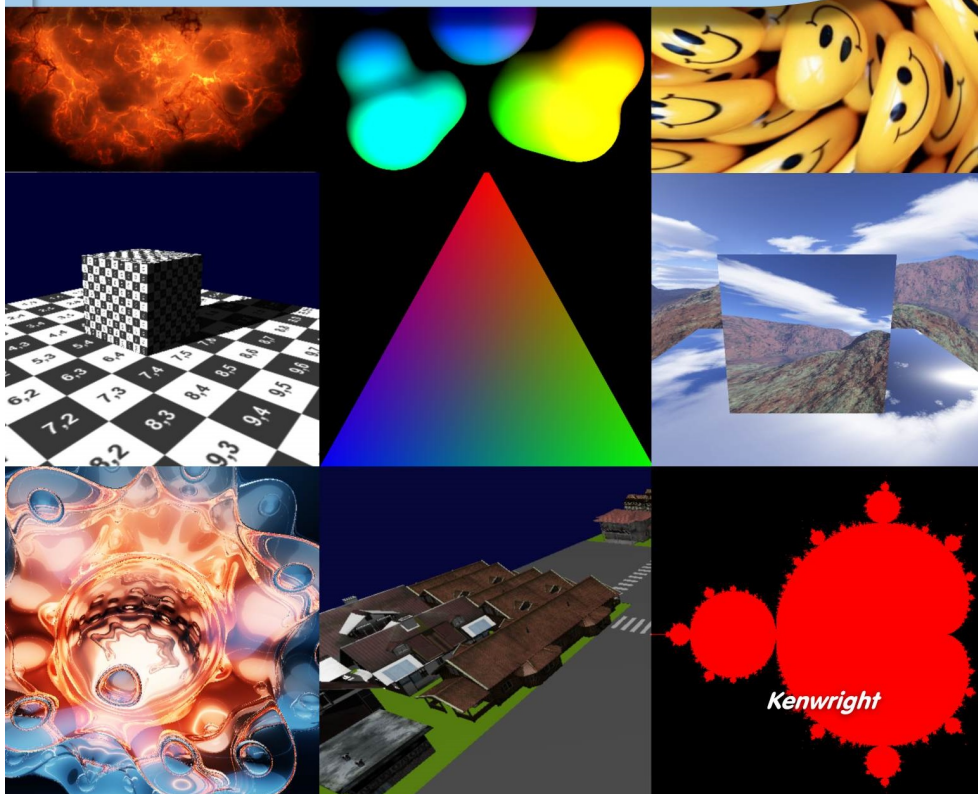


101 VULKAN Projects

A Hands-On Journey Through 101
Vulkan Programming Examples



Copyright © 2024 Kenwright
All rights reserved.

No part of this book may be used or reproduced in any manner whatsoever without written permission of the author except in the case of brief quotations embodied in critical articles and reviews.

BOOK TITLE:

101 Vulkan Projects

A Hands-On Journey Through 101 Vulkan Programming Examples

(First Edition)

ISBN-13: 9x7-8-34-4x18x8-0

The author accepts no responsibility for the accuracy, completeness or quality of the information provided, nor for ensuring that it is up to date. Liability claims against the author relating to material or non-material damages arising from the information provided being used or not being used or from the use of inaccurate and incomplete information are excluded if there was no intentional or gross negligence on the part of the author. The author expressly retains the right to change, add to or delete parts of the book or the whole book without prior notice or to withdraw the information temporarily or permanently.

Edition: 12019121240

Preface

The Vulkan API stands as one of the most versatile and high-performance tools in modern graphics programming, offering developers direct access to the full power of today's GPUs. Its flexibility and performance, however, come with a steep learning curve. Unlike more traditional graphics APIs, Vulkan demands an active understanding of the hardware and precise control over every aspect of rendering and computation. For newcomers and even experienced developers, mastering Vulkan can be both challenging and rewarding.

This book, **101 Project Examples with Vulkan API** takes an active approach to learning, designed to keep the process both engaging and practical. Instead of merely reading about Vulkan's complex systems, you'll build your understanding project by project, starting from the basics and steadily working your way up to advanced techniques. Each project is not only an exercise but also a potential reference point, allowing you to revisit concepts and implementations as your experience with Vulkan grows.

Whether you're just starting out or seeking to stretch your understanding, this book is crafted to meet a variety of needs:

- **A More Active Way to Learn:** Hands-on practice is at the heart of every example, ensuring you gain real-world experience with Vulkan's capabilities.
- **A Structured Pathway:** The projects are carefully ordered to increase in difficulty, ensuring you build on your skills with each chapter.
- **Flexible Utility:** Use it as a learning guide to grasp new concepts or as a tool for testing your knowledge with challenging exercises.
- **Comprehensive Coverage:** From drawing your first triangle to advanced features like ray tracing and compute shaders, the projects span Vulkan's full range.
- **Theory Meets Practice:** Each project balances explanations of underlying principles with the practical know-how to implement them.
- **Keeping it Fun:** Focusing on creative and varied projects, the book transforms Vulkan's complexity into an engaging learning experience.

You'll find that the projects in this book aren't just exercises—they're carefully crafted to deepen your understanding. Each chapter introduces a concept in the context of solving a specific challenge, from rendering a textured triangle to simulating physics with compute shaders or implementing cutting-edge ray tracing techniques.

At the end of this book, you'll have built a foundation of skills that allows you to confidently explore Vulkan's potential. You'll have the tools to tackle practical problems in graphics and compute programming, the experience to debug and optimize your code, and the inspiration to create your own groundbreaking projects.

101 Project Examples with Vulkan API is more than a guide—it's a resource to grow with you as a developer. Whether you're a beginner, an educator looking for structured exercises, or an experienced programmer seeking fresh challenges, this book will serve as both a roadmap and a source of inspiration for your Vulkan journey.



Brief Contents

I	Basic Vulkan Setup	4
1	Set Up Vulkan Instance: Create and manage a Vulkan instance	6
2	Enumerate Physical Devices: List available GPUs on your system	12
3	Enable Validation Layers: Debug Vulkan code with validation layers	17
4	Create Logical Device and Queues: Create logical devices and retrieve queue handles	21
5	Present a Blank Swapchain Image: Present an empty window via Vulkan	25
6	Set Up a Vulkan Pipeline: Understand and implement the Vulkan graphics pipeline	29
7	Clear the Screen with a Color: Implement screen-clearing using Vulkan commands	33
8	Draw a Single Triangle (Solid Color): Render a triangle with a single color	37
9	Render a Quad: Create and render a basic quad using two triangles	41
10	Add a Depth Buffer: Integrate depth buffering for 3D scenes	44
II	Intermediate Vulkan Rendering	49
11	Use Descriptor Sets: Implement descriptor sets for texture binding	51
12	Draw a Textured Triangle: Add a basic texture to your triangle	55
13	Animate Triangle Movement: Add dynamic transformation (e.g., rotate, scale)	59
14	Add Push Constants: Use Push Constants for Efficient Data Passing to Shaders	62
15	Create a Uniform Buffer: Pass data (e.g., transformations) to the shader	67
16	Use Descriptor Sets for Multiple Objects: Manage Data for Multiple Objects	71
17	Render Multiple Objects: Draw multiple triangles or quads in a single pass	75
18	Add Staging Buffers: Optimize data transfer between CPU and GPU	79
19	Generate a Procedural Gradient: Render a triangle with a procedurally generated gradient	83
20	Add Anti-Aliasing: Use multisample anti-aliasing (MSAA)	87

III	Advanced Rendering Techniques	91
21	Render to a Texture: Create an off-screen framebuffer	93
22	Implement a Frame Graph: Manage frame dependencies	97
23	Deferred Rendering: Render multiple passes for better lighting quality	101
24	Add Multiple Light Sources: Implement dynamic lighting in the scene	105
25	Physically Based Rendering (PBR): Start with simple PBR shaders	109
26	Implement Shadow Mapping: Add basic shadow rendering	113
27	Omnidirectional Shadows: Implement cube map shadows for point lights	117
28	Create a Skybox: Render a cubemap as the background	121
29	Add Post-Processing Effects: Add effects like bloom or vignette	124
30	Create a Particle System: Simulate particles using a vertex shader	129
IV	Compute Shader Projects	133
31	Hello, Compute Shader: Write your first compute shader	135
32	Compute a Mandelbrot Set: Render fractals using compute shaders	139
33	Add Image Blur Effect: Apply a Gaussian blur using compute shaders	143
34	Simulate Physics: Implement a simple physics engine with compute shaders	147
35	Game of Life Simulation: Use compute shaders for cellular automata	151
36	Generate Perlin Noise: Use compute shaders to create noise textures	155
37	Sort Data with Compute Shaders: Implement a parallel sorting algorithm	159
38	Create a GPU Particle System: Simulate particles entirely on the GPU	163
39	Simulate Fluid Dynamics: Implement 2D fluid simulation	167
40	Generate a Procedural Heightmap: Create terrains with compute shaders	171
V	Optimization Projects	175
41	Use Pipeline Caching: Reduce pipeline creation overhead	177
42	Add Command Buffer Reuse: Reuse recorded command buffers for efficiency	181
43	Implement Instanced Rendering: Draw multiple objects with fewer draw calls	185
44	Optimize Descriptor Updates: Batch descriptor set updates	188
45	Use Dynamic Buffers: Optimize buffer memory usage	193
46	Asynchronous Compute and Graphics: Use separate queues for compute and graphics	197
47	Measure Performance with Queries: Add timestamp queries to measure GPU time	201
48	Use Vulkan Subpasses: Optimize render passes with subpasses	204
49	Implement a Frame Limiter: Limit the FPS for consistency	209
50	Use Device Extensions: Enable advanced GPU features via extensions	212
VI	Advanced Vulkan Features	217
51	Ray Tracing Basics: Render a simple scene with ray tracing	219
52	Add BVH Acceleration Structures: Create bounding volume hierarchies	223
53	Render Reflections: Add ray-traced reflections	227
54	Render Refractions: Implement glass-like materials	230
55	Path Tracing Basics: Render a simple path-traced scene	235
56	Simulate Global Illumination: Add realistic lighting effects	239
57	Implement Vulkan Video Decode: Use Vulkan for video decoding	243
58	Render HDR Content: Render high-dynamic-range content	247

59	Add Vulkan Mesh Shaders: Experiment with next-gen mesh shaders	251
60	Add Tiled Rendering: Use tile-based techniques for mobile GPUs	255

VII Cross-Platform Development 259

61	Add GLFW for Windowing: Use GLFW for cross-platform Vulkan windows	261
62	Add SDL2 for Windowing: Integrate SDL2 with Vulkan	265
63	Cross-Compile for Android: Build a Vulkan app for Android	269
64	Test on macOS (MoltenVK): Run Vulkan apps on macOS using MoltenVK	273
65	Cross-Compile for Raspberry Pi: Use Vulkan on a Raspberry Pi	277
66	Test on Linux: Set up a Vulkan development environment for Linux	281
67	Use Vulkan in Unity: Integrate Vulkan into Unity projects	285
68	Integrate Vulkan with Unreal Engine: Use Vulkan in Unreal Engine	288

VIII Fun and Interactive Projects 293

69	Create a Vulkan Game Engine: Start with a simple engine core	295
70	Build a Vulkan Game (e.g., Pong): Create a basic 2D game	300
71	Render a 3D Model: Load and render 3D models (e.g., OBJ or GLTF)	305
72	Add Skeletal Animation: Render animated 3D characters	309
73	Simulate a Water Surface: Use compute shaders for water ripples	313
74	Render a Procedural Planet: Generate planets using Vulkan	317
75	Add an Interactive GUI: Use a library like Dear ImGui	321
76	Simulate Fireworks: Create a dynamic fireworks display	325
77	Render a Terrain from Heightmaps: Load heightmap data for terrain rendering	329
78	Add VR Support: Use Vulkan for VR rendering	333
79	Build an Interactive Solar System: Simulate planetary orbits	337
80	Create a Vulkan Screensaver: Use Vulkan for a graphical screensaver	340

IX Tools and Integration Projects 344

81	Integrate with OpenCV: Process Vulkan-rendered images with OpenCV	346
82	Export Frames to Images: Save Vulkan-rendered frames as PNG files	351
83	Integrate Vulkan with TensorFlow: Use Vulkan for ML model inference	354
84	Build a Vulkan Render Doc Clone: Debug Vulkan calls visually	359
85	Add Vulkan Support to Blender: Create a plugin for Vulkan rendering	363
86	Integrate Vulkan with Qt: Build Vulkan-based GUI applications with Qt	367
87	Create Vulkan Debugging Tools: Write tools for visualizing Vulkan resources	372

X Miscellaneous Projects 377

88	Render a Wireframe Model: Add wireframe rendering	379
89	Add Tessellation Shaders: Implement tessellation for smoother models	382
90	Render Text with Vulkan: Use a font library for text rendering	387
91	Add Compute-Based Occlusion Culling: Optimize scene rendering	391
92	Render Procedural Clouds: Create dynamic, realistic clouds	395
93	Create a Vulkan Benchmark Tool: Measure Vulkan performance on hardware	399
94	Render a Vulkan-Based UI Library: Develop a custom UI library	403
95	Create a Vulkan Music Visualizer: Sync graphics with music	407
96	Simulate Explosions: Render dynamic explosion effects	411
97	Add Vulkan Networking: Sync data across networks for multiplayer apps	415

98	Render Vulkan-Based Fur Simulation: Create realistic furry objects	419
99	Integrate Vulkan with WebAssembly: Run Vulkan in a web browser (experimental)	423
100	Simulate Cloth Physics: Use compute shaders for cloth simulation	426
101	Build a Vulkan-Based Ray Tracing Engine: Fully utilize Vulkan's ray tracing features	431
	Appendix	437

