, 0.0, init.config.width as f32,
            init.config.height as f32, scale as f32];
        let uniform_buffer = init.device.create_buffer_init(&wgpu::util::BufferInitDescriptor {
            label: Some("Uniform Buffer"),
            contents: bytemuck::cast_slice(&param_data),
            usage: wgpu::BufferUsages::UNIFORM | wgpu::BufferUsages::COPY_DST,
        });

        let bind_group_layout = init.device.create_bind_group_layout(&wgpu::BindGroupLayoutDescriptor{
            entries: &[
                wgpu::BindGroupLayoutEntry {
                    binding: 0,
                    visibility: wgpu::ShaderStages::FRAGMENT,
                    ty: wgpu::BindingType::Buffer {
                        ty: wgpu::BufferBindingType::Uniform,
                        has_dynamic_offset: false,
                        min_binding_size: None,
                    },
                    count: None,
                },
            ],
            label: Some("Uniform Bind Group Layout"),
        });

        let bind_group = init.device.create_bind_group(&wgpu::BindGroupDescriptor{
            layout: &bind_group_layout,

```

```

entries: &[
    wgpu::BindGroupEntry {
        binding: 0,
        resource: uniform_buffer.as_entire_binding(),
    },
],
label: Some("Uniform Bind Group"),
});

let pipeline_layout = init.device.create_pipeline_layout(&wgpu::PipelineLayoutDescriptor {
    label: Some("Render Pipeline Layout"),
    bind_group_layouts: &[&bind_group_layout],
    push_constant_ranges: &[],
});

let pipeline = init.device.create_render_pipeline(&wgpu::RenderPipelineDescriptor {
    label: Some("Render Pipeline"),
    layout: Some(&pipeline_layout),
    vertex: wgpu::VertexState {
        module: &shader,
        entry_point: "vs_main",
        buffers: &[],

    },
    fragment: Some(wgpu::FragmentState {
        module: &shader,
        entry_point: "fs_main",
        targets: &[wgpu::ColorTargetState {
            format: init.config.format,
            blend: Some(wgpu::BlendState {
                color: wgpu::BlendComponent::REPLACE,
                alpha: wgpu::BlendComponent::REPLACE,
            }),
            write_mask: wgpu::ColorWrites::ALL,
        }],
    }),
    primitive: wgpu::PrimitiveState{
        topology: wgpu::PrimitiveTopology::TriangleStrip,
        strip_index_format: Some(wgpu::IndexFormat::UInt32),
        ..Default::default()
    },
    depth_stencil: None,
    multisample: wgpu::MultisampleState::default(),
}),
Self {
    init,
    pipeline,
    bind_group,
    uniform_buffer: uniform_buffer,
    start,
    time: 0.0,
    max_iter,
    mousex : 0.0,
    mousey : 0.0,
    mouse_press : false,
    scale: scale as f32,
    mouse_control,
}
}
}

```

```

pub fn resize(&mut self, new_size: winit::dpi::PhysicalSize<u32>) {
    if new_size.width > 0 && new_size.height > 0 {
        self.init.size = new_size;
        self.init.config.width = new_size.width;
        self.init.config.height = new_size.height;
        self.init.surface.configure(&self.init.device, &self.init.config);
    }
}

#[allow(unused_variables)]
fn input(&mut self, event: &DeviceEvent) -> bool {
    match event {
        DeviceEvent::Button {
            button: 1,
            state,
        } => {
            self.mouse_press = *state == ElementState::Pressed;
            true
        }
        DeviceEvent::MouseMove { delta } => {
            if self.mouse_press && self.mouse_control {
                self.mousex = self.mousex + delta.0 as f32;
                self.mousey = self.mousey + delta.1 as f32;
            }
            true
        }
        _ => false,
    }
}

fn update(&mut self) {}

fn render(&mut self) -> Result<(), wgpu::SurfaceError> {

    self.time = self.start.elapsed().unwrap().as_secs_f32();
    let param_data = vec![self.time, self.max_iter, self.mousex, self.mousey,
        self.init.config.width as f32, self.init.config.height as f32, self.scale as f32];
    self.init.queue.write_buffer(&self.uniform_buffer, 0, bytemuck::cast_slice(&param_data));

    let output = self.init.surface.get_current_texture()?;
    let view = output
        .texture
        .create_view(&wgpu::TextureViewDescriptor::default());

    let mut encoder = self
        .init.device
        .create_command_encoder(&wgpu::CommandEncoderDescriptor {
            label: Some("Render Encoder"),
        });
    {
        let mut render_pass = encoder.begin_render_pass(&wgpu::RenderPassDescriptor {
            label: Some("Render Pass"),
            color_attachments: &[wgpu::RenderPassColorAttachment {
                view: &view,
                resolve_target: None,
                ops: wgpu::Operations {
                    load: wgpu::LoadOp::Clear(wgpu::Color {

```

```
        r: 0.2,
        g: 0.247,
        b: 0.314,
        a: 1.0,
    )),
    store: true,
),
],
depth_stencil_attachment: None,
));

render_pass.set_pipeline(&self.pipeline);
render_pass.set_bind_group(0, &self.bind_group, &[]);
render_pass.draw(0..4, 0..1);
}

self.init.queue.submit(iter::once(encoder.finish()));
output.present();

Ok(())
}
}

pub fn run(shader_source: ShaderSource<'_,>, max_iter:f32, scale:f32, mouse_control:bool, title: &str) {
env_logger::init();
let event_loop = EventLoop::new();
let window = winit::window::WindowBuilder::new().build(&event_loop).unwrap();
window.set_title(&format!("ch14_{}", title));
let mut state = pollster::block_on(State::new(&window, shader_source, max_iter, scale,
mouse_control));

event_loop.run(move |event, _, control_flow| {
match event {
Event::DeviceEvent {
ref event,
..
} => {
state.input(event);
}
Event::WindowEvent {
ref event,
window_id,
} if window_id == window.id() => {
match event {
WindowEvent::CloseRequested
| WindowEvent::KeyboardInput {
input:
KeyboardInput {
state: ElementState::Pressed,
virtual_keycode: Some(VirtualKeyCode::Escape),
..
},
..
} => *control_flow = ControlFlow::Exit,
WindowEvent::Resized(physical_size) => {
state.resize(*physical_size);
}
WindowEvent::ScaleFactorChanged { new_inner_size, .. } => {
state.resize(**new_inner_size);
}
}
}
}
}
```