DRAFT
version
0.681



Content

Preface	v
Brief Contents	vi
Contents	xi
I Basic Vulkan Setup	4
1 Set Up Vulkan Instance: Create and manage a Vulkan instance	6
1.1 Introduction	7
1.2 Understanding the Vulkan Instance	7
1.3 Setting Up the Vulkan Instance	8
1.3.1 Including Vulkan Headers	8
1.3.2 Defining Extensions and Validation Layers	8
1.3.3 Creating the Vulkan Instance	8
1.3.4 Cleaning Up the Vulkan Instance	9
1.3.5 Complete Example Program	9
1.4 Summary	11
2 Enumerate Physical Devices: List available GPUs on your system	12
2.1 Introduction	13
2.2 Understanding Physical Devices	13
2.3 Enumerating Physical Devices	14
2.3.1 Step 1: Retrieve the Number of Physical Devices	14
2.3.2 Step 2: Retrieve the Device Handles	14
2.3.3 Step 3: Query Device Properties	14
2.3.4 Step 4: Query Device Features and Extensions	15
2.4 Complete Example Program	15
2.5 Summary	16
3 Enable Validation Layers: Debug Vulkan code with validation layers	17

3.1	Introduction	17
3.2	How Validation Layers Work	17
3.3	Step-by-Step Implementation	18
3.3.1	Step 1: Check Validation Layer Support	18
3.3.2	Step 2: Enable Validation Layers During Instance Creation	18
3.3.3	Step 3: Set Up a Debug Callback Function	19
3.3.4	Step 4: Create a Debug Messenger Object	19
3.4	Using Validation Layer Messages	20
3.5	Summary	20
4	Create Logical Device and Queues: Create logical devices and retrieve queue handles	21
4.1	Introduction	21
4.2	Understanding Logical Devices and Queues	21
4.3	Creating a Logical Device	22
4.3.1	Step 1: Identify Queue Families	22
4.3.2	Step 2: Select a Suitable Queue Family	22
4.3.3	Step 3: Specify Queue Creation Information	22
4.3.4	Step 4: Configure the Logical Device	23
4.3.5	Step 5: Retrieve Queue Handles	23
4.4	Complete Example Program	23
4.5	Summary	24
5	Present a Blank Swapchain Image: Present an empty window via Vulkan	25
5.1	Introduction	25
5.2	Understanding Swapchain Presentation	25
5.3	Step-by-Step Implementation	26
5.3.1	Step 1: Acquire a Swapchain Image	26
5.3.2	Step 2: Record a Command Buffer	26
5.3.3	Step 3: Submit the Command Buffer	27
5.3.4	Step 4: Present the Image	27
5.4	Complete Example Program	28
5.5	Rendering Output	28
5.6	Summary	28
6	Set Up a Vulkan Pipeline: Understand and implement the Vulkan graphics pipeline	29
6.1	Introduction	29
6.2	Understanding the Graphics Pipeline Stages	29
6.3	Step-by-Step Implementation	30
6.3.1	Step 1: Define Shader Stages	30
6.3.2	Step 2: Configure Fixed-Function Stages	30
6.3.3	Step 3: Create the Pipeline Layout	31
6.3.4	Step 4: Create the Graphics Pipeline	31
6.4	Complete Example	32
6.5	Summary	32
7	Clear the Screen with a Color: Implement screen-clearing using Vulkan commands	33
7.1	Introduction	33
7.2	Understanding Clear Values	33
7.3	Step-by-Step Implementation	34

7.3.1	Step 1: Define the Clear Color	34
7.3.2	Step 2: Update the Render Pass Configuration	34
7.3.3	Step 3: Specify Clear Values in the Render Pass Begin Info	34
7.3.4	Step 4: Record and Execute the Command Buffer	35
7.4	Complete Example Program	35
7.5	Rendering Output	35
7.6	Summary	36
8	Draw a Single Triangle (Solid Color): Render a triangle with a single color	37
8.1	Introduction	37
8.2	The Rendering Pipeline Overview	37
8.3	Step-by-Step Implementation	38
8.3.1	Step 1: Prepare a Swapchain and Render Pass	38
8.3.2	Step 2: Define the Triangle's Vertex Data	38
8.3.3	Step 3: Create a Vertex Buffer	38
8.3.4	Step 4: Write the Shaders	38
8.3.5	Step 5: Create a Graphics Pipeline	39
8.3.6	Step 6: Record Command Buffers	39
8.3.7	Step 7: Submit Commands and Present the Triangle	40
8.4	Complete Example Program	40
8.5	Summary	40
9	Render a Quad: Create and render a basic quad using two triangles	41
9.1	Introduction	41
9.2	Vertex Data for a Quad	41
9.3	Using an Index Buffer	42
9.4	Creating Buffers for Vertices and Indices	42
9.5	Updating the Graphics Pipeline	42
9.6	Recording Command Buffers	43
9.7	Complete Example Program	43
9.8	Rendering Output	43
9.9	Summary	43
10	Add a Depth Buffer: Integrate depth buffering for 3D scenes	44
10.1	Introduction	45
10.2	Understanding Depth Buffers	45
10.3	Step-by-Step Implementation	46
10.3.1	Step 1: Create the Depth Image	46
10.3.2	Step 2: Allocate Memory for the Depth Image	46
10.3.3	Step 3: Create a Depth Image View	47
10.3.4	Step 4: Attach the Depth Buffer to the Render Pass	47
10.3.5	Step 5: Enable Depth Testing in the Graphics Pipeline	47
10.4	Complete Example	48
10.5	Summary	48
II	Intermediate Vulkan Rendering	49
11	Use Descriptor Sets: Implement descriptor sets for texture binding	51
11.1	Introduction	51
11.2	Overview of Descriptor Sets	51

11.3	Step 1: Create a Descriptor Set Layout	52
11.4	Step 2: Create a Descriptor Pool	52
11.5	Step 3: Allocate and Write to Descriptor Sets	52
11.6	Step 4: Bind the Descriptor Set During Rendering	53
11.7	Using the Texture in Shaders	53
11.8	Summary	54
12	Draw a Textured Triangle: Add a basic texture to your triangle	55
12.1	Introduction	55
12.2	Steps to Add a Texture	55
12.2.1	Step 1: Load the Texture Image	56
12.2.2	Step 2: Create a Vulkan Image and Transfer the Data	56
12.2.3	Step 3: Create an Image View and Sampler	56
12.2.4	Step 4: Update Shaders to Handle Textures	57
12.2.5	Step 5: Update Vertex Data and Descriptor Sets	57
12.3	Complete Example	58
12.4	Summary	58
13	Animate Triangle Movement: Add dynamic transformation (e.g., rotate, scale)	59
13.1	Introduction	59
13.2	Steps to Animate a Triangle	59
13.2.1	Step 1: Create and Update a Transformation Matrix	60
13.2.2	Step 2: Pass the Matrix to the Vertex Shader	60
13.2.3	Step 3: Apply the Transformation in the Vertex Shader	61
13.2.4	Integrate Animation into the Render Loop	61
13.3	Summary	61
14	Add Push Constants: Use Push Constants for Efficient Data Passing to Shaders	62
14.1	Introduction	63
14.2	Setting Up Push Constants in Vulkan	63
14.2.1	Step 1: Define a Push Constant Range	64
14.2.2	Step 2: Update the Pipeline Layout	64
14.3	Using Push Constants in Shaders	64
14.4	Passing Push Constants During Rendering	65
14.5	Practical Example: Dynamic Object Color	65
14.6	Summary	65
15	Create a Uniform Buffer: Pass data (e.g., transformations) to the shader	67
15.1	Introduction	67
15.2	Understanding Uniform Buffers	67
15.3	Steps to Create and Use a Uniform Buffer	67
15.3.1	Step 1: Define and Create a Uniform Buffer	68
15.3.2	Step 2: Update the Uniform Buffer with Data	68
15.3.3	Step 3: Bind the Uniform Buffer to Shaders	69
15.3.4	Access the Uniform Buffer in Shaders	69
15.4	Summary	70
16	Use Descriptor Sets for Multiple Objects: Manage Data for Multiple Objects	71
16.1	Introduction	71

16.2	Understanding Descriptor Sets for Multiple Objects	71
16.3	Steps to Use Descriptor Sets for Multiple Objects	71
16.3.1	Step 1: Create a Uniform Buffer for Each Object	72
16.3.2	Step 2: Allocate Descriptor Sets for Each Object	72
16.3.3	Step 3: Update Descriptor Sets During Rendering	73
16.4	Accessing Data in the Shader	73
16.5	Summary	74
17	Render Multiple Objects: Draw multiple triangles or quads in a single pass	75
17.1	Introduction	75
17.2	Drawing Multiple Objects with Separate Draw Calls	75
17.2.1	Step 1: Create Per-Object Resources	75
17.2.2	Step 2: Render Objects in a Loop	76
17.3	Using Instanced Rendering for Multiple Objects	76
17.3.1	Step 1: Create an Instance Buffer	76
17.3.2	Step 2: Update the Instance Buffer	77
17.3.3	Step 3: Bind the Instance Buffer and Issue a Single Draw Call	77
17.4	Accessing Instance Data in the Shader	77
17.5	Summary	78
18	Add Staging Buffers: Optimize data transfer between CPU and GPU	79
18.1	Introduction	79
18.2	What Are Staging Buffers?	79
18.3	Steps to Use Staging Buffers	79
18.3.1	Step 1: Create a Host-Visible Staging Buffer	80
18.3.2	Step 2: Create a GPU-Visible Buffer	80
18.3.3	Step 3: Copy Data to the Staging Buffer	80
18.3.4	Step 4: Transfer Data to the GPU Buffer	81
18.4	Full Implementation Example	81
18.5	Performance Benefits of Staging Buffers	82
18.6	Summary	82
19	Generate a Procedural Gradient: Render a triangle with a procedurally generated gradient	83
19.1	Introduction	83
19.2	What is a Procedural Gradient?	83
19.3	Shader Implementation for Gradient	84
19.3.1	Vertex Shader	84
19.3.2	Fragment Shader	84
19.4	Vulkan Pipeline Setup	84
19.4.1	Vertex Data	84
19.4.2	Graphics Pipeline	85
19.4.3	Rendering Loop	85
19.5	Result	85
19.6	Summary	86
20	Add Anti-Aliasing: Use multisample anti-aliasing (MSAA)	87
20.1	Introduction	87
20.2	Overview of MSAA in Vulkan	87
20.3	Step 1: Determine Supported Sample Counts	88
20.4	Step 2: Create Multisampled Image Attachments	88

20.5	Step 3: Update the Render Pass	89
20.6	Step 4: Modify the Graphics Pipeline	90
20.7	Rendering with MSAA	90
20.8	Summary	90

III Advanced Rendering Techniques 91

21 Render to a Texture: Create an off-screen framebuffer 93

21.1	Introduction	93
21.2	Concept Overview	93
21.3	Step 1: Create an Off-Screen Image	94
21.4	Step 2: Create an Off-Screen Framebuffer	94
21.5	Step 3: Set Up an Off-Screen Render Pass	94
21.6	Step 4: Render to the Off-Screen Framebuffer	95
21.7	Step 5: Sample the Off-Screen Texture	96
21.8	Summary	96

22 Implement a Frame Graph: Manage frame dependencies 97

22.1	Introduction	97
22.2	Understanding a Frame Graph	97
22.3	Step 1: Define Frame Graph Resources	98
22.4	Step 2: Define Frame Graph Passes	98
22.5	Step 3: Build the Frame Graph	98
22.6	Step 4: Create Resources and Passes	99
22.7	Step 5: Execute the Frame Graph	100
22.8	Advantages of Using a Frame Graph	100
22.9	Summary	100

23 Deferred Rendering: Render multiple passes for better lighting quality 101

23.1	Introduction	101
23.2	Overview of Deferred Rendering	101
23.3	Step 1: Create the G-buffer	102
23.4	Step 2: Create the Geometry Pass	102
23.5	Step 3: Perform the Lighting Pass	103
23.6	Step 4: Combine Passes in the Frame Graph	104
23.7	Summary	104

24 Add Multiple Light Sources: Implement dynamic lighting in the scene 105

24.1	Introduction	105
24.2	Understanding Multiple Light Sources	105
24.3	Step 1: Define a Light Structure	106
24.4	Step 2: Pass Light Data to the Shader	106
24.5	Step 3: Modify the Lighting Shader	107
24.6	Step 4: Add Lights to the Scene	108
24.7	Performance Optimization	108
24.8	Summary	108

25 Physically Based Rendering (PBR): Start with simple PBR shaders 109

25.1	Introduction	109
25.2	Overview of PBR Concepts	109
25.3	Step 1: Extend the G-buffer for PBR	110

25.4	Step 2: Update Geometry Pass Shaders	110
25.5	Step 3: Implement the PBR Lighting Shader	111
25.6	Step 4: Integrate and Test	112
25.7	Summary	112
26	Implement Shadow Mapping: Add basic shadow rendering	113
26.1	Introduction	113
26.2	Overview of Shadow Mapping	113
26.3	Step 1: Create a Shadow Map Render Pass	113
26.4	Step 2: Create the Shadow Map Framebuffer	114
26.5	Step 3: Render the Scene to the Shadow Map	115
26.6	Step 4: Sample the Shadow Map in the Lighting Pass	115
26.7	Step 5: Handle Shadow Artifacts	116
26.8	Step 6: Integrate and Test	116
26.9	Summary	116
27	Omnidirectional Shadows: Implement cube map shadows for point lights	117
27.1	Introduction	117
27.2	Overview of Omnidirectional Shadow Mapping	117
27.3	Step 1: Create a Cube Map for Shadows	117
27.4	Step 2: Set Up a Shadow Map Render Pass and Framebuffer	118
27.5	Step 3: Render the Scene to Each Cube Map Face	118
27.6	Step 4: Sample the Cube Map in the Lighting Pass	119
27.7	Step 5: Optimize and Test	119
27.8	Summary	120
28	Create a Skybox: Render a cubemap as the background	121
28.1	Introduction	121
28.2	Overview of Skybox Rendering	121
28.3	Step 1: Load the Cubemap Texture	121
28.4	Step 2: Create the Skybox Pipeline	122
28.5	Step 3: Write the Skybox Shaders	122
28.6	Step 4: Render the Skybox	123
28.7	Step 5: Optimize the Skybox Rendering	123
28.8	Summary	123
29	Add Post-Processing Effects: Add effects like bloom or vignette	124
29.1	Introduction	125
29.2	Overview of Post-Processing in Vulkan	125
29.3	Step 1: Create an Off-Screen Framebuffer	126
29.4	Step 2: Render to the Off-Screen Framebuffer	126
29.5	Step 3: Set Up Post-Processing Pipeline	127
29.6	Step 4: Write Post-Processing Shaders	127
29.7	Step 5: Render the Post-Processed Image	128
29.8	Summary	128
30	Create a Particle System: Simulate particles using a vertex shader	129
30.1	Introduction	129
30.2	Overview of Particle Systems in Vulkan	129
30.3	Step 1: Define the Particle Structure	129
30.4	Step 2: Set Up the Graphics Pipeline	130
30.5	Step 3: Write the Vertex Shader	131

30.6	Step 4: Write the Fragment Shader	131
30.7	Step 5: Render the Particle System	131
30.8	Summary	132

IV Compute Shader Projects 133

31	Hello, Compute Shader: Write your first compute shader	135
31.1	Introduction	135
31.2	Overview of Compute Shaders in Vulkan	135
31.3	Step 1: Define the Input and Output Buffers	136
31.4	Step 2: Write the Compute Shader	136
31.5	Step 3: Create the Compute Pipeline	136
31.6	Step 4: Set Up Descriptor Sets	137
31.7	Step 5: Dispatch the Compute Shader	137
31.8	Step 6: Retrieve and Verify Results	138
31.9	Summary	138
32	Compute a Mandelbrot Set: Render fractals using compute shaders	139
32.1	Introduction	139
32.2	Overview of Mandelbrot Set Computation	139
32.3	Step 1: Set Up a Compute Shader for the Mandelbrot Set	139
32.4	Step 2: Create a Storage Image	140
32.5	Step 3: Set Up the Compute Pipeline	140
32.6	Step 4: Dispatch the Compute Shader	141
32.7	Step 5: Visualize the Result	141
32.8	Interactive Exploration	141
32.9	Summary	142
33	Add Image Blur Effect: Apply a Gaussian blur using compute shaders	143
33.1	Introduction	143
33.2	Overview of Gaussian Blur	143
33.3	Step 1: Prepare Input and Output Images	144
33.4	Step 2: Write the Compute Shader for Gaussian Blur	144
33.5	Step 3: Set Up the Compute Pipeline	145
33.6	Step 4: Dispatch Horizontal and Vertical Blur Passes	145
33.7	Step 5: Visualize the Result	145
33.8	Tuning the Blur Effect	146
33.9	Summary	146
34	Simulate Physics: Implement a simple physics engine with compute shaders	147
34.1	Introduction	147
34.2	Overview of the Physics Simulation	147
34.3	Step 1: Define the Particle Data Structure	148
34.4	Step 2: Write the Physics Compute Shader	148
34.5	Step 3: Create the SSBO and Initialize Particles	149
34.6	Step 4: Configure the Compute Pipeline	149
34.7	Step 5: Dispatch the Compute Shader	150
34.8	Step 6: Visualize the Particles	150
34.9	Interactive Features	150
34.10	Summary	150

35	Game of Life Simulation: Use compute shaders for cellular automata	151
35.1	Introduction	151
35.2	Overview of the Rules	151
35.3	Step 1: Define the Grid Data Structure	152
35.4	Step 2: Write the Compute Shader for the Game of Life	152
35.5	Step 3: Create and Configure the Compute Pipeline	153
35.6	Step 4: Execute the Compute Shader	153
35.7	Step 5: Visualize the Grid	154
35.8	Interactive Features	154
35.9	Summary	154
36	Generate Perlin Noise: Use compute shaders to create noise textures	155
36.1	Introduction	155
36.2	Understanding Perlin Noise	155
36.3	Step 1: Define the Compute Shader for Perlin Noise	155
36.4	Step 2: Create the Compute Pipeline	157
36.5	Step 3: Allocate and Bind an Image for Output	157
36.6	Step 4: Dispatch the Compute Shader	157
36.7	Step 5: Visualize the Noise Texture	158
36.8	Extending the Perlin Noise Implementation	158
36.9	Summary	158
37	Sort Data with Compute Shaders: Implement a parallel sorting algorithm	159
37.1	Introduction	159
37.2	Overview of Bitonic Sort	159
37.3	Step 1: Prepare the Input Data	159
37.4	Step 2: Write the Compute Shader for Bitonic Sort	160
37.5	Step 3: Create and Configure the Compute Pipeline	161
37.6	Step 4: Execute the Sorting Algorithm	161
37.7	Step 5: Retrieve and Verify the Sorted Data	161
37.8	Optimizations and Extensions	162
37.9	Summary	162
38	Create a GPU Particle System: Simulate particles entirely on the GPU	163
38.1	Introduction	163
38.2	Overview of a GPU Particle System	163
38.3	Step 1: Define Particle Data Structures	164
38.4	Step 2: Write the Compute Shader for Particle Simulation	164
38.5	Step 3: Create and Bind Descriptor Sets	165
38.6	Step 4: Render Particles Using the Graphics Pipeline	165
38.7	Step 5: Execute the Simulation and Rendering Loop	166
38.8	Extensions and Optimizations	166
38.9	Summary	166
39	Simulate Fluid Dynamics: Implement 2D fluid simulation	167
39.1	Introduction	167
39.2	Overview of Fluid Dynamics Simulation	167
39.3	Step 1: Set Up Grid Buffers	168
39.4	Step 2: Write the Compute Shader for Fluid Simulation	168
39.5	Step 3: Visualize the Simulation	169
39.6	Step 4: Combine Simulation and Rendering	169
39.7	Extensions and Optimizations	170

39.8	Summary	170
40	Generate a Procedural Heightmap: Create terrains with compute shaders	171
40.1	Introduction	171
40.2	Overview of Heightmap Generation	171
40.3	Step 1: Set Up the Heightmap Texture	172
40.4	Step 2: Write the Compute Shader for Heightmap Generation	172
40.5	Step 3: Visualize the Heightmap as a 3D Terrain	173
40.6	Step 4: Combine Compute and Graphics Pipelines	173
40.7	Extensions and Optimizations	174
40.8	Summary	174
V	Optimization Projects	175
41	Use Pipeline Caching: Reduce pipeline creation overhead	177
41.1	Introduction	177
41.2	Overview of Pipeline Caching	177
41.2.1	Save Pipeline Cache Data	178
41.2.2	Load Pipeline Cache Data	179
42	Add Command Buffer Reuse: Reuse recorded command buffers for efficiency	181
42.1	Introduction	181
42.2	Overview of Command Buffer Reuse	181
42.3	Step 1: Record the Command Buffer	182
42.4	Step 2: Submit the Command Buffer for Execution	182
42.5	Step 3: Use Secondary Command Buffers for Modularization	183
42.6	Step 4: Manage Synchronization	183
42.7	Benefits and Considerations	184
42.8	Summary	184
43	Implement Instanced Rendering: Draw multiple objects with fewer draw calls	185
43.1	Introduction	185
43.2	What is Instanced Rendering?	185
43.3	Step 1: Prepare Instance Data	186
43.4	Step 2: Update the Vertex Shader	186
43.5	Step 3: Bind Multiple Buffers	187
43.6	Step 4: Issue the Draw Call	187
43.7	Best Practices for Instanced Rendering	187
43.8	Summary	187
44	Optimize Descriptor Updates: Batch descriptor set updates	188
44.1	Introduction	189
44.2	Understanding Descriptor Updates	189
44.3	Step 1: Batch Descriptor Writes and Copies	190
44.4	Step 2: Use Dynamic Descriptors for Flexibility	190
44.5	Step 3: Use Descriptor Indexing for Large Resource Sets	191
44.6	Step 4: Optimize Descriptor Pool Allocation	192
44.7	Summary	192

45	Use Dynamic Buffers: Optimize buffer memory usage	193
45.1	Introduction	193
45.2	What are Dynamic Buffers?	193
45.3	Step 1: Create a Dynamic Buffer	194
45.4	Step 2: Update the Buffer with Per-Object Data	194
45.5	Step 3: Create and Bind a Dynamic Descriptor	194
45.6	Step 4: Issue Draw Calls with Dynamic Offsets	195
45.7	Advantages of Dynamic Buffers	195
45.8	Limitations and Considerations	195
45.9	Summary	196
46	Asynchronous Compute and Graphics: Use separate queues for compute and graphics	197
46.1	Introduction	197
46.2	Understanding Queue Families and Parallel Execution	197
46.3	Step 1: Query and Select Queue Families	198
46.4	Step 2: Create Logical Device with Multiple Queues	198
46.5	Step 3: Record Command Buffers for Compute and Graphics	199
46.6	Step 4: Synchronize Queues with Semaphores	199
46.7	Advantages of Asynchronous Compute and Graphics	200
46.8	Limitations and Considerations	200
46.9	Summary	200
47	Measure Performance with Queries: Add timestamp queries to measure GPU time	201
47.1	Introduction	201
47.2	What are Timestamp Queries?	201
47.3	Step 1: Create a Query Pool	201
47.4	Step 2: Record Timestamps in the Command Buffer	202
47.5	Step 3: Retrieve Query Results	202
47.6	Best Practices for Using Timestamp Queries	203
47.7	Limitations of Timestamp Queries	203
47.8	Summary	203
48	Use Vulkan Subpasses: Optimize render passes with subpasses	204
48.1	Introduction	205
48.2	What Are Subpasses?	205
48.3	Step 1: Define Subpasses in a Render Pass	206
48.4	Step 2: Add Dependencies Between Subpasses	206
48.5	Step 3: Record Commands for Subpasses	207
48.6	Advantages of Using Subpasses	207
48.7	Limitations and Considerations	208
48.8	Summary	208
49	Implement a Frame Limiter: Limit the FPS for consistency	209
49.1	Introduction	209
49.2	Why Use a Frame Limiter?	209
49.3	Step 1: Understand Timing in Vulkan	210
49.4	Step 2: Calculate the Target Frame Time	210
49.5	Step 3: Measure Frame Execution Time	210
49.6	Step 4: Implement Frame Delay	210
49.7	Step 5: Integrate into the Main Loop	211

49.8	Considerations and Best Practices	211
49.9	Summary	211
50	Use Device Extensions: Enable advanced GPU features via extensions	212
50.1	Introduction	213
50.2	What Are Device Extensions?	213
50.3	Step 1: Query Supported Extensions	214
50.4	Step 2: Specify Required Extensions	214
50.5	Step 3: Check for Extension Support	214
50.6	Step 4: Enable Extensions During Logical Device Creation	215
50.7	Common Device Extensions	215
50.8	Considerations When Using Extensions	215
50.9	Conclusion	216
VI	Advanced Vulkan Features	217
51	Ray Tracing Basics: Render a simple scene with ray tracing	219
51.1	Introduction	219
51.2	Overview of Vulkan Ray Tracing	219
51.3	Step 1: Enable Ray Tracing Extensions	220
51.4	Step 2: Create Acceleration Structures	220
51.5	Step 3: Create the Ray Tracing Pipeline	221
51.6	Step 4: Create and Use Shader Binding Tables (SBTs)	222
51.7	Step 5: Dispatch Rays	222
51.8	Summary	222
52	Add BVH Acceleration Structures: Create bounding volume hierarchies	223
52.1	Introduction	223
52.2	Overview of Acceleration Structures in Vulkan	223
52.3	Step 1: Define the Geometry for BLAS	224
52.4	Step 2: Build the BLAS	224
52.5	Step 3: Build the TLAS	225
52.6	Step 4: Integrate BVH with Ray Tracing Pipeline	226
52.7	Summary	226
53	Render Reflections: Add ray-traced reflections	227
53.1	Introduction	227
53.2	Reflection Principles	227
53.3	Step 1: Extend the Ray Tracing Pipeline	228
53.4	Step 2: Update Descriptor Sets	229
53.5	Step 3: Integrate Reflections into the Render Pass	229
53.6	Step 4: Optimize Reflection Performance	229
53.7	Summary	229
54	Render Refractions: Implement glass-like materials	230
54.1	Introduction	231
54.2	Understanding Refraction	231
54.3	Step 1: Extend the Ray Tracing Shaders	232
54.4	Step 2: Update Vulkan Resources	233
54.5	Step 3: Integrate Refractions into the Render Pass	233
54.6	Step 4: Optimize Refractions	233

54.7	Summary	234
55	Path Tracing Basics: Render a simple path-traced scene	235
55.1	Introduction	235
55.2	What is Path Tracing?	235
55.3	Step 1: Extend the Ray Tracing Pipeline for Path Tracing	236
55.4	Step 2: Configure Vulkan Resources	237
55.5	Step 3: Integrate Path Tracing into the Render Loop	238
55.6	Optimizations and Challenges	238
55.7	Summary	238
56	Simulate Global Illumination: Add realistic lighting effects	239
56.1	Introduction	239
56.2	What is Global Illumination?	239
56.3	Step 1: Extend Path Tracing for GI	240
56.4	Step 2: Implement Light Sampling Techniques	241
56.5	Step 3: Optimize for Performance	241
56.6	Step 4: Integrate into Vulkan's Render Loop	242
56.7	Summary	242
57	Implement Vulkan Video Decode: Use Vulkan for video decoding	243
57.1	Introduction	243
57.2	What is Vulkan Video?	243
57.3	Prerequisites	244
57.4	Step 1: Enable Vulkan Video Extensions	244
57.5	Step 2: Set Up Video Decode Resources	244
57.6	Step 3: Prepare the Video Decode Pipeline	245
57.7	Step 4: Submit Video Decode Commands	245
57.8	Step 5: Use Decoded Frames in Rendering	245
57.9	Optimizations	246
57.10	Summary	246
58	Render HDR Content: Render high-dynamic-range content	247
58.1	Introduction	247
58.2	What is HDR Rendering?	247
58.3	Step 1: Configure the Swapchain for HDR	248
58.4	Step 2: Use HDR-Compatible Framebuffers	248
58.5	Step 3: Implement Tone Mapping in Shaders	248
58.6	Step 4: Set Proper Color Spaces and Metadata	249
58.7	Step 5: Debug and Test HDR Output	249
58.8	Summary	250
59	Add Vulkan Mesh Shaders: Experiment with next-gen mesh shaders	251
59.1	Introduction	251
59.2	What Are Mesh Shaders?	251
59.3	Step 1: Enable Mesh Shader Extensions	252
59.4	Step 2: Create a Pipeline with Mesh Shaders	252
59.5	Step 3: Write Mesh and Task Shaders	252
59.6	Step 4: Dispatch Mesh Shaders	253
59.7	Step 5: Debug and Optimize	253
59.8	Summary	254