```

        }
        _ => {}
    }
}

Event::RedrawRequested(_) => {
    state.update();
    match state.render() {
        Ok(_) => {}
        Err(wgpu::SurfaceError::Lost) => state.resize(state.init.size),
        Err(wgpu::SurfaceError::OutOfMemory) => *control_flow = ControlFlow::Exit,
        Err(e) => eprintln!("{}: {}", e),
    }
}
Event::MainEventsCleared => {
    window.request_redraw();
}
_ => {}
});;
}
}

```

This code is similar to that used in the preceding examples, except for the uniform parameters. Here, the uniform parameters include the *time*, *max\_iter*, *mousex*, *mousey*, the window size, and the scaling factor. The *time* parameter will be used to animate fractal images, and the mouse movement parameters, *mousex* and *mousey*, allow the user to use mouse to interact with the fractal. Notice that we define *mousex* and *mousey* parameters inside the *input* function.

## 14.7.2 Mandelbulb

Mandelbulb is a 3D fractal, constructed in the same way as the Mandelbrot set, but using 3D vectors instead of 2D complex numbers. White and Nylander's formula for the  $n^{th}$  power of a 3D vector  $v = (x, y, z)$  can be expressed in the form:

$$v^n = r^n [\sin(n\theta) \cos(n\phi), \sin(n\theta) \sin(n\phi), \cos(n\theta)]$$

where  $r$ ,  $\theta$ , and  $\phi$  are variables in the spherical coordinates:

$$r = \sqrt{x^2 + y^2 + z^2}, \quad \theta = \arctan \frac{\sqrt{x^2 + y^2}}{z}, \quad \phi = \arctan \frac{y}{x}$$

The Mandelbulb can then be defined as the set of a constant  $c$  in 3D space under the iteration:

$$v_{i+1} = v_i^n + c$$

For  $n > 3$ , the result will be a 3D bulb-like structure with fractal surface detail and a number of lobes depending on  $n$ .

Now, we will implement our 3D Mandelbulb in the shader program, which is based on a WebGL implementation at <https://www.shadertoy.com/view/wsByzw>. Add a new shader file called *mandelbulb.wgsl* to the *examples/ch14/* folder with the following code:

```

var<private> backgr:bool = false;
var<private> max_iter:f32;
var<private> i_time:f32;

fn rotate(p:vec3<f32>, thetaX:f32, thetaY:f32) -> vec3<f32>{
    let cy = cos(thetaY);

```

## 438 | Practical GPU Graphics with wgpu and Rust

```
let sy = sin(thetaY);
let r = vec3<f32>(p.x, sy*p.z + cy*p.y, cy*p.z - sy*p.y);
let cx = cos(thetaX);
let sx = sin(thetaX);
return -vec3<f32>(cx*r.x - sx*r.z, r.y, sx*r.x + cx*r.z);
}

fn hsv2rgb(c:vec3<f32>) ->vec3<f32>{
    let f = abs((c.x*6.0 + vec3<f32>(0.0,4.0,2.0))%6.0 - 3.0) - 1.0;
    let rgb = clamp(f, vec3<f32>(0.0,0.0,0.0), vec3<f32>(1.0,1.0,1.0));
    return vec3<f32>(c.z*mix(vec3<f32>(1.0, 1.0, 1.0), rgb, c.y));
}

fn dist_sphere(p:vec3<f32>, r:f32) -> f32{
    return f32(length(p) - r);
}

fn dist_estimate(p:vec3<f32>) ->f32{
    let bailout = 2.0;
    let d_sphere = -dist_sphere(p, 13.0);
    var v = p;
    var r = 0.0;
    var dr = 1.0;
    let power = abs(cos(i_time*0.1))*10.0 + 3.0;
    for(var n:f32 = 0.0; n <= max_iter; n=n+1.0){
        r = length(v);
        if(r > bailout) { break;}
        var theta = acos(v.z/r);
        var phi = atan2(v.y, v.x);
        dr = pow(r, power - 1.0) * power*dr + 1.0;

        let vr = pow(r, power);
        theta = theta*power;
        phi = phi*power;
        v = vr * vec3<f32>(sin(theta)*cos(phi), sin(phi)*sin(theta), cos(theta));
        v = v + p;
    }
    let d_frac = 0.5*log(r)*r/dr;
    if(d_sphere < d_frac) {
        backgr = true;
    } else {
        backgr = false;
    }
    return min(d_frac, d_sphere);
}

fn get_normal(p:vec3<f32>, d:f32) -> vec3<f32>{
    let eps = vec3<f32>(0.001, 0.0, 0.0);
    return normalize(vec3<f32>(dist_estimate(p+eps.xyy), dist_estimate(p+eps.yxy),
    dist_estimate(p+eps.yyx)) - d);
}

fn soft_shadow(ro:vec3<f32>, rd:vec3<f32>, k:f32) -> f32{
    var res = 1.0;
    var t = 0.0;
    for(var i:i32 = 0; i<64; i=i+1){
        let d = dist_estimate(ro+rd*t);
        res = min(res, k*d/t);
        if(res<0.001) {break;}
    }
}
```

```

        t = t + clamp(d, 0.01, 0.2);
    }
    return clamp(res, 0.0, 1.0);
}

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>(+1.0, -1.0),
        vec2<f32>(-1.0, +1.0),
        vec2<f32>(+1.0, +1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    time: f32;
    max_iter: f32;
    mousex: f32;
    mousey: f32;
    width: f32;
    height: f32;
    scale: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms: FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    i_time = f_uniforms.time;
    max_iter = f_uniforms.max_iter;
    let mousex = f_uniforms.mousex;
    let mousey = f_uniforms.mousey;
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;
    let scale = f_uniforms.scale;

    let z = vec2<f32>(scale*(coord_in.x - 0.5*w)/w, -scale*(h/w)*(coord_in.y - 0.5*h)/h);

    let mouse = vec2<f32>(7.0*(mousex/w - 0.5), 7.0*(mousey/h - 0.5));

    var ro = rotate(vec3<f32>(0.0,0.0,2.5), mouse.x, mouse.y);
    var rd = -rotate(vec3<f32>(z,1.0), mouse.x, mouse.y);
    let light = rotate(vec3<f32>(0.0, 0.3, 0.77), mouse.x, mouse.y);

    var light_color = vec3<f32>(0.8, 0.9, 1.0);
    var material:vec3<f32>;
    var color:vec3<f32>;
    var frag_color:vec4<f32>;
    let eps = 0.002;
    var dist:f32;

    for( var n:f32 = 0.0; n < 100.0; n=n+1.0){
        dist = dist_estimate(ro);
        if(dist < eps) {break;}
        ro = ro + rd*dist*0.5;
    }
}

```