60	Add Tiled Rendering: Use tile-based techniques for mobile GPUs	255
60.1	Introduction	255
60.2	What is Tiled Rendering?	255
60.3	Step 1: Design Render Passes for Tiled Rendering	256
60.4	Step 2: Utilize Load and Store Operations	256
60.5	Step 3: Optimize Shaders for Tiles	256
60.6	Step 4: Manage Tile-Specific Resources	257
60.7	Step 5: Test and Debug Tiled Rendering	257
60.8	Summary	257
VII	Cross-Platform Development	259
61	Add GLFW for Windowing: Use GLFW for cross-platform Vulkan windows	261
61.1	Introduction	261
61.2	What is GLFW?	261
61.3	Step 1: Install GLFW	261
61.4	Step 2: Initialize GLFW	262
61.5	Step 3: Create a GLFW Window	262
61.6	Step 4: Retrieve Vulkan Extensions from GLFW	262
61.7	Step 5: Create a Vulkan Surface with GLFW	263
61.8	Step 6: Handle Window Events	263
61.9	Step 7: Clean Up Resources	263
61.10	Summary	264
62	Add SDL2 for Windowing: Integrate SDL2 with Vulkan	265
62.1	Introduction	265
62.2	Why SDL2?	265
62.3	Step 1: Install SDL2	265
62.4	Step 2: Initialize SDL2	266
62.5	Step 3: Retrieve Vulkan Extensions from SDL2	266
62.6	Step 4: Create a Vulkan Surface with SDL2	267
62.7	Step 5: Handle SDL2 Events	267
62.8	Step 6: Clean Up Resources	267
62.9	Summary	268
63	Cross-Compile for Android: Build a Vulkan app for Android	269
63.1	Introduction	269
63.2	Overview of Vulkan on Android	269
63.3	Step 1: Set Up the Android Development Environment	269
63.4	Step 2: Create an Android Project	270
63.5	Step 3: Write Vulkan Initialization Code for Android	270
63.6	Step 4: Configure Build Files	271
63.7	Step 5: Cross-Compile the Application	271
63.8	Step 6: Deploy and Test on an Android Device	271
63.9	Step 7: Debug and Optimize	272
63.10	Summary	272
64	Test on macOS (MoltenVK): Run Vulkan apps on macOS using MoltenVK	273
64.1	Introduction	273
64.2	Why Use MoltenVK?	273

64.3	Step 1: Install the Vulkan SDK with MoltenVK	273
64.4	Step 2: Modify Your Vulkan Application for macOS	274
64.5	Step 3: Configure the Build System	275
64.6	Step 4: Build and Run on macOS	275
64.7	Step 5: Debugging and Validation Layers	275
64.8	Step 6: Optimize for Metal via MoltenVK	275
64.9	Summary	276
65	Cross-Compile for Raspberry Pi: Use Vulkan on a Raspberry Pi	277
65.1	Introduction	277
65.2	Why Use Vulkan on Raspberry Pi?	277
65.3	Step 1: Set Up the Raspberry Pi	278
65.4	Step 2: Set Up a Cross-Compilation Environment	278
65.5	Step 3: Configure Vulkan Application for Raspberry Pi	279
65.6	Step 4: Cross-Compile the Application	279
65.7	Step 5: Deploy and Run on the Raspberry Pi	279
65.8	Step 6: Debugging and Performance Optimization	280
65.9	Summary	280
66	Test on Linux: Set up a Vulkan development environment for Linux	281
66.1	Introduction	281
66.2	Why Develop Vulkan Applications on Linux?	281
66.3	Step 1: Install Vulkan Drivers	282
66.4	Step 2: Install the Vulkan SDK	282
66.5	Step 3: Configure Your Build Environment	283
66.6	Step 4: Create a Vulkan Surface on Linux	283
66.7	Step 5: Test and Debug on Linux	284
66.8	Summary	284
67	Use Vulkan in Unity: Integrate Vulkan into Unity projects	285
67.1	Introduction	285
67.2	Why Use Vulkan in Unity?	285
67.3	Step 1: Check Unity Version and GPU Support	285
67.4	Step 2: Enable Vulkan in Unity	286
67.5	Step 3: Test Vulkan Rendering in the Unity Editor	286
67.6	Step 4: Build and Test on Target Platforms	286
67.7	Step 5: Debug and Optimize Vulkan in Unity	287
67.8	Step 6: Leverage Vulkan-Specific Features	287
67.9	Common Issues and Troubleshooting	287
67.10	Summary	287
68	Integrate Vulkan with Unreal Engine: Use Vulkan in Unreal Engine	288
68.1	Introduction	289
68.2	Why Use Vulkan with Unreal Engine?	289
68.3	Step 1: Verify System Requirements	289
68.4	Step 2: Enable Vulkan in Unreal Engine	290
68.5	Step 3: Test Vulkan Rendering in the Editor	290
68.6	Step 4: Build and Test on Target Platforms	290
68.7	Step 5: Debug and Optimize Vulkan in Unreal Engine	291
68.8	Step 6: Experiment with Vulkan-Specific Features	291
68.9	Common Issues and Troubleshooting	292
68.10	Summary	292

VIII Fun and Interactive Projects

293

69 Create a Vulkan Game Engine: Start with a simple engine core	295
69.1 Introduction	295
69.2 Why Build a Vulkan Game Engine?	295
69.3 Step 1: Design the Engine Structure	296
69.4 Step 2: Initialize the Engine Core	296
69.4.1 Engine Class Structure	296
69.4.2 Initialization Routine	296
69.4.3 Main Loop and Cleanup	296
69.5 Step 3: Set Up Vulkan for Rendering	297
69.5.1 Graphics Module Class Structure	297
69.5.2 Vulkan Initialization	297
69.5.3 Rendering Frames	298
69.6 Step 4: Modular Input Handling	298
69.7 Step 5: Build a Simple Demo Scene	298
69.8 Step 6: Plan for Future Features	298
69.9 Summary	299
70 Build a Vulkan Game (e.g., Pong): Create a basic 2D game	300
70.1 Introduction	301
70.2 Game Design Overview	301
70.3 Step 1: Initialize Vulkan and Set Up the Game Loop	302
70.3.1 Engine Initialization	302
70.3.2 Game Loop Structure	302
70.4 Step 2: Render Game Elements	302
70.4.1 Vertex Data for 2D Shapes	302
70.4.2 Rendering the Scene	303
70.5 Step 3: Implement Input Handling	303
70.6 Step 4: Add Game Logic	303
70.6.1 Ball Movement and Collision	303
70.6.2 Collision Detection	304
70.7 Step 5: Display the Score	304
70.8 Step 6: Test and Optimize	304
70.9 Summary	304
71 Render a 3D Model: Load and render 3D models (e.g., OBJ or GLTF)	305
71.1 Introduction	305
71.2 Step 1: Understand the 3D Model Format	305
71.3 Step 2: Load the Model Data	306
71.4 Step 3: Create Vulkan Buffers	306
71.4.1 Vertex Buffer	306
71.4.2 Index Buffer	307
71.5 Step 4: Update the Pipeline for 3D Rendering	307
71.5.1 Vertex Shader (GLSL)	307
71.5.2 Fragment Shader (GLSL)	307
71.6 Step 5: Render the Model	308
71.7 Step 6: Add Transformations and Camera Controls	308
71.8 Step 7: Test and Extend	308
71.9 Summary	308
72 Add Skeletal Animation: Render animated 3D characters	309

72.1	Introduction	309
72.2	Step 1: Understand the Skeleton and Skinning Process	309
72.3	Step 2: Load a Rigged Model	310
72.4	Step 3: Pass Bone Transformations to the GPU	310
72.4.1	Uniform Buffer for Bone Data	310
72.5	Step 4: Update the Vertex Shader for Skinning	311
72.5.1	Vertex Shader (GLSL)	311
72.6	Step 5: Update the Rendering Code	311
72.7	Step 6: Render the Animated Model	312
72.8	Step 7: Test and Extend	312
72.9	Summary	312
73	Simulate a Water Surface: Use compute shaders for water ripples	313
73.1	Introduction	313
73.2	Step 1: Understand the Wave Equation for Water Ripples	313
73.3	Step 2: Create Buffers for the Heightfield	314
73.4	Step 3: Implement the Compute Shader for Wave Simulation	314
73.4.1	Compute Shader (GLSL)	314
73.5	Step 4: Dispatch the Compute Shader	315
73.6	Step 5: Render the Water Surface	315
73.6.1	Vertex Shader (GLSL)	315
73.6.2	Fragment Shader (GLSL)	316
73.7	Step 6: Add Interactivity	316
73.8	Summary	316
74	Render a Procedural Planet: Generate planets using Vulkan	317
74.1	Introduction	317
74.2	Step 1: Generate a Spherical Mesh	317
74.2.1	UV Sphere Mesh Generation	317
74.3	Step 2: Add Procedural Terrain	318
74.3.1	Compute Shader for Height Displacement	318
74.3.2	Dispatch the Compute Shader	319
74.4	Step 3: Render the Planet	319
74.4.1	Vertex Shader for Spherical Transformations	319
74.4.2	Fragment Shader for Shading and Atmosphere	319
74.5	Step 4: Add Rotation and Animation	320
74.6	Step 5: Combine with Stars or a Skybox	320
74.7	Conclusion	320
75	Add an Interactive GUI: Use a library like Dear ImGui	321
75.1	Introduction	321
75.2	Step 1: Integrate Dear ImGui	321
75.2.1	Add ImGui Files	321
75.3	Step 2: Initialize ImGui with Vulkan	322
75.3.1	Create a Descriptor Pool for ImGui	322
75.3.2	Initialize Dear ImGui Context	322
75.4	Step 3: Render ImGui in Vulkan	323
75.4.1	Update Command Buffers	323
75.4.2	Ensure Synchronization	323
75.5	Step 4: Customize the GUI	323
75.6	Step 5: Cleanup	324
75.7	Summary	324

76	Simulate Fireworks: Create a dynamic fireworks display	325
76.1	Introduction	325
76.2	Step 1: Design the Particle System	325
76.3	Step 2: Create a Compute Shader for Particle Updates	326
76.4	Step 3: Initialize Particles in Vulkan	326
76.5	Step 4: Set Up the Graphics Pipeline	327
76.6	Step 5: Simulate Fireworks	327
76.7	Summary	328
77	Render a Terrain from Heightmaps: Load heightmap data for terrain rendering	329
77.1	Introduction	329
77.2	Step 1: Understanding Heightmaps	329
77.3	Step 2: Load the Heightmap	330
77.4	Step 3: Generate Terrain Mesh	330
77.5	Step 4: Create Buffers for the Terrain	331
77.6	Step 5: Render the Terrain	331
77.7	Step 6: Improve Terrain Rendering	332
77.8	Summary	332
78	Add VR Support: Use Vulkan for VR rendering	333
78.1	Introduction	333
78.2	Step 1: Understanding VR Rendering	333
78.3	Step 2: Set Up a VR Runtime	334
78.4	Step 3: Create Vulkan Instance and Surface for VR	334
78.5	Step 4: Set Up Stereoscopic Rendering	335
78.6	Step 5: Synchronize with Head Tracking	335
78.7	Step 6: Render and Present to the VR Display	336
78.8	Summary	336
79	Build an Interactive Solar System: Simulate planetary orbits	337
79.1	Introduction	337
79.2	Rendering Spheres for Planets and Moons	337
79.3	Simulating Orbital Mechanics	338
79.4	Hierarchical Transformations	338
79.5	Adding Interactivity	338
79.6	Rendering the Solar System	339
79.7	Summary	339
80	Create a Vulkan Screensaver: Use Vulkan for a graphical screensaver	340
80.1	Introduction	341
80.2	Setting Up the Vulkan Application	341
80.3	Designing the Screensaver Scene	342
80.3.1	Particle System Generation	342
80.3.2	Animating Particles	342
80.4	Rendering the Screensaver	343
80.4.1	Post-Processing Effects	343
80.5	Idle Mode Optimization	343
80.6	Summary	343

IX Tools and Integration Projects 344

81 Integrate with OpenCV: Process Vulkan-rendered images with OpenCV	346
81.1 Introduction	347
81.2 Capturing Vulkan-Rendered Images	347
81.2.1 Creating the Staging Buffer	347
81.2.2 Copying Image Data to the Buffer	348
81.2.3 Mapping the Buffer to CPU Memory	348
81.3 Processing with OpenCV	348
81.3.1 Converting Vulkan Image to <code>cv::Mat</code>	348
81.3.2 Applying OpenCV Processing	349
81.3.3 Displaying or Saving the Processed Image	349
81.4 Integrating the Workflow	349
81.5 Summary	350
82 Export Frames to Images: Save Vulkan-rendered frames as PNG files	351
82.1 Introduction	351
82.2 Capturing Rendered Frames	351
82.2.1 Preparing a Staging Buffer	351
82.2.2 Copying the Image Data	352
82.2.3 Mapping the Staging Buffer	352
82.3 Saving Frames as PNG Images	352
82.3.1 Integrating <code>stb_image_write</code>	352
82.3.2 Saving the Image	353
82.3.3 Alternative Libraries	353
82.4 Automating Frame Export	353
82.5 Summary	353
83 Integrate Vulkan with TensorFlow: Use Vulkan for ML model inference	354
83.1 Introduction	355
83.2 Overview of Vulkan and TensorFlow Integration	355
83.3 Setting Up Vulkan for TensorFlow	356
83.3.1 Configuring TensorFlow for External Operations	356
83.4 Writing a Custom TensorFlow Operation	356
83.4.1 Define the Custom Operation	356
83.4.2 Implement the Vulkan Kernel	356
83.4.3 Register the Operation Kernel	357
83.5 Integrating with a TensorFlow Model	357
83.5.1 Building TensorFlow with the Custom Operation	358
83.5.2 Using the Operation in Python	358
83.6 Summary	358
84 Build a Vulkan Render Doc Clone: Debug Vulkan calls visually	359
84.1 Introduction	359
84.2 Overview of the Approach	359
84.3 Setting Up a Vulkan Debug Layer	359
84.3.1 Creating a Custom Vulkan Layer	360
84.3.2 Loading the Vulkan Layer	360
84.4 Recording Frame Data	361
84.4.1 Intercepting Command Buffers	361
84.4.2 Capturing Textures and Buffers	361
84.5 Building a Visual Debugger	362

84.5.1	Creating the Frame Viewer	362
84.5.2	Rendering Captured Textures	362
84.6	Summary	362
85	Add Vulkan Support to Blender: Create a plugin for Vulkan rendering	363
85.1	Introduction	363
85.2	Overview of Blender Rendering Plugins	363
85.3	Set Up Blender Development Environment	364
85.3.1	Dependencies and Build Tools	364
85.3.2	Compiling Blender with Plugin Support	364
85.4	Creating the Vulkan Rendering Context	364
85.4.1	Integrating Vulkan Initialization	364
85.4.2	Viewport Rendering	365
85.5	Exporting Scene Data for Vulkan Rendering	365
85.5.1	Exporting Mesh Data	365
85.5.2	Exporting Materials and Textures	366
85.6	Rendering the Scene	366
85.7	Testing the Plugin	366
85.8	Summary	366
86	Integrate Vulkan with Qt: Build Vulkan-based GUI applications with Qt	367
86.1	Introduction	367
86.2	Why Use Qt with Vulkan?	367
86.3	Setting Up a Qt Project with Vulkan	368
86.3.1	Install Qt	368
86.3.2	Create a New Qt Application	368
86.3.3	Add Vulkan SDK to the Project	368
86.4	Creating a Vulkan Window in Qt	368
86.4.1	Define a Custom Vulkan Window	368
86.4.2	Initialize Vulkan Resources	369
86.4.3	Render a Triangle	370
86.5	Integrating Vulkan Window with Qt Widgets	370
86.6	Advanced Features	370
86.7	Summary	371
87	Create Vulkan Debugging Tools: Write tools for visualizing Vulkan resources	372
87.1	Introduction	373
87.2	Why Create Debugging Tools?	373
87.3	Setting Up the Debugging Tool Framework	374
87.3.1	Integrating Dear ImGui with Vulkan	374
87.4	Displaying Vulkan Resource Information	374
87.4.1	Buffer Viewer	375
87.4.2	Texture Viewer	375
87.4.3	Pipeline Viewer	375
87.5	Advanced Debugging Features	376
87.6	Summary	376
X	Miscellaneous Projects	377
88	Render a Wireframe Model: Add wireframe rendering	379

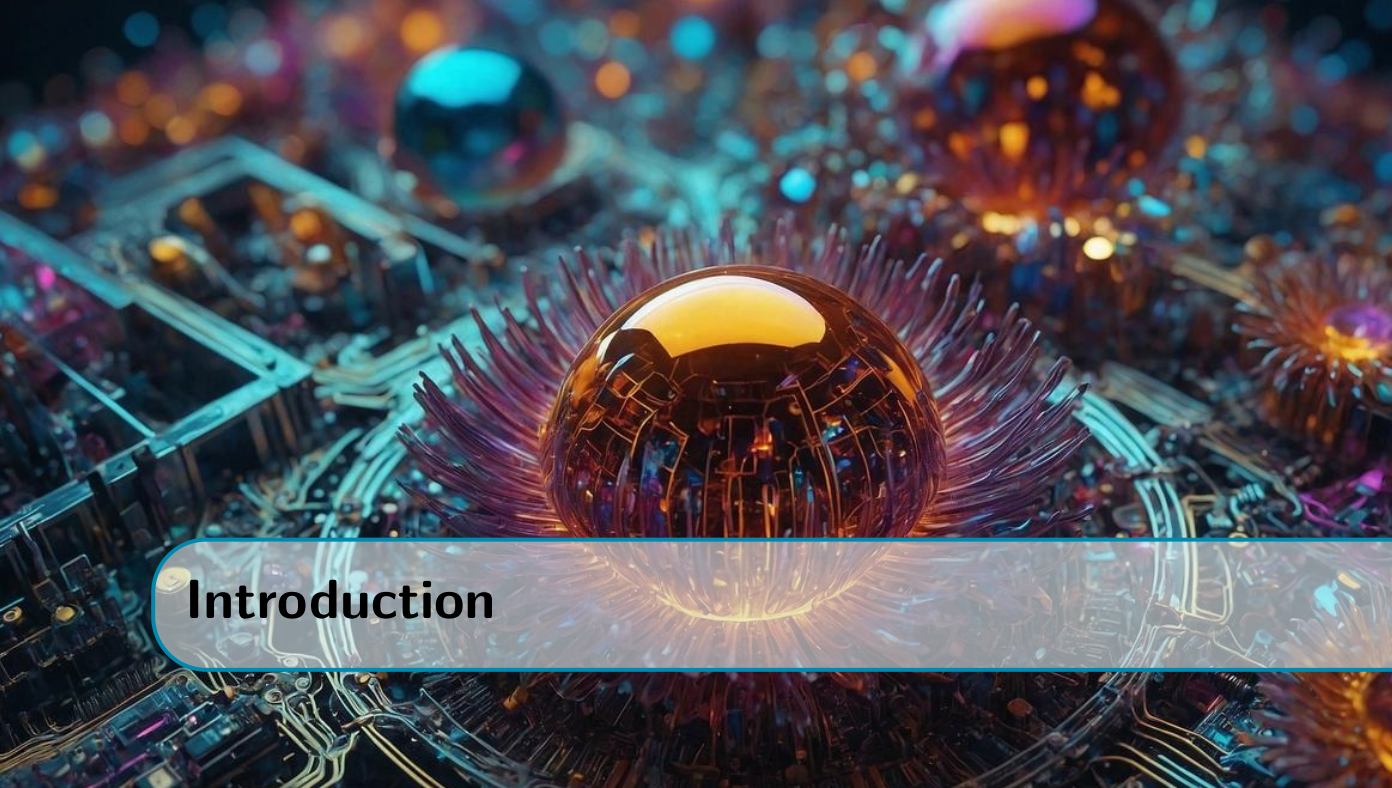
88.1	Introduction	379
88.2	Setting Up Wireframe Rendering	379
88.2.1	Configuring the Rasterization State	379
88.2.2	Recreating the Graphics Pipeline	380
88.3	Rendering in Wireframe Mode	380
88.4	Dynamic Wireframe Toggling	380
88.5	Adjusting Line Width	381
88.6	Practical Applications of Wireframe Rendering	381
88.7	Summary	381
89	Add Tessellation Shaders: Implement tessellation for smoother models	382
89.1	Introduction	383
89.2	Overview of Tessellation Stages	383
89.3	Enabling Tessellation in Vulkan	384
89.3.1	Updating the Pipeline State	384
89.3.2	Specifying Patch Topology	384
89.4	Writing Tessellation Shaders	385
89.4.1	Tessellation Control Shader (TCS)	385
89.4.2	Tessellation Evaluation Shader (TES)	385
89.5	Drawing Tessellated Geometry	385
89.6	Dynamic Tessellation Levels	386
89.7	Summary	386
90	Render Text with Vulkan: Use a font library for text rendering	387
90.1	Introduction	387
90.2	Overview of Text Rendering	387
90.3	Setting Up FreeType for Font Loading	388
90.4	Creating a Glyph Texture Atlas	388
90.4.1	Generating the Atlas	388
90.4.2	Sampling the Texture in Shaders	389
90.5	Placing Glyphs on the Screen	389
90.6	Integrating with Vulkan	389
90.7	Summary	390
91	Add Compute-Based Occlusion Culling: Optimize scene rendering	391
91.1	Introduction	391
91.2	What is Occlusion Culling?	391
91.3	Overview of the Implementation	391
91.4	Preparing the Data	392
91.5	Writing the Compute Shader	392
91.6	Dispatching the Compute Shader	393
91.7	Using the Results to Filter Draw Commands	393
91.8	Integrating with Vulkan Pipelines	394
91.9	Summary	394
92	Render Procedural Clouds: Create dynamic, realistic clouds	395
92.1	Introduction	395
92.2	Understanding Procedural Clouds	395
92.3	Setting Up the Compute Pipeline for 3D Noise	396
92.4	Raymarching to Render Clouds	396
92.5	Lighting the Clouds	397
92.6	Integrating into the Vulkan Pipeline	397

92.7	Summary	398
93	Create a Vulkan Benchmark Tool: Measure Vulkan performance on hardware	399
93.1	Introduction	399
93.2	Purpose of a Vulkan Benchmark Tool	399
93.3	Framework Overview	399
93.4	Setting Up Timestamp Queries	400
93.5	Calculating FPS and Metrics	401
93.6	Visualizing the Metrics	401
93.7	Exporting Results	401
93.8	Advanced Features	401
93.9	Summary	402
94	Render a Vulkan-Based UI Library: Develop a custom UI library	403
94.1	Introduction	403
94.2	Goals of a Custom UI Library	403
94.3	Framework Overview	403
94.4	Setting Up the UI Rendering System	404
94.5	Handling Input Events	405
94.6	Advanced Features for the UI Library	405
94.7	Performance Considerations	406
94.8	Summary	406
95	Create a Vulkan Music Visualizer: Sync graphics with music	407
95.1	Introduction	407
95.2	Overview of the Music Visualizer	407
95.3	Audio Analysis	408
95.4	Integrating Audio with Vulkan Graphics	408
95.5	Rendering Visual Effects	409
95.6	Performance Considerations	410
95.7	Expanding the Visualizer	410
95.8	Summary	410
96	Simulate Explosions: Render dynamic explosion effects	411
96.1	Introduction	411
96.2	Overview of Explosion Effects	411
96.3	Creating a Particle System for Explosions	412
96.4	Simulating Particle Dynamics	412
96.5	Rendering Explosion Effects	413
96.6	Expanding the Effect	414
96.7	Performance Optimization	414
96.8	Summary	414
97	Add Vulkan Networking: Sync data across networks for multiplayer apps	415
97.1	Introduction	415
97.2	Networking Basics	415
97.3	Choosing a Networking Library	416
97.4	Setting Up the Networking Layer	416
97.5	Synchronizing Vulkan Data	417
97.6	Handling Latency	417
97.7	Debugging and Optimization	418

97.8	Summary	418
98	Render Vulkan-Based Fur Simulation: Create realistic furry objects	419
98.1	Introduction	419
98.2	Overview of Fur Simulation	419
98.3	Step 1: Fur Geometry Generation	420
98.4	Step 2: Rendering Fur Layers	420
98.5	Step 3: Adding Dynamics and Interactivity	421
98.6	Optimizing Performance	421
98.7	Summary	422
99	Integrate Vulkan with WebAssembly: Run Vulkan in a web browser (experimental)	423
99.1	Introduction	423
99.2	Overview of WebAssembly	423
99.3	Step 1: Set Up the Development Environment	423
99.4	Step 2: Adjust Your Vulkan Application for WebAssembly	424
99.5	Step 3: Compile the Application to WebAssembly	424
99.6	Step 4: Create the HTML Wrapper	424
99.7	Step 5: Run and Test in a Browser	425
99.8	Optimizing for WebAssembly	425
99.9	Summary	425
100	Simulate Cloth Physics: Use compute shaders for cloth simulation	426
100.1	Introduction	427
100.2	Understanding Cloth Simulation	427
100.3	Step 1: Define the Particle Grid	428
100.4	Step 2: Write the Compute Shader	428
100.5	Step 3: Handle Constraints	429
100.6	Step 4: Integrate with Vulkan Pipeline	429
100.7	Step 5: Render the Cloth	429
100.8	Step 6: Add Interactivity and Forces	429
100.9	Summary	430
101	Build a Vulkan-Based Ray Tracing Engine: Fully utilize Vulkan's ray tracing features	431
101.1	Introduction	431
101.2	Overview of Vulkan Ray Tracing	431
101.3	Step 1: Initialize Ray Tracing Extensions	432
101.4	Step 2: Build Acceleration Structures	432
101.5	Step 3: Write Ray Tracing Shaders	433
101.6	Step 4: Build the Ray Tracing Pipeline	434
101.7	Step 5: Dispatch Rays and Render	434
101.8	Step 6: Add Advanced Features	434
101.9	Summary	435
Appendix		437
A.1	Further Reading	437
A.2	Vulkan Compute Shader Key Functions/Members	437
A.3	GLSL Compute Shader Reserved Words	438
A.4	GLSL Compute Shader Keywords	438
A.5	GLSL Compute Shader Built-in Variables	438

A.6	GLSL Compute Shader Syntactic Tokens	438
A.7	GLSL Compute Shader Built-in Functions	439
A.8	GLM Library	439

DRAFT
version
0.681



Introduction

Introduction

The Vulkan API represents a new era of graphics and compute programming. Designed by the Khronos Group, Vulkan offers developers unprecedented control over the GPU, enabling high-performance, low-overhead programming across a wide range of platforms. Whether you're developing a game engine, a scientific simulation, or a real-time rendering system, Vulkan provides the tools to push hardware to its limits. However, its explicit nature and steep learning curve make it a challenging API to master.