```

if(backgr == true){
    color = vec3<f32>(0.3, 0.8, 1.0) * (0.5 - 0.4*z.x);
    frag_color = vec4<f32>(color, 1.0);
    return frag_color;
}

let norm = get_normal(ro, dist);
material = hsv2rgb(vec3<f32>(dot(ro, ro) - 0.27, 1.2, 1.0));

let light_dir = normalize(light - rd);
let shadow = soft_shadow(ro + 0.001*norm, light, 5.0);
let ambient = 0.22;
let diff = clamp(dot(light, norm), 0.0, 1.0)*shadow*0.9;
let spec = pow(clamp(dot(norm, light_dir), 0.0, 1.0), 32.0)*shadow*1.8;
color = light_color*(ambient + diff + spec)*material;
color = pow(color, light_color);
let fd = vec2<f32>((6.0*coord_in.x - w)/h, 6.0*(coord_in.y-h)/h);
color = color*(1.0-length(fd)*0.07);
return vec4<f32>(color, 1.0);
}

```

This code uses a raymarching technique to render our Mandelbulb fractal. Raymarching is an iterative ray intersection test in which you step along a ray and test for intersections. It renders with a point that closes enough to the surface of our Mandelbulb, which is much faster and allows real-time rendering for a complicated scene.

To make raymarching effective, we define a signed distance function (SDF) called *dist\_sphere* because the distance is negative or positive depending on whether we are inside or outside the sphere. Our goal is to get close enough to the surface, so we do not need to find the exact intersection. This is the key idea of raymarching.

The *dist\_estimate* function is a distance estimator that calculates distances from a given point to our Mandelbulb surface and returns a lower bound of these distances. Inside this function, we define the order of power varying with a uniform parameter, *time*:

```
let power = abs(cos(i_time*0.1))*10.0 + 3.0;
```

This means that the power of a 3D vector, which we use to describe our Mandelbulb, will vary in the range [3, 13], resulting in different 3D Mandelbulb images depending on the *time* parameter.

We set the maximum iteration to a user-specified parameter *max\_iter* for the loop within the *dist\_estimate* function. The loop contains a break when  $r > 2$  to stop the loop. This will guarantee that either  $r$  will go to infinity, or the number of maximum iterations will be reached.

We use the other helper functions to calculate rotation, color, normal, and lighting.

Now, add a new Rust file called *mandelbulb.rs* to the *examples/ch14/* folder with the following code:

```

use std::borrow::Cow;
mod fractal3d;

fn main(){
    let mut max_iter:f32 = 8.0;
    let mut scale:f32 = 2.0;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        max_iter = args[1].parse().unwrap();
    }
}

```

```

if args.len() > 2 {
    scale = args[2].parse().unwrap();
}

let source = wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("mandelbulb.wgsl")));
fractal3d::run(source, max_iter, scale, true, "mandelbulb");
}

```

This code simply set the maximum iteration, scaling factor, and the shader file as the input arguments for the `fractal3d::run` function.

Now, add the following code snippet to the `Cargo.toml` file:

```

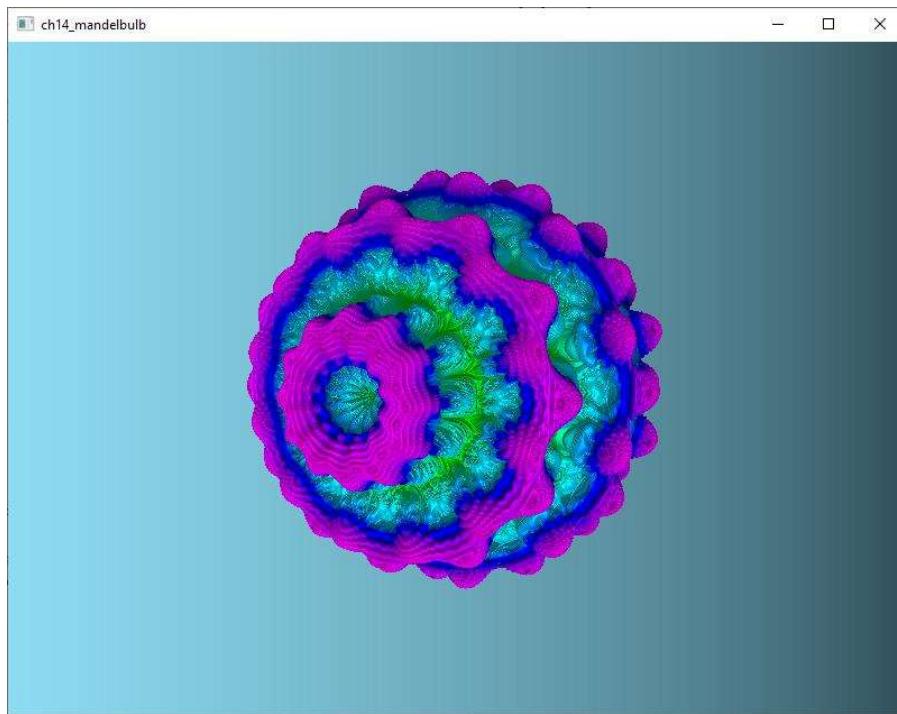
[[example]]
name = "ch14_mandelbulb"
path = "examples/ch14/mandelbulb.rs"

```

We can then run the following command to create 3D Mandelbulb:

```
cargo run --example ch14_mandelbulb
```

This produces the results shown in Fig.14-28. This example allows you to interact with the fractal image using your mouse.



*Fig.14-28. Animated 3D Mandelbulb.*

### 14.7.3 Mandelbrot in 3D Space

Here, we will create a 3D Mandelbrot set in the shader program, which is based on a WebGL implementation at <https://www.shadertoy.com/view/MtfGWM>. This example will also contain several interesting 3D Julia sets. Add a new shader file called `mandelbrot3d.wgsl` to the `examples/ch14/` folder with the following code:

```

let pi:f32 = 3.14159265359;
var<private> cv:vec3<f32>;
var<private> mcol:vec3<f32>;
var<private> b_color:bool = false;
var<private> L:vec3<f32>;
var<private> max_iter: i32;
var<private> i_time:f32;

fn de(p1: vec3<f32>) -> f32{
    var dr = 1.0;
    var r = length(p1);
    var p = p1;
    for(var i:i32 = 0; i< max_iter; i=i+1){
        if(r>2.0) {break;}
        dr = 2.0*dr*r;
        let psi = abs((atan2(p.z, p.y)+pi/8.0)%(pi/4.0) - pi/8.0);
        p.y = cos(psi)*length(p.yz);
        p.z = sin(psi)*length(p.yz);
        let p2 = p*p;
        p = vec3<f32>(vec2<f32>(p2.x - p2.y, 2.0*p.x*p.y)*(1.0-p2.z/(p2.x+p2.y+p2.z)),
                      2.0*p.z*sqrt(p2.x+p2.y)) + cv;
        r = length(p);
        if(b_color == true && i == 3){mcol = p;}
    }
    return min(log(r)*r/max(dr, 1.0), 1.0);
}

fn rnd(c:vec2<f32>) -> f32{
    return fract(sin(dot(vec2<f32>(1.317, 19.753), c))*413.7972);
}

fn rnd_start(frag_coord:vec2<f32>) -> f32{
    return 0.5+0.5*rnd(frag_coord.xy+vec2<f32>(i_time*217.0, i_time*217.0));
}

fn shadao(ro:vec3<f32>, rd:vec3<f32>, px:f32, frag_coord:vec2<f32>) ->f32{
    var res = 1.0;
    var t = 2.0*px*rnd_start(frag_coord);
    var d:f32;
    for(var i:i32 = 0; i<4; i=i+1){
        d = max(px, de(ro+rd*t)*1.5);
        t = t+d;
        res = min(res, d/t + t*0.1);
    }
    return res;
}

fn sky(rd: vec3<f32>) -> vec3<f32>{
    return vec3<f32>(0.5 + 0.5*rd.y, 0.5 + 0.5*rd.y, 0.5 + 0.5*rd.y);
}

fn color(r1:vec3<f32>, rd:vec3<f32>, t:f32, px:f32, col:vec3<f32>, b_fill:bool, frag_coord:vec2<f32>) -> vec3<f32>{
    var ro = r1+rd*t;
    b_color = true;
    var d = de(ro);
    b_color = false;
    let e = vec2<f32>(px*t, 0.0);
    let dn = vec3<f32>(de(ro-e.xyy), de(ro-e.yxy), de(ro-e.yyx));
}

```

```

let dp = vec3<f32>(de(ro+e.xyy), de(ro+e.yxy), de(ro+e.yyx));
let N = (dp-dn)/(length(dp-vec3<f32>(d,d,d))+ length(vec3<f32>(d,d,d)-dn));
let R = reflect(rd, N);
let lc = vec3<f32>(1.0, 0.9, 0.8);
let sc = sqrt(abs(sin(mcol)));
let rc = sky(R);
var sh = clamp(shadow(ro, L, px*t, frag_coord)+0.2, 0.0, 1.0);
sh = sh*(0.5+0.5*dot(N,L))*exp(-t*0.125);
let scol = sh*lc*(sc+rc*pow(max(0.0, dot(R,L)), 4.0));

if(b_fill){ d= d*0.05;}
let col1 = mix(scol, col, clamp(d/(px*t), 0.0, 1.0));
return col1;
}

fn look_at(p:vec3<f32>) -> mat3x3<f32>{
    let p1 = normalize(p);
    let rt = normalize(cross(p1, vec3<f32>(0.0, 1.0, 0.0)));
    return mat3x3<f32>(rt, cross(rt,p1), p1);
}

fn julia(t:f32) -> vec3<f32>{
    let t1 = t%7.0;
    if(t1<1.0) { return vec3<f32>(-0.8, 0.0, 0.0);}
    if(t1<2.0) { return vec3<f32>(0.28, -0.5, 0.0);}
    if(t1<3.0) { return vec3<f32>(-0.8, 1.0, -0.69);}
    if(t1<4.0) { return vec3<f32>( 0.5, -0.84, -0.13);}
    if(t1<5.0) { return vec3<f32>( 0.5, -0.34, 0.13);}
    if(t1<6.0) { return vec3<f32>(-0.16, 0.657, 0.0);}

    return vec3<f32>(0.0, 1.0, -1.0);
}

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>( 1.0, -1.0),
        vec2<f32>(-1.0, 1.0),
        vec2<f32>( 1.0, 1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    time: f32;
    max_iter: f32;
    mousex: f32;
    mousey: f32;
    width: f32;
    height: f32;
    scale: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms: FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {

```

## 444 | Practical GPU Graphics with wgpu and Rust

```
i_time = f_uniforms.time;
max_iter = i32(f_uniforms.max_iter);
let w:f32 = f_uniforms.width;
let h:f32 = f_uniforms.height;
let scale = f_uniforms.scale;

let px = 0.5/h;
L = normalize(vec3<f32>(0.4, 0.8, -0.6));
var tim = 0.5*i_time;

let ro = vec3<f32>(cos(1.3*tim), sin(0.4*tim), sin(tim))*3.0;
let rd = look_at(vec3<f32>(-0.1, -0.1, -0.1)-ro)*normalize(vec3<f32>(scale*(2.0*coord_in.x-w)/h,
-scale*(2.0*coord_in.y-h)/h, 3.0));

tim = tim*0.6;
if(tim%15.0 < 7.0){
    cv = mix(julia(tim - 1.0), julia(tim), smoothStep(0.0, 1.0, fract(tim)*7.0));
} else {
    cv = vec3<f32>(-cos(tim), cos(tim)*abs(sin(tim*0.3)), -0.5*abs(sin(tim)));
}

var t = de(ro)*rnd_start(coord_in.xy);
var d = 0.0;
var od = 10.0;
var edge = vec3<f32>(-1.0, -1.0, -1.0);
var b_grab = false;
var col = sky(rd);

for(var i:i32 = 0; i<78; i=i+1){
    t = t + 0.5*d;
    d = de(ro+rd*t);
    if(d>od){
        if(b_grab && od < px*t && edge.x < 0.0){
            edge = vec3<f32>(edge.yz, t-od);
            b_grab = false;
        }
    } else {
        b_grab = true;
    }
    od = d;
    if(t > 10.0 || d < 0.00001) { break; }
}

var b_fill = false;
d = d*0.05;
if(d < px*t && t < 10.0){
    if (edge.x > 0.0) {edge = edge.zxy;}
    edge = vec3<f32>(edge.yz, t);
    b_fill = true;
}
for(var i:i32 = 0; i < 3; i = i+1){
    if(edge.z > 0.0) {
        col = color(ro, rd, edge.z, px, col, b_fill, coord_in.xy);
        edge = edge.zxy;
        b_fill = false;
    }
}
return vec4<f32>(col, 1.0);
```

Here, we also use the raymarching technique to render our 3D fractal. The *de* function is a distance estimator, in which we use a *for*-loop to iterate the function  $v_{i+1} = v_i^2 + c$  by setting the maximum iteration to *max\_iter* specified by the user. The constant *cv* is constructed using the *julia* function that contains seven Julia variants.

Now, add a new Rust file called *mandelbrot3d.rs* to the *examples/ch14/* folder with the following code:

```
use std::borrow::Cow;
mod fractal3d;

fn main(){
    let mut max_iter:f32 = 3.0;
    let mut scale:f32 = 1.0;
    let args: Vec<String> = std::env::args().collect();
    if args.len() > 1 {
        max_iter = args[1].parse().unwrap();
    }
    if args.len() > 2 {
        scale = args[2].parse().unwrap();
    }

    let source = wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("mandelbrot3d.wgsl")));
    fractal3d::run(source, max_iter, scale, false, "mandelbrot3d");
}
```

This code simply set the maximum iteration, scaling factor, and the shader file as the input arguments for the *fractal3d::run* function.

Now, add the following code snippet to the *Cargo.toml* file:

```
[[example]]
name = "ch14_mandelbrot3d"
path = "examples/ch14/mandelbrot3d.rs"
```

We can then run the following commands to create 3D Mandelbulb:

```
cargo run --example ch14_mandelbrot3d "3"
cargo run --example ch14_mandelbrot3d "8"
```

This produces the results shown in Fig.14-29.

Note that in this example, the fractal images will continuously change according to the constant *cv* specified by different Julia variants. With the increase of the *max\_iter* parameter, the images will gradually converge to the final fractals.

## 14.7.4 Mandelbox Sweeper

The 3D Mandelbox is a box-like fractal discovered by Tom Lowe in 2010. It is defined in a similar way to the 2D Mandelbrot set as a map of continuous Julia sets. We calculate the Mandelbox using a function repeatedly to every point in 3D space. A 3D point will be part of a Mandelbox if it does not escape to infinity.

Here, we will create a 3D Mandelbox sweeper in the shader program, which is based on a WebGL implementation at <https://www.shadertoy.com/view/3lyXDm>. Add a new shader file called *mandelbox.wgsl* to the *examples/ch14/* folder with the following code:

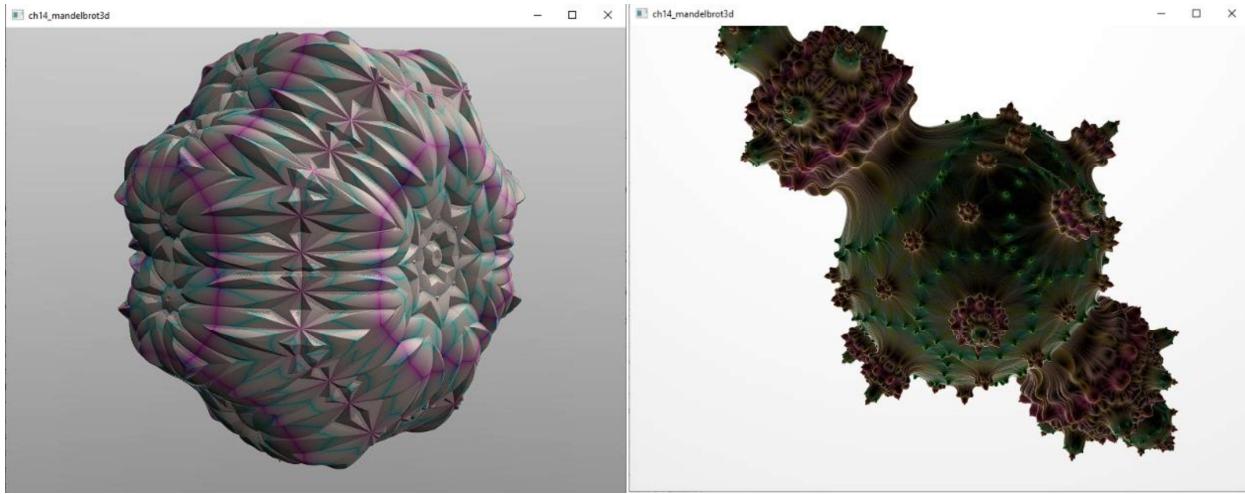


Fig.14-29. 3D Mandelbrot Sets with different maximum iteration = 3 (left) and 8 (right).

```
// define global variables
let e = vec2<f32>(0.000035, -0.000035);
var<private> z:vec2<f32>;
var<private> v:vec2<f32>;
var<private> t:f32;
var<private> tt:f32;
var<private> b:f32;
var<private> g:f32;
var<private> g2:f32;
var<private> bb:f32;
var<private> np:vec3<f32>;
var<private> bp:vec3<f32>;
var<private> pp:vec3<f32>;
var<private> po:vec3<f32>;
var<private> no:vec3<f32>;
var<private> a1:vec3<f32>;
var<private> ld:vec3<f32>;

// define helper functions
fn bo(p:vec3<f32>, r:vec3<f32>) -> f32{ // box primitive function
    let p1 = abs(p) - r;
    return max(max(p1.x, p1.y), p1.z);
}

fn r2(r:f32) -> mat2x2<f32>{           // rotation function
    return mat2x2<f32>(vec2<f32>(cos(r), sin(r)), vec2<f32>(-sin(r), cos(r)));
}

fn fb(p:vec3<f32>, m:f32) -> vec2<f32> {
    var p1 = p;
    p1.y = p1.y + 0.05*bb;
    var h = vec2<f32>(bo(p1, vec3<f32>(5.0, 1.0, 3.0)), 3.0);
    var t = vec2<f32>(bo(p1, vec3<f32>(5.0, 1.0, 3.0)), 3.0);
    t.x = max(t.x,-(length(p1) - 2.5));
    t.x = max(abs(t.x) - 0.2, p1.y - 0.4);
    h = vec2<f32>(bo(p1, vec3<f32>(5.0,1.0,3.0)),6.0);
    h.x = max(h.x,-(length(p1) - 2.5));
    h.x = max(abs(h.x) - 0.1, p1.y - 0.5);
    if(t.x >= h.x) {t = h;}
}
```