That's where this book comes in. **101 Project Examples with Vulkan API** is a hands-on guide designed to help you learn Vulkan effectively, one project at a time. Through carefully structured examples, this book will take you from your first triangle to advanced techniques like ray tracing and compute-based simulations. Each project introduces key concepts in a practical, engaging way, so you'll build a solid understanding of Vulkan while having fun.

Why Learn Vulkan?

Vulkan has become a favorite API for developers who want:

- **Control:** Fine-grained control over rendering and compute operations.
- **Performance:** Low-overhead, multi-threaded capabilities for maximizing GPU utilization.
- **Cross-Platform Support:** A single API for Windows, Linux, Android, and more.
- **Future-Proofing:** Support for advanced features like ray tracing and mesh shading.

Learning Vulkan is more than just learning an API—it's an opportunity to develop a deeper understanding of how modern GPUs work and how to write high-performance, scalable code.

How This Book Is Structured

This book is organized into 101 projects, divided into three categories:

1. Fundamentals (Projects 1–30)

These projects focus on the basics, such as setting up Vulkan, drawing simple shapes, and understanding the rendering pipeline. You'll start with a single triangle and gradually introduce features like textures, shaders, and depth buffering.

2. Intermediate (Projects 31–70)

The intermediate section explores more complex topics like multiple render passes, deferred rendering, and dynamic objects. You'll also learn how to use compute shaders for tasks like physics simulations and post-processing effects.

3. Advanced (Projects 71–101)

Here, we tackle cutting-edge techniques, including ray tracing, global illumination, and real-time reflections. These projects will stretch your understanding of Vulkan and help you master its most advanced features.

Who Is This Book For?

This book is designed for:

- **Beginners:** If you're new to Vulkan, the structured, hands-on approach will guide you through the fundamentals step by step.
- **Intermediate Developers:** Expand your knowledge with challenging projects that deepen your understanding of Vulkan's capabilities.
- **Advanced Users:** Use the later projects to explore Vulkan's latest features and as a reference for implementing cutting-edge techniques.
- **Educators and Students:** A perfect resource for assignments, assessments, and guided learning in graphics programming courses.

What You'll Learn

Through this book, you'll:

- Develop a strong foundation in Vulkan's graphics and compute capabilities.
- Gain hands-on experience with 101 unique projects, ranging from basic to advanced.
- Learn both the theory behind Vulkan and practical tips for implementing efficient, maintainable code.
- Explore advanced features like ray tracing, particle systems, and GPU-based simulations.

-
- Build a portfolio of projects that showcase your skills as a Vulkan developer.

How to Use This Book

This book is both a learning guide and a reference:

- **Follow the Structure:** Start with the basics and progress through the projects sequentially. Each project builds on the previous ones, helping you develop a strong foundation.
- **Reference Individual Projects:** Each chapter is self-contained, so you can jump to specific topics when you need them.
- **Experiment and Expand:** Use the projects as a starting point for your own experiments. Modify and expand the examples to suit your needs.

Prerequisites

To get the most out of this book, you should have:

- Basic programming knowledge in C or C++.
- Familiarity with basic graphics concepts (e.g., vertices, shaders, textures).
- A system capable of running Vulkan applications (a modern GPU and up-to-date drivers).
- Vulkan SDK installed on your system (<https://vulkan.lunarg.com/sdk/home>).

Let's Begin

Learning Vulkan is an exciting and rewarding journey. At the end of this book, you'll have not only mastered the API but also built a collection of projects that demonstrate your skills and creativity. Let's dive into the world of Vulkan and start creating something amazing!

Part I

Basic Vulkan Setup



1

Set Up Vulkan Instance: Create and manage a Vulkan instance

1.1 Introduction

Before you can begin rendering or performing compute tasks with Vulkan, you must establish a Vulkan instance. This instance serves as the foundation of any Vulkan application, representing the connection between your application and the Vulkan library. It allows you to interact with the Vulkan API, query available devices, and manage system-level resources. Setting up a Vulkan instance is the first critical step in any Vulkan program.

In this chapter, we will walk through the process of creating a Vulkan instance from scratch. You will learn how to specify extensions and enable validation layers, which are essential for debugging and ensuring proper API usage. At the end of this chapter, you'll have a working Vulkan instance, ready to serve as the basis for more complex tasks.

1.2 Understanding the Vulkan Instance

The Vulkan instance acts as a gateway between your application and the Vulkan runtime. It allows you to initialize Vulkan functionality, query available physical devices, and enable features like debugging tools or platform-specific extensions. While the Vulkan instance itself doesn't perform rendering or computation, it sets up the necessary groundwork for interacting with the GPU.

A Vulkan instance is created using the `vkCreateInstance` function, which takes detailed configuration data to specify how the instance should behave. This includes information about your application, extensions to enable, and optional debugging tools like validation

layers. Understanding how to correctly configure the Vulkan instance is crucial for any Vulkan developer.

1.3 Setting Up the Vulkan Instance

To create a Vulkan instance, we need to follow a series of steps that involve configuring application information, enabling extensions, and specifying validation layers for debugging. Let's dive into the process.

1.3.1 Including Vulkan Headers

The first step is to include the Vulkan header file in your project. This file provides access to all Vulkan functions and structures. You'll also want to include standard libraries for debugging and handling errors.

```
1 #include <vulkan/vulkan.h>
2 #include <iostream>
3 #include <vector>
```

1.3.2 Defining Extensions and Validation Layers

Extensions and validation layers are a critical part of Vulkan's flexibility and robustness. Extensions allow you to enable platform-specific features, such as window system integration or debugging utilities. Validation layers help catch errors and ensure proper API usage during development, making debugging significantly easier.

Here's how you can define the validation layers and extensions for your Vulkan instance:

```
1 const std::vector<const char*> validationLayers = {
2     "VK_LAYER_KHRONOS_validation"
3 };
4
5 std::vector<const char*> getRequiredExtensions() {
6     std::vector<const char*> extensions;
7     extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
8     return extensions;
9 }
```

In this example, we enable the `VK_LAYER_KHRONOS_validation` layer, which performs extensive error checking. We also specify the `VK_EXT_DEBUG_UTILS_EXTENSION_NAME` extension to enable debugging utilities.

1.3.3 Creating the Vulkan Instance

With the extensions and validation layers defined, we can now create the Vulkan instance using the `vkCreateInstance` function. This function requires a `VkInstanceCreateInfo` structure, which contains all the necessary configuration data.

Here's the code to create a Vulkan instance:

```

1  VkInstance createVulkanInstance() {
2      VkInstance instance;
3
4      // Fill in application info
5      VkApplicationInfo appInfo{};
6      appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
7      appInfo.pApplicationName = "Vulkan App";
8      appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
9      appInfo.pEngineName = "No Engine";
10     appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
11     appInfo.apiVersion = VK_API_VERSION_1_3;
12
13     // Fill in instance creation info
14     VkInstanceCreateInfo createInfo{};
15     createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
16     createInfo.pApplicationInfo = &appInfo;
17
18     // Add extensions
19     auto extensions = getRequiredExtensions();
20     createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
21     createInfo.ppEnabledExtensionNames = extensions.data();
22
23     // Add validation layers
24     createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
25     createInfo.ppEnabledLayerNames = validationLayers.data();
26
27     // Create the Vulkan instance
28     if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
29         throw std::runtime_error("Failed to create Vulkan instance");
30     }
31
32     return instance;
33 }
```

This code includes an optional `VkApplicationInfo` structure that provides metadata about your application, such as its name and version. While not strictly required, including this information is recommended for clarity and compatibility.

1.3.4 Cleaning Up the Vulkan Instance

After creating a Vulkan instance, you must also manage its lifecycle. When your application finishes using Vulkan, you should destroy the instance to free up system resources. This is done using the `vkDestroyInstance` function:

```

1  void cleanup(VkInstance instance) {
2      vkDestroyInstance(instance, nullptr);
3  }
```

Failing to clean up resources properly can lead to memory leaks and other issues, especially in long-running applications.

1.3.5 Complete Example Program

Here's the complete example that sets up, uses, and cleans up a Vulkan instance:

```

1  #include <vulkan/vulkan.h>
2  #include <iostream>
3  #include <vector>
4
5  const std::vector<const char*> validationLayers = {
6      "VK_LAYER_KHRONOS_validation"
7  };
8
9  std::vector<const char*> getRequiredExtensions() {
10     std::vector<const char*> extensions;
11     extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);
12     return extensions;
13 }
14
15 VkInstance createVulkanInstance() {
16     VkInstance instance;
17
18     VkApplicationInfo appInfo{};
19     appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
20     appInfo.pApplicationName = "Vulkan App";
21     appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
22     appInfo.pEngineName = "No Engine";
23     appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
24     appInfo.apiVersion = VK_API_VERSION_1_3;
25
26     VkInstanceCreateInfo createInfo{};
27     createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
28     createInfo.pApplicationInfo = &appInfo;
29
30     auto extensions = getRequiredExtensions();
31     createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.size());
32     createInfo.ppEnabledExtensionNames = extensions.data();
33
34     createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
35     createInfo.ppEnabledLayerNames = validationLayers.data();
36
37     if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
38         throw std::runtime_error("Failed to create Vulkan instance");
39     }
40
41     return instance;
42 }
43
44 void cleanup(VkInstance instance) {
45     vkDestroyInstance(instance, nullptr);
46 }
47
48 int main() {
49     VkInstance instance;
50
51     try {
52         instance = createVulkanInstance();
53         std::cout << "Vulkan Instance created successfully!" << std::endl;
54     } catch (const std::exception& e) {
55         std::cerr << e.what() << std::endl;
56         return EXIT_FAILURE;
57     }
58
59     cleanup(instance);
60     return EXIT_SUCCESS;
61 }

```

1.4 Summary

In this chapter, you've created a Vulkan instance, the first essential step in any Vulkan application. You've learned how to configure the instance with validation layers and extensions, and you've implemented error handling to ensure the instance is created successfully. Managing the instance's lifecycle properly, you've set the stage for more advanced Vulkan programming.

With the instance ready, you're now prepared to move on to selecting a physical device (GPU) and exploring its properties, which we'll cover in the next chapter.

DRAFT
version
0.681

2

Enumerate Physical Devices: List available GPUs on your system

2.1 Introduction

With a Vulkan instance successfully created, the next step is to enumerate the physical devices (GPUs) available on your system. Vulkan supports multiple GPUs, and each physical device represents a hardware GPU that your application can use. Enumerating these devices allows you to identify their properties and capabilities, which will help you choose the most suitable GPU for your application.

In this chapter, we will explore how to list the physical devices on your system, query their properties, and print relevant information. This will provide insight into the hardware capabilities and serve as a foundation for selecting the optimal GPU in the next steps.

2.2 Understanding Physical Devices

A `VkPhysicalDevice` represents a physical GPU or other Vulkan-compatible hardware. Once enumerated, physical devices can be queried for details such as their name, type, supported features, and limits. Vulkan's explicit nature requires the developer to select the desired device explicitly, making this enumeration process a critical part of Vulkan application development.

The function `vkEnumeratePhysicalDevices` is used to retrieve the list of physical devices associated with a Vulkan instance. Each device can then be queried individually to obtain specific properties.

2.3 Enumerating Physical Devices

To list the available GPUs, we use `vkEnumeratePhysicalDevices`. This function first retrieves the count of physical devices and then fills an array with the handles for each device. Let's break the process into steps.

2.3.1 Step 1: Retrieve the Number of Physical Devices

The first call to `vkEnumeratePhysicalDevices` retrieves the count of available devices. You need to pass a pointer to an unsigned integer to store this count:

```
1 // Get the number of physical devices
2 uint32_t deviceCount = 0;
3 vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
4
5 if (deviceCount == 0) {
6     throw std::runtime_error("No Vulkan-compatible GPUs found.");
7 }
```

This snippet ensures that at least one physical device is available. If `deviceCount` is zero, Vulkan cannot run on the system.

2.3.2 Step 2: Retrieve the Device Handles

The second call to `vkEnumeratePhysicalDevices` fills an array with the handles to each available physical device. These handles can then be used to query the properties of each device:

```
1 // Retrieve the physical device handles
2 std::vector<VkPhysicalDevice> physicalDevices(deviceCount);
3 vkEnumeratePhysicalDevices(instance, &deviceCount, physicalDevices.data());
```

Now that you have the device handles, you can inspect each device to determine its properties and capabilities.

2.3.3 Step 3: Query Device Properties

Once you have a list of physical devices, you can query their properties using `vkGetPhysicalDeviceProperties`. This function provides details such as the device name, type, and Vulkan API version supported:

```
1 // Query and print properties of each physical device
2 for (const auto& device : physicalDevices) {
3     VkPhysicalDeviceProperties deviceProperties;
4     vkGetPhysicalDeviceProperties(device, &deviceProperties);
5
6     std::cout << "Device Name: " << deviceProperties.deviceName << "\n";
7     std::cout << "Device Type: " << deviceProperties.deviceType << "\n";
8     std::cout << "API Version: "
9         << VK_VERSION_MAJOR(deviceProperties.apiVersion) << "."
10        << VK_VERSION_MINOR(deviceProperties.apiVersion) << "."
11        << VK_VERSION_PATCH(deviceProperties.apiVersion) << "\n";
12 }
```

The `VkPhysicalDeviceProperties` structure contains various fields, including `deviceName` (a human-readable name for the GPU), `deviceType` (indicating whether the device is integrated or discrete), and `apiVersion` (the highest Vulkan version supported).

2.3.4 Step 4: Query Device Features and Extensions

In addition to basic properties, you can also query the features and extensions supported by each device. For example, you can check whether the device supports specific Vulkan features like geometry shaders or compute shaders:

```
1 // Query supported features
2 VkPhysicalDeviceFeatures deviceFeatures;
3 vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
4
5 std::cout << "Geometry Shader Support: "
6             << (deviceFeatures.geometryShader ? "Yes" : "No") << "\n";
```

This information is crucial for determining whether a device meets the requirements of your application.

2.4 Complete Example Program

The following example demonstrates the complete process of enumerating physical devices and printing their properties:

```
1 #include <vulkan/vulkan.h>
2 #include <iostream>
3 #include <vector>
4
5 void listPhysicalDevices(VkInstance instance) {
6     uint32_t deviceCount = 0;
7     vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
8
9     if (deviceCount == 0) {
10         throw std::runtime_error("No Vulkan-compatible GPUs found.");
11     }
12
13     std::vector<VkPhysicalDevice> physicalDevices(deviceCount);
14     vkEnumeratePhysicalDevices(instance, &deviceCount, physicalDevices.data());
15
16     for (const auto& device : physicalDevices) {
17         VkPhysicalDeviceProperties deviceProperties;
18         vkGetPhysicalDeviceProperties(device, &deviceProperties);
19
20         std::cout << "Device Name: " << deviceProperties.deviceName << "\n";
21         std::cout << "Device Type: " << deviceProperties.deviceType << "\n";
22         std::cout << "API Version: "
23                 << VK_VERSION_MAJOR(deviceProperties.apiVersion) << "."
24                 << VK_VERSION_MINOR(deviceProperties.apiVersion) << "."
25                 << VK_VERSION_PATCH(deviceProperties.apiVersion) << "\n";
26
27         VkPhysicalDeviceFeatures deviceFeatures;
28         vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
29
30         std::cout << "Geometry Shader Support: "
31                 << (deviceFeatures.geometryShader ? "Yes" : "No") << "\n\n";
```

```
32     }
33 }
34
35 int main() {
36     VkInstance instance;
37
38     try {
39         // Assume createVulkanInstance() is implemented from the previous chapter
40         instance = createVulkanInstance();
41         listPhysicalDevices(instance);
42     } catch (const std::exception& e) {
43         std::cerr << e.what() << std::endl;
44         return EXIT_FAILURE;
45     }
46
47     vkDestroyInstance(instance, nullptr);
48     return EXIT_SUCCESS;
49 }
```

2.5 Summary

In this chapter, you learned how to enumerate physical devices on your system and retrieve their properties and features. This process provides insight into the available GPUs and prepares you for selecting a suitable device in the next step. The ability to query device capabilities is an essential part of Vulkan development, ensuring your application can adapt to various hardware configurations.

3

Enable Validation Layers: Debug Vulkan code with validation layers

3.1 Introduction

One of Vulkan's most powerful debugging features is its validation layers. Validation layers provide runtime checks to help you catch errors, warn about suboptimal practices, and understand potential pitfalls in your Vulkan code. Enabling these layers, you gain valuable insights into your application's behavior, making the debugging process more efficient.

Why Use Validation Layers? Vulkan's low-level nature gives you full control but also increases the likelihood of mistakes. Validation layers act as a safeguard, offering detailed feedback whenever something goes wrong or deviates from best practices. They are an indispensable tool, especially when starting out with Vulkan development.

In this chapter, we will walk through the steps to enable validation layers, configure them for optimal debugging, and interpret their outputs.

3.2 How Validation Layers Work

Validation layers intercept Vulkan API calls and check them against a set of rules. They monitor:

- Resource management, such as memory allocation and deallocation.
- Proper usage of Vulkan commands and parameters.
- Synchronization issues.
- Compatibility between pipeline stages and resources.

When an issue is detected, the validation layers generate detailed messages, including errors, warnings, and performance hints. These messages are sent to a debug callback function, which you can customize.

3.3 Step-by-Step Implementation

3.3.1 Step 1: Check Validation Layer Support

Before enabling validation layers, ensure that the desired layers are available on the system.

```

1  // List the desired validation layers
2  const std::vector<const char*> validationLayers = {
3      "VK_LAYER_KHRONOS_validation"
4  };
5
6  // Check if validation layers are supported
7  bool checkValidationLayerSupport() {
8      uint32_t layerCount;
9      vkEnumerateInstanceLayerProperties(&layerCount, nullptr);
10
11      std::vector<VkLayerProperties> availableLayers(layerCount);
12      vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data());
13
14      for (const char* layerName : validationLayers) {
15          bool layerFound = false;
16
17          for (const auto& layerProperties : availableLayers) {
18              if (strcmp(layerName, layerProperties.layerName) == 0) {
19                  layerFound = true;
20                  break;
21              }
22          }
23
24          if (!layerFound) {
25              return false;
26          }
27      }
28
29      return true;
30  }

```

If the function returns `false`, the validation layers are not available on the system. Ensure that the Vulkan SDK is installed, as it includes the necessary layers.

3.3.2 Step 2: Enable Validation Layers During Instance Creation

To use validation layers, include them in the Vulkan instance creation process.

```

1  // Add validation layers to the instance creation info
2  VkInstanceCreateInfo createInfo{};
3  createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
4
5  VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo{};
6  if (enableValidationLayers) {
7      createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
8      createInfo.ppEnabledLayerNames = validationLayers.data();

```



```

9
10 populateDebugMessengerCreateInfo(debugCreateInfo);
11 createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*) &debugCreateInfo;
12 } else {
13     createInfo.enabledLayerCount = 0;
14     createInfo.pNext = nullptr;
15 }

```

3.3.3 Step 3: Set Up a Debug Callback Function

A debug callback function captures and processes messages generated by the validation layers.

```

1 // Define the debug callback
2 VKAPI_ATTR VKBool32 VKAPI_CALL debugCallback(
3     VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
4     VkDebugUtilsMessageTypeFlagsEXT messageType,
5     const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
6     void* pUserData) {
7
8     std::cerr << "Validation Layer: " << pCallbackData->pMessage << std::endl;
9     return VK_FALSE;
10 }

```

This function logs messages to the standard error stream (`std::cerr`). You can customize it to integrate with logging systems or break on specific errors.

3.3.4 Step 4: Create a Debug Messenger Object

The debug messenger routes messages from the validation layers to your callback.

```

1 // Populate debug messenger creation info
2 void populateDebugMessengerCreateInfo(VkDebugUtilsMessengerCreateInfoEXT& createInfo) {
3     createInfo = 0;
4     createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
5     createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT |
6                                 VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |
7                                 VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
8     createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT |
9                             VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |
10                             VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
11     createInfo.pfnUserCallback = debugCallback;
12     createInfo.pUserData = nullptr; // Optional
13 }
14
15 // Create the debug messenger
16 VkDebugUtilsMessengerEXT debugMessenger;
17
18 if (enableValidationLayers) {
19     if (CreateDebugUtilsMessengerEXT(instance, &debugCreateInfo, nullptr, &debugMessenger) != VK_SUCCESS) {
20         throw std::runtime_error("Failed to set up debug messenger!");
21     }
22 }

```

Make sure to destroy the debug messenger during cleanup.

```

1 if (enableValidationLayers) {
2     DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);

```

3 }

3.4 Using Validation Layer Messages

Once validation layers are enabled, you'll see messages in the console when issues arise. These messages include:

- **Severity:** Indicates the importance (verbose, warning, error).
- **Type:** General, validation, or performance.
- **Description:** A detailed explanation of the issue.

For example:

```
1 Validation Layer: [Error] vkQueueSubmit: The submitted command buffer uses an uninitialized image.
```

Warning: Address warnings and errors promptly. Validation messages often reveal deeper issues that can cause crashes or performance bottlenecks.

3.5 Summary

This chapter introduced Vulkan's validation layers, a critical tool for debugging and optimizing Vulkan applications. Enabling these layers and configuring a debug callback, you can gain valuable insights into your application's behavior.

4

Create Logical Device and Queues: Create logical devices and retrieve queue handles

4.1 Introduction

After enumerating physical devices and determining their properties, the next step in using Vulkan is to create a logical device. While a physical device represents the hardware, a logical device provides an interface for interacting with the GPU. It is through this logical device that we access queues, which are essential for submitting rendering or compute commands to the GPU.

In this chapter, we will discuss the process of creating a logical device, specifying the required queues, and retrieving queue handles. The end, you will have a fully configured logical device and the means to begin issuing commands to the GPU.

4.2 Understanding Logical Devices and Queues

A logical device is an abstraction that allows applications to interact with a physical device. Unlike the physical device, which cannot be modified directly, the logical device is configured to enable specific features and extensions. This customization ensures that the application has access to only the features and queues it needs, reducing overhead.

Queues are the primary mechanism for submitting work to the GPU. Each physical device has multiple queue families, where each family supports specific types of operations (e.g., graphics, compute, or transfer). When creating a logical device, you specify the queue families you want to use and the number of queues to create.

4.3 Creating a Logical Device

To create a logical device, you need to provide details about the desired queues, enabled features, and any required device extensions. Here is the step-by-step process.

4.3.1 Step 1: Identify Queue Families

Before creating a logical device, you must identify the queue families available on the physical device. Queue families are queried using `vkGetPhysicalDeviceQueueFamilyProperties`, which returns a list of properties for each queue family.

```
1 // Query queue family properties
2 uint32_t queueFamilyCount = 0;
3 vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, nullptr);
4
5 std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
6 vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, queueFamilies.data());
```

Each `VkQueueFamilyProperties` structure describes the capabilities of a queue family, such as whether it supports graphics, compute, or transfer operations.

4.3.2 Step 2: Select a Suitable Queue Family

For most Vulkan applications, you need a queue family that supports graphics operations. The following code snippet demonstrates how to find a suitable queue family:

```
1 // Find a queue family that supports graphics
2 int graphicsQueueFamilyIndex = -1;
3
4 for (uint32_t i = 0; i < queueFamilies.size(); i++) {
5     if (queueFamilies[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
6         graphicsQueueFamilyIndex = i;
7         break;
8     }
9 }
10
11 if (graphicsQueueFamilyIndex == -1) {
12     throw std::runtime_error("Failed to find a suitable graphics queue family.");
13 }
```

Here, we search for a queue family with the `VK_QUEUE_GRAPHICS_BIT` flag, which indicates support for graphics commands.

4.3.3 Step 3: Specify Queue Creation Information

Once you have identified the desired queue family, you must specify the queues you want to create. Vulkan allows you to create multiple queues from the same family, and each queue can be configured with a priority (a value between 0.0 and 1.0).

```
1 // Specify the queue creation information
2 float queuePriority = 1.0f;
```

```

3
4 VkDeviceQueueCreateInfo queueCreateInfo{};
5 queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
6 queueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex;
7 queueCreateInfo.queueCount = 1;
8 queueCreateInfo.pQueuePriorities = &queuePriority;

```

The `queueFamilyIndex` specifies which queue family to create queues from, while `queueCount` specifies the number of queues to create.

4.3.4 Step 4: Configure the Logical Device

Next, you configure the logical device by specifying the enabled features, extensions, and queue creation information.

```

1 // Specify logical device creation info
2 VkDeviceCreateInfo deviceCreateInfo{};
3 deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
4 deviceCreateInfo.pQueueCreateInfos = &queueCreateInfo;
5 deviceCreateInfo.queueCreateInfoCount = 1;
6
7 VkPhysicalDeviceFeatures deviceFeatures{};
8 deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
9
10 if (vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device) != VK_SUCCESS) {
11     throw std::runtime_error("Failed to create logical device.");
12 }

```

The `vkCreateDevice` function creates the logical device and returns a handle to it. This handle is required for all subsequent interactions with the GPU.

4.3.5 Step 5: Retrieve Queue Handles

Once the logical device is created, you can retrieve handles to the queues using `vkGetDeviceQueue`. These handles are used to submit commands to the GPU.

```

1 // Retrieve the queue handle
2 VkQueue graphicsQueue;
3 vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &graphicsQueue);

```

The `vkGetDeviceQueue` function retrieves a queue handle from the specified queue family. The second parameter, `queueIndex`, allows you to access multiple queues if you created more than one.

4.4 Complete Example Program

Here is a complete example that creates a logical device and retrieves a graphics queue:

```

1 #include <vulkan/vulkan.h>
2 #include <iostream>
3 #include <vector>

```

```

4
5 VkDevice createLogicalDevice(VkPhysicalDevice physicalDevice, int graphicsQueueFamilyIndex) {
6     float queuePriority = 1.0f;
7
8     VkDeviceQueueCreateInfo queueCreateInfo{};
9     queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
10    queueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex;
11    queueCreateInfo.queueCount = 1;
12    queueCreateInfo.pQueuePriorities = &queuePriority;
13
14    VkPhysicalDeviceFeatures deviceFeatures{};
15
16    VkDeviceCreateInfo deviceCreateInfo{};
17    deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
18    deviceCreateInfo.pQueueCreateInfos = &queueCreateInfo;
19    deviceCreateInfo.queueCreateInfoCount = 1;
20    deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
21
22    VkDevice device;
23    if (vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device) != VK_SUCCESS) {
24        throw std::runtime_error("Failed to create logical device.");
25    }
26
27    return device;
28 }
29
30 VkQueue getGraphicsQueue(VkDevice device, int graphicsQueueFamilyIndex) {
31     VkQueue queue;
32     vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &queue);
33     return queue;
34 }
35
36 int main() {
37     // Assume instance and physicalDevice are already created
38     VkInstance instance = createVulkanInstance();
39     VkPhysicalDevice physicalDevice = pickPhysicalDevice(instance); // Implement as needed
40
41     int graphicsQueueFamilyIndex = findGraphicsQueueFamily(physicalDevice); // Implement as needed
42
43     VkDevice device = createLogicalDevice(physicalDevice, graphicsQueueFamilyIndex);
44     VkQueue graphicsQueue = getGraphicsQueue(device, graphicsQueueFamilyIndex);
45
46     std::cout << "Logical device and queue created successfully." << std::endl;
47
48     vkDestroyDevice(device, nullptr);
49     vkDestroyInstance(instance, nullptr);
50
51     return 0;
52 }

```

4.5 Summary

In this chapter, you learned how to create a logical device and retrieve queue handles. These steps are essential for interacting with the GPU, enabling you to submit rendering or compute commands. The logical device serves as the primary interface to the hardware, while queues provide the means to execute workloads efficiently.