```

h = vec2<f32>(bo(p1+vec3<f32>(0.0,0.4,0.0),vec3<f32>(5.4,0.4,3.4)),m);
h.x = max(h.x,-(length(p1) - 2.5));
if(t.x >= h.x) {t = h;}
h = vec2<f32>(length(p1) - 2.0, m);
if(t.x >= h.x) {t = h;}
t.x = 0.7*t.x;
return t;
}

fn mp(p1:vec3<f32>) -> vec2<f32>{
    pp = p1;
    bp = p1;
    var p= p1;
    var pyz = p.yz*r2(sin(pp.x*0.3-tt*0.5)*0.4);
    p.y = pyz.x;
    p.z = pyz.y;
    bp.y = p.y;
    bp.z = p.z;
    pyz = pyz*r2(1.57);
    p.y = pyz.x;
    p.z = pyz.y;
    b = sin(pp.x*0.2 + tt);
    bb = cos(pp.x*0.2 + tt);
    p.x = (p.x-tt*2.0)%10.0 - 5.0;
    var np = vec4<f32>(p*0.4, 0.4);
    for(var i:i32=0; i<4; i=i+1){
        np.x = abs(np.x) - 1.0;
        np.y = abs(np.y) - 1.2;
        np.z = abs(np.z);
        np.x = 2.0*clamp(np.x, 0.0, 2.0) - np.x;
        np.y = 2.0*clamp(np.y, 0.0, 0.0) - np.y;
        np.z = 2.0*clamp(np.z, 0.0, 4.3+bb) - np.z;
        np = np*(1.3)/clamp(dot(np.xyz,np.xyz), 0.1, 0.92);
    }
    var h = fb(abs(np.xyz)-vec3<f32>(2.0,0.0,0.0),5.0);
    var t = fb(abs(np.xyz)-vec3<f32>(2.0,0.0,0.0),5.0);
    t.x = t.x/np.w;
    t.x = max(t.x,bo(p, vec3<f32>(5.0,5.0,10.0)));
    np = 0.5*np;
    let np_yz = np.yz * r2(0.785);
    np.y = np_yz.x;
    np.z = np_yz.y;
    np.y = np.y + 2.5;
    np.z = np.z + 2.5;
    h = fb(abs(np.xyz)-vec3<f32>(0.0,4.5,0.0),7.0);
    h.x = max(h.x,-bo(p,vec3<f32>(20.0,5.0,5.0)));
    h.x = h.x/(np.w*1.5);
    if(t.x >= h.x) {t = h;}
    h = vec2<f32>(bo(np.xyz,vec3<f32>(0.0,b*20.0,0.0)),6.0);
    h.x = h.x/(np.w*1.5);
    g2 = g2 + 0.1/(0.1*h.x*h.x*(1000.0-b*998.0));
    if(t.x >= h.x) {t = h;}
    h = vec2<f32>(0.6*bp.y+sin(p.y*5.0)*0.03,6.0);
    if(t.x >= h.x) {t = h;}
    h = vec2<f32>(length(cos(bp.xyz*0.6+vec3<f32>(tt,tt,0.0)))+0.003,6.0);
    g = g + 0.1/(0.1*h.x*h.x*4000.0);
    if(t.x >= h.x) {t = h;}
    return t;
}

```

## 448 | Practical GPU Graphics with wgpu and Rust

```
fn tr(ro:vec3<f32>, rd:vec3<f32>) -> vec2<f32>{
    var h = vec2<f32>(0.1, 0.1);
    var t = vec2<f32>(0.1, 0.1);
    for(var i:i32=0; i<100; i=i+1){
        h = mp(ro+rd*t.x);
        //if(h.x < 0.0001 || t.x > 40.0) { break; }
        if(h.x < 0.0001*(1.0+length(t)) || t.x > 40.0) { break; }
        t.x = t.x + h.x;
        t.y = h.y;
    }
    if(t.x > 40.0) {t.y = 0.0;}
    return t;
}

fn a(d:f32) -> f32{
    return clamp(mp(po+no*d).x/d, 0.0, 1.0);
}

fn s(d:f32) -> f32{
    return smoothStep(0.0,1.0,mp(po+ld*d).x/d);
}

// vertex shader
[[stage(vertex)]]
fn vs_main([[builtin(vertex_index)]] in_vertex_index: u32) -> [[builtin(position)]] vec4<f32> {
    var pos = array<vec2<f32>,4>(
        vec2<f32>(-1.0, -1.0),
        vec2<f32>(-1.0, 1.0),
        vec2<f32>(1.0, -1.0),
        vec2<f32>(1.0, 1.0),
    );
    return vec4<f32>(pos[in_vertex_index], 0.0, 1.0);
}

// fragment shader
[[block]] struct FragUniforms {
    time: f32;
    max_iter: f32;
    mousex: f32;
    mousey: f32;
    width: f32;
    height: f32;
    scale: f32;
};
[[binding(0), group(0)]] var<uniform> f_uniforms: FragUniforms;

[[stage(fragment)]]
fn fs_main([[builtin(position)]] coord_in : vec4<f32>) -> [[location(0)]] vec4<f32> {
    let i_time = f_uniforms.time;
    let w:f32 = f_uniforms.width;
    let h:f32 = f_uniforms.height;

    let uv = vec2<f32>((coord_in.x/w - 0.5)*w/h, -(coord_in.y/h - 0.5));
    tt = i_time%62.8318;
    var ro = mix(vec3<f32>(1.0,1.0,1.0),vec3<f32>(-0.5,1.0,-1.0),ceil(sin(tt*0.5)))*
        vec3<f32>(10.0,2.8+0.75*smoothStep(-1.5,1.5,1.5*cos(tt+0.2)),cos(tt*0.3)*3.1);
    var cw = normalize(vec3<f32>(0.0,0.0,0.0)-ro);
    var cu = normalize(cross(cw,normalize(vec3<f32>(0.0,1.0,0.0))));
```

```

var cv = normalize(cross(cu,cw));
var rd = mat3x3<f32>(cu,cv,cw)*normalize(vec3<f32>(uv, 0.5));
ld = normalize(vec3<f32>(0.2,0.4,-0.3));
var co = vec3<f32>(0.1,0.2,0.3)-length(uv)*0.1 - rd.y*0.2;
var fo = vec3<f32>(0.1,0.2,0.3)-length(uv)*0.1 - rd.y*0.2;
z = tr(ro, rd);
t = z.x;
if(z.y > 0.0){
    po = ro + rd*t;
    no = normalize(e.xyy*mp(po+e.xyy).x+e.yyx*
        mp(po+e.yyx).x+e.yxy*mp(po+e.yxy).x+e.xxx*mp(po+e.xxx).x);
    al = mix(vec3<f32>(0.1,0.2,0.4), vec3<f32>(0.1,0.4,0.7), 0.5+0.5*sin(bp.y*7.0));
    if(z.y < 5.0) { al=vec3<f32>(0.0,0.0,0.0);}
    if(z.y > 5.0) { al=vec3<f32>(1.0,1.0,1.0);}
    if(z.y > 6.0) { al=mix(vec3<f32>(1.0,0.5,0.0),vec3<f32>(0.9,0.3,0.1), 0.5+0.5*sin(bp.y*7.0));}
    var dif = max(0.0, dot(no, ld));
    var fr = pow(1.0 + dot(no, rd), 4.0);
    var sp = pow(max(dot(reflect(-ld,no),-rd),0.0),40.0);
    co = mix(sp+mix(vec3<f32>(0.8,0.8,0.8),vec3<f32>(1.0,1.0,1.0),
        abs(rd))*al*(a(0.1)*a(0.2)+0.2)*(dif+s(2.0)),fo,min(fr,0.2));
    co = mix(fo,co,exp(-0.0003*t*t*t));
}
return vec4<f32>(pow(co+g*0.2+g2*mix(vec3<f32>(1.0,0.5,0.0),vec3<f32>(0.9,0.3,0.1),
    0.5+0.5*sin(bp.y*3.0)),vec3<f32>(0.65, 0.65, 0.65)),1.0);
}

```

Here, we also use the raymarching technique and distance estimator to render our 3D fractal.

Now, add a new Rust file called *mandelbox.rs* to the *examples/ch14/* folder with the following code:

```

use std::borrow::Cow;
mod fractal3d;

fn main(){
    let source = wgpu::ShaderSource::Wgsl(Cow::Borrowed(include_str!("mandelbox.wgsl")));
    fractal3d::run(source, 10.0, 1.0, false, "mandelbox");
}

```

This code simply set the shader file as the input arguments for our *fractal3d::run* function.

Now, add the following code snippet to the *Cargo.toml* file:

```

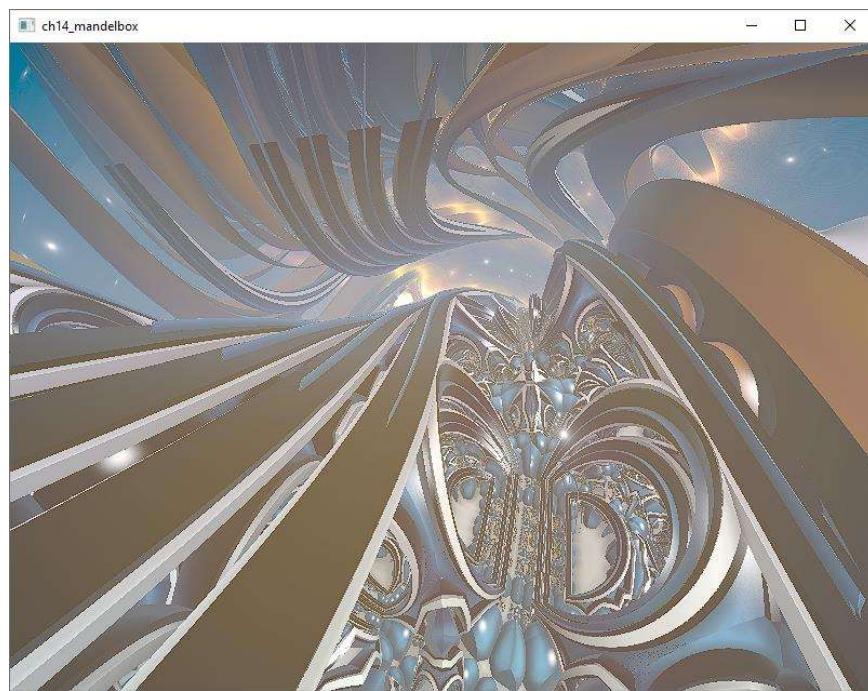
[[example]]
name = "ch14_mandelbox"
path = "examples/ch14/mandelbox.rs"

```

We can then run the following commands to create 3D Mandelbox sweeper:

```
cargo run --example ch14_mandelbox
```

This produces the results shown in Fig.14-30.



*Fig. 14-30. 3D Mandelbox sweeper.*

# Index

- 3D charts with multiple pipelines, 315
  - Rust code, 318
  - shader program, 317
  - wireframe, 316
- 3D charts with multiple render passes, 334
  - Rust code, 334
- 3D fractals, 432
  - common code, 432
- 3D lines, 89
  - common code, 89
  - Rust code, 92
  - shader program, 98
- 3D Mandelbrot, 441
- 3D shapes with texture, 232
  - cube with texture, 232
  - cylinder with texture, 237
  - sphere with texture, 235
- 3D surface charts, 257
- 3D transformation matrix, 67
- 3D transformations
  - rotation, 72
  - scaling, 69
  - translation, 71
- 3D vector and matrix, 68
- 3D wireframe, 131
  - common code, 131
- ambient light, 153
- arrays, 21
- ASP.NET Core, 4
- bind group, 87
- Blinn-Phong, 157
- camera code, 121
- camera control, 121
- Rust code, 123
- camera space, 76
- Cargo.toml* configuration, 12
- cgmath* crate, 67
- charts with coordinate axes, 329
  - Rust code, 330
  - shader code, 329
  - TypeScript code, 334
- clip coordinates, 76
- color conversion in shader, 387
- color interpolation, 187
- color model, 185
- colored triangles
  - Rust code, 53
  - shader code, 60
- colored triangles, 52
- colormap data, 186
- colormaps in shader
- combining transformations, 74
- complex functions, 383
- complex functions in shader
  - addition, 384
  - argument, 384
  - color functions, 387
  - complex conjugate, 384
  - cosine, 386
  - division, 384
  - exponential, 385
  - inverse, 385
  - inverse hyperbolic sine function, 386
  - logarithm, 385
  - math operations, 384
  - multiplication, 384

power function, 385  
 sine, 385  
 square root, 385  
 tangent, 386  
 complex numbers, 383  
 compute boids, 347  
     Rust code, 347  
     shader code, 354  
 compute shader, 339, 340  
     invocations, 340  
 cone wireframe, 146  
     Rust code, 147  
 cone with lighting, 178  
     normals, 179  
     Rust code, 180  
 control flows  
     if and if let, 18  
 control flows  
     for loop, 19  
     loop, 20  
     match, 19  
     while loop, 20  
 creating a window, 15  
 cube wireframe, 136  
     TypeScript code, 136  
 cube with distinct face color  
     Rust Code, 102  
 cube with distinct face colors, 100  
     shader program, 109  
     vertex data, 100  
 cube with distinct vertex color  
     Rust Code, 111  
 cube with distinct vertex colors, 111  
     vertex data, 111  
 cube with lighting, 170  
     Rust code, 170  
 cube with multiple textures  
     Rust code, 254  
 cylinder wireframe, 141  
     Rust code, 143  
 cylinder with lighting, 174  
     normals, 175  
     Rust code, 177  
 cylindrical coordinate system, 142  
 device manager, 7  
 different objects, 301  
     Rust code, 301  
 diffuse light intensity, 156  
 diffuse lighting, 154  
 DirectX11, 1  
 DirectX12, 1  
 domain coloring for complex functions, 392  
     Rust code, 392  
     shader code, 397  
 domain coloring for iterated functions, 407  
     shader code, 407  
 drawing surface, 31  
 eye coordinates, 76  
 first *wgpu* example, 25  
 fractal, 417  
     Julia set, 424  
 fragment shader, 38  
 global workgroup, 340  
 GLSL shader, 38  
 GPU buffers, 51  
 GPU command encoder, 342  
 GPU queue commands, 342  
*GPUAddressMode*, 220  
*GPUSampler*, 219  
 half-angle vector, 157  
 hardware requirements, 7  
 image textures, 218  
 Julia set  
     Rust code, 424  
     shader code, 429  
 light components, 153  
 lighting, 153  
     common code, 161  
 lighting calculation, 156  
 lighting in shaders, 158  
     transform normals, 158  
     WGLL shader, 159  
 lighting model, 153  
 line primitives, 45  
*line-list*, 45  
*line-strip*, 45  
 local workgroup, 340  
 magnification filter, 219  
 Mandelbox sweeper, 445  
 Mandelbrot set, 417  
     formula, 417  
     Rust code, 418  
     shader code, 422  
 Mandelbulb, 437  
 Metal, 1  
 minification filter, 219  
 mipmap, 219  
 modeling transformations, 76

- modelview, 76
- multiple cubes
  - Rust code, 293
  - shader code, 300
- multiple cubes with instancing, 292
- multiple objects, 285
- multiple textures, 252
- normal vectors, 154
  - cube, 155
  - cylinder, 155
  - polyhedral surface, 155
  - quad, 156
  - sphere, 155
- normalized device coordinates, 37, 76
- object coordinates, 76, 217
- objects with multiple pipelines, 304
  - Rust code, 304
- OpenGL, 1
- OPENGL\_TO\_WGPU\_MATRIX*, 91
- OpenGLES, 1
- orthographic projection, 83
- parametric 3D surface charts, 275
  - Klein bottle, 280
  - sphere surface, 277
  - torus surface, 278
  - Wellenkugel surface, 282
- parametric coordinates, 217
- parametric surface, 205
  - breather surface, 213
  - Klein bottle, 206
  - normals, 205
  - seashell surface, 210
  - Sievert-Enneper surface, 212
  - Wellenkugel surface, 209
- parametric surfaces with texture, 246
  - Klein bottle, 248
  - Wellenkugel surface, 250
- particle collision, 370
  - Rust code, 370
  - shader code, 378
- particle systems, 339
- particles under gravity, 358
  - Rust code, 358
  - shader code, 366
- perspective projection, 79
- Phong model, 157
- point primitives, 42
  - Rust code, 42
- shader code, 43
- projection transform, 76
- rendering output, 34
- rendering pipeline, 32
- rotate objects, 119
  - Rust code, 119
- rotation in GPU, 343
  - Rust code, 343
  - shader code, 346
- Rust
  - control clows, 18
  - data types, 17
  - editions, 13
  - enum, 23
  - examples configuration, 14
  - function, 17
  - generics, 24
  - run application, 11
  - Rust installer, 10
  - struct, 22
- Rust basics, 15
- Rust code, 26
- Rust crates
  - anyhow*, 13
  - cgmath*, 12
  - env\_logger*, 12
  - futures*, 12
  - gfx-hal*, 12
  - image*, 12
  - log*, 12
  - pollster*, 13
  - wgpu*, 12
  - winit*, 13
- Rust installation, 10
- Rust main function, 28
- Rust tools
  - cargo*, 10
  - rustc*, 10
  - rustup*, 10
- screen coordinates, 217
- shader code, 36
- shader module, 32
- shader program, 35
- shaders for 3D charts, 259
  - WGSL shader, 259
- shaders with lighting and vertex color, 188
- shaders with texture, 222
- simple 3D surface charts, 269

peaks chart, 274  
 sinc chart, 271  
 simple surface  
     normals, 191  
     peaks surface, 203  
     sinc surface, 201  
 simple surfaces, 190  
 simple surfaces with texture, 241  
     peaks surface, 245  
     sinc surface, 243  
 slices, 21  
 specular light, 154  
 specular light intensity, 157  
 sphere wireframe, 138  
     Rust code, 140  
 sphere with lighting, 171  
     normals, 172  
     Rust code, 173  
 spherical coordinate system, 138  
 SPIR-V, 36  
 SPIR-V binaries, 36  
 square textures, 258  
 squares, 61  
     Rust code, 62  
 squares with index, 63  
     Rust code, 63  
 surface charts, 257  
     common code, 261  
 texture, 217  
 texture coordinates, 217, 252, 259  
 TOML format, 12  
 torus wireframe, 149  
     Rust code, 150  
 torus with lighting, 181  
     normals, 182  
     Rust code, 183  
 transform coordinates, 75  
 triangle example, 38  
     Rust code, 39  
     shader code, 39  
 Triangle Primitives, 47  
     Rust code, 47  
     shader code, 48  
*triangle-list*, 47  
*triangle-strip*, 47  
 two cubes, 285  
     Rust code, 285  
 UBO, 87  
 uniform buffer, 87  
 uniform buffer object, 87  
 vectors, 21  
 vertex shader, 38  
*VertexBufferLayout*, 58  
 view frustum, 79  
 viewing and projection, 75  
 viewing transform, 76  
 viewing volume, 79  
 viewport transform, 76  
 Visual Studio Code, 11  
 Vulkan, 1  
 WASM, 16  
 WebAssembly, 16  
 WebGPU Shading Language, 36  
*wgpu*  
     API, 30  
*player*, 7  
     texture mapping, 218  
     transformations, 86  
*wgpu-core*, 7  
*wgpu-hal*, 7  
*wgpu-types*, 7  
*wgpu* API, 1  
*wgpu* backends, 30  
     *BROWSER\_WEBGPU*, 30  
     *DX11*, 30  
     *DX12*, 30  
     *GL*, 30  
     *METAL*, 30  
     *PRIMARY*, 30  
     *SECONDARY*, 30  
     *VULKAN*, 30  
*wgpu* primitives, 41  
 WGSL, 2  
 WGSL extension, 13  
 WGSL shaders, 25, 36  
 wireframe as texture, 258  
 workgroup, 340  
 world coordinates, 76  
 write and read buffer, 341