In the next chapter, we will discuss how to set up a Vulkan surface, which is the next step toward rendering to a window.

5

Present a Blank Swapchain Image: Present an empty window via Vulkan

5.1 Introduction

Before rendering complex graphics, it is essential to understand how to present a blank swapchain image. This is the simplest operation in Vulkan's rendering workflow: creating an empty window and presenting a clear image to it. This chapter walks you through the process of creating and presenting a blank swapchain image.

Why Present a Blank Swapchain Image? This operation forms the foundation of the Vulkan rendering loop. It verifies that your Vulkan setup—instance, device, swapchain, and presentation—is working correctly. It also establishes the workflow for submitting work to the GPU and presenting the results to the screen.

5.2 Understanding Swapchain Presentation

The swapchain is a collection of images that are displayed sequentially on the screen. Presenting a blank image involves acquiring an image from the swapchain, submitting it to the queue, and presenting it using the Vulkan presentation interface. This process consists of three main stages:

1. **Acquire an Image:** Retrieve the next available image from the swapchain.
2. **Submit Commands:** Use a command buffer to transition the image for presentation.
3. **Present the Image:** Queue the image for display on the screen.

5.3 Step-by-Step Implementation

5.3.1 Step 1: Acquire a Swapchain Image

To acquire an image from the swapchain, use the `vkAcquireNextImageKHR` function. This retrieves the index of the next available image for rendering.

```

1 // Acquire the next image from the swapchain
2 uint32_t imageIndex;
3 VkResult result = vkAcquireNextImageKHR(
4     device,
5     swapchain,
6     UINT64_MAX, // Timeout in nanoseconds
7     imageAvailableSemaphore, // Semaphore to signal availability
8     VK_NULL_HANDLE, // Optional fence
9     &imageIndex
10 );
11
12 if (result != VK_SUCCESS) {
13     throw std::runtime_error("Failed to acquire swapchain image!");
14 }
```

Here, `imageAvailableSemaphore` ensures synchronization, signaling that the image is ready for rendering.

5.3.2 Step 2: Record a Command Buffer

To prepare the blank image for presentation, record a command buffer that transitions the swapchain image layout to `VK_IMAGE_LAYOUT_PRESENT_SRC_KHR`. This layout is required for presenting images to the screen.

```

1 // Record command buffer to transition image layout
2 VkCommandBuffer commandBuffer = beginCommandBuffer();
3
4 VkImageMemoryBarrier barrier{};
5 barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
6 barrier.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
7 barrier.newLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
8 barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
9 barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
10 barrier.image = swapchainImages[imageIndex];
11 barrier.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
12 barrier.subresourceRange.baseMipLevel = 0;
13 barrier.subresourceRange.levelCount = 1;
14 barrier.subresourceRange.baseArrayLayer = 0;
15 barrier.subresourceRange.layerCount = 1;
16
17 vkCmdPipelineBarrier(
18     commandBuffer,
19     VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, // Source stage
20     VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, // Destination stage
21     0, // Dependency flags
22     0, nullptr, // Memory barriers
23     0, nullptr, // Buffer memory barriers
24     1, &barrier // Image memory barrier
25 );
26
27 endCommandBuffer(commandBuffer);
```

This step ensures the swapchain image is in the correct state for presentation.

5.3.3 Step 3: Submit the Command Buffer

Submit the recorded command buffer to the graphics queue. This schedules the work required to prepare the image for presentation.

```

1  // Submit the command buffer
2  VkSubmitInfo submitInfo{};
3  submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
4
5  VkSemaphore waitSemaphores[] = {imageAvailableSemaphore};
6  VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT};
7  submitInfo.waitSemaphoreCount = 1;
8  submitInfo.pWaitSemaphores = waitSemaphores;
9  submitInfo.pWaitDstStageMask = waitStages;
10
11  submitInfo.commandBufferCount = 1;
12  submitInfo.pCommandBuffers = &commandBuffer;
13
14  VkSemaphore signalSemaphores[] = {renderFinishedSemaphore};
15  submitInfo.signalSemaphoreCount = 1;
16  submitInfo.pSignalSemaphores = signalSemaphores;
17
18  if (vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE) != VK_SUCCESS) {
19      throw std::runtime_error("Failed to submit draw command buffer!");
20  }

```

The `imageAvailableSemaphore` ensures synchronization between image acquisition and rendering, while `renderFinishedSemaphore` signals when the rendering is complete.

5.3.4 Step 4: Present the Image

Finally, present the image to the screen using `vkQueuePresentKHR`.

```

1  // Present the image
2  VkPresentInfoKHR presentInfo{};
3  presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
4
5  VkSemaphore waitSemaphores[] = {renderFinishedSemaphore};
6  presentInfo.waitSemaphoreCount = 1;
7  presentInfo.pWaitSemaphores = waitSemaphores;
8
9  VkSwapchainKHR swapchains[] = {swapchain};
10  presentInfo.swapchainCount = 1;
11  presentInfo.pSwapchains = swapchains;
12  presentInfo.pImageIndices = &imageIndex;
13
14  VkResult presentResult = vkQueuePresentKHR(presentQueue, &presentInfo);
15  if (presentResult != VK_SUCCESS) {
16      throw std::runtime_error("Failed to present swapchain image!");
17  }

```

This function queues the image for presentation. The synchronization semaphore ensures that the image is presented only after the rendering commands are complete.

5.4 Complete Example Program

Here is a high-level summary of the workflow for presenting a blank swapchain image:

1. Acquire an image from the swapchain using `vkAcquireNextImageKHR`.
2. Record a command buffer to transition the image layout.
3. Submit the command buffer to the graphics queue.
4. Present the image using `vkQueuePresentKHR`.

5.5 Rendering Output

Once completed, the program will display a blank window filled with the default swapchain clear color (usually black or another specified color). No geometry or textures are rendered at this stage, but the setup verifies that the swapchain presentation is working correctly.

5.6 Summary

In this chapter, you learned how to present a blank swapchain image using Vulkan. This foundational workflow prepares you for rendering actual geometry and textures. Mastering image acquisition, command buffer submission, and presentation, you have taken a crucial step toward building a Vulkan rendering engine.

6

Set Up a Vulkan Pipeline: Understand and implement the Vulkan graphics pipeline

6.1 Introduction

The Vulkan graphics pipeline is the backbone of rendering. It is a series of fixed-function and programmable stages that process input data into pixels on the screen. Unlike other graphics APIs, Vulkan requires developers to explicitly configure the pipeline, offering unparalleled flexibility and control. In this chapter, we will explore how to set up a Vulkan graphics pipeline, covering its key stages and implementation.

Why Set Up a Graphics Pipeline? The graphics pipeline is essential for rendering objects to the screen. Configuring the pipeline, you define how vertices are processed, how shaders operate, and how fragments are displayed. A well-designed pipeline is the cornerstone of efficient Vulkan applications.

6.2 Understanding the Graphics Pipeline Stages

The Vulkan graphics pipeline consists of several stages, each with a specific role:

1. **Input Assembly:** Gathers vertex data and assembles primitives (e.g., triangles).
2. **Vertex Shader:** Processes individual vertices, applying transformations and other operations.
3. **Tessellation and Geometry Shaders (Optional):** Modify geometry or add detail.
4. **Rasterization:** Converts primitives into fragments.
5. **Fragment Shader:** Processes fragments to determine their final color and depth.
6. **Color Blending:** Combines fragment outputs with existing framebuffer content.

6.3 Step-by-Step Implementation

6.3.1 Step 1: Define Shader Stages

Shaders are small programs that run on the GPU. They are compiled into SPIR-V format and linked into the graphics pipeline.

```

1 // Load SPIR-V shader code
2 VkShaderModule vertShaderModule = createShaderModule("vert.spv");
3 VkShaderModule fragShaderModule = createShaderModule("frag.spv");
4
5 // Define shader stages
6 VkPipelineShaderStageCreateInfo vertShaderStageInfo{};
7 vertShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
8 vertShaderStageInfo.stage = VK_SHADER_STAGE_VERTEX_BIT;
9 vertShaderStageInfo.module = vertShaderModule;
10 vertShaderStageInfo.pName = "main";
11
12 VkPipelineShaderStageCreateInfo fragShaderStageInfo{};
13 fragShaderStageInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
14 fragShaderStageInfo.stage = VK_SHADER_STAGE_FRAGMENT_BIT;
15 fragShaderStageInfo.module = fragShaderModule;
16 fragShaderStageInfo.pName = "main";
17
18 VkPipelineShaderStageCreateInfo shaderStages[] = {vertShaderStageInfo, fragShaderStageInfo};

```

Each `VkPipelineShaderStageCreateInfo` structure specifies a shader module and its entry point (`"main"`).

6.3.2 Step 2: Configure Fixed-Function Stages

Fixed-function stages, like the input assembly and rasterizer, require explicit configuration.

Input Assembly Stage:

```

1 // Configure input assembly
2 VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
3 inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
4 inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST; // Triangles
5 inputAssembly.primitiveRestartEnable = VK_FALSE; // No primitive restart

```

Viewport and Scissor:

```

1 // Configure viewport and scissor
2 VkViewport viewport{};
3 viewport.x = 0.0f;
4 viewport.y = 0.0f;
5 viewport.width = (float) swapchainExtent.width;
6 viewport.height = (float) swapchainExtent.height;
7 viewport.minDepth = 0.0f;
8 viewport.maxDepth = 1.0f;
9
10 VkRect2D scissor{};
11 scissor.offset = {0, 0};
12 scissor.extent = swapchainExtent;
13
14 VkPipelineViewportStateCreateInfo viewportState{};

```

```

15 viewportState.sType = VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
16 viewportState.viewportCount = 1;
17 viewportState.pViewports = &viewport;
18 viewportState.scissorCount = 1;
19 viewportState.pScissors = &scissor;

```

Rasterizer:

```

1 // Configure rasterizer
2 VkPipelineRasterizationStateCreateInfo rasterizer{};
3 rasterizer.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
4 rasterizer.depthClampEnable = VK_FALSE;
5 rasterizer.rasterizerDiscardEnable = VK_FALSE;
6 rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
7 rasterizer.lineWidth = 1.0f;
8 rasterizer.cullMode = VK_CULL_MODE_BACK_BIT;
9 rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;
10 rasterizer.depthBiasEnable = VK_FALSE;

```

Color Blending:

```

1 // Configure color blending
2 VkPipelineColorBlendAttachmentState colorBlendAttachment{};
3 colorBlendAttachment.colorWriteMask = VK_COLOR_COMPONENT_R_BIT |
4                                     VK_COLOR_COMPONENT_G_BIT |
5                                     VK_COLOR_COMPONENT_B_BIT |
6                                     VK_COLOR_COMPONENT_A_BIT;
7 colorBlendAttachment.blendEnable = VK_FALSE;
8
9 VkPipelineColorBlendStateCreateInfo colorBlending{};
10 colorBlending.sType = VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
11 colorBlending.logicOpEnable = VK_FALSE;
12 colorBlending.logicOp = VK_LOGIC_OP_COPY;
13 colorBlending.attachmentCount = 1;
14 colorBlending.pAttachments = &colorBlendAttachment;
15 colorBlending.blendConstants[0] = 0.0f;
16 colorBlending.blendConstants[1] = 0.0f;
17 colorBlending.blendConstants[2] = 0.0f;
18 colorBlending.blendConstants[3] = 0.0f;

```

6.3.3 Step 3: Create the Pipeline Layout

The pipeline layout defines how shader resources (e.g., uniforms) are organized. For this example, we use an empty layout.

```

1 // Create an empty pipeline layout
2 VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
3 pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
4
5 VkPipelineLayout pipelineLayout;
6 if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {
7     throw std::runtime_error("Failed to create pipeline layout!");
8 }

```

6.3.4 Step 4: Create the Graphics Pipeline

Finally, combine all the configurations into a graphics pipeline.

```

1  // Combine all pipeline configurations
2  VkGraphicsPipelineCreateInfo pipelineInfo{};
3  pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
4  pipelineInfo.stageCount = 2;
5  pipelineInfo.pStages = shaderStages;
6  pipelineInfo.pVertexInputState = &vertexInputInfo;
7  pipelineInfo.pInputAssemblyState = &inputAssembly;
8  pipelineInfo.pViewportState = &viewportState;
9  pipelineInfo.pRasterizationState = &rasterizer;
10 pipelineInfo.pMultisampleState = &multisampling;
11 pipelineInfo.pColorBlendState = &colorBlending;
12 pipelineInfo.layout = pipelineLayout;
13 pipelineInfo.renderPass = renderPass;
14 pipelineInfo.subpass = 0;
15
16 VkPipeline graphicsPipeline;
17 if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &graphicsPipeline) !=
    ↪ VK_SUCCESS) {
18     throw std::runtime_error("Failed to create graphics pipeline!");
19 }

```

6.4 Complete Example

Setting up a Vulkan pipeline involves:

1. Defining shader stages.
2. Configuring fixed-function stages.
3. Creating a pipeline layout.
4. Combining these into a graphics pipeline.

6.5 Summary

This chapter introduced the Vulkan graphics pipeline, detailing its stages and configuration. Understanding the pipeline is crucial for rendering in Vulkan. In the next chapter, we will extend this knowledge by rendering a textured triangle, exploring descriptor sets and samplers.

7

Clear the Screen with a Color: Implement screen-clearing using Vulkan commands

7.1 Introduction

Before drawing any objects, it is common practice to clear the screen to a specific color. This ensures that any leftover data from the previous frame does not interfere with the current frame's rendering. In Vulkan, clearing the screen is performed as part of the render pass using **clear values** and Vulkan's subpass operations.

Why Clear the Screen? Clearing the screen is essential for initializing the framebuffer with a clean state. It provides a consistent background for rendering and helps avoid visual artifacts. This chapter guides you through configuring a Vulkan render pass to clear the screen with a solid color.

7.2 Understanding Clear Values

In Vulkan, a render pass specifies how attachments (e.g., color and depth buffers) are processed during rendering. Each attachment can be assigned a **clear value**, which is used to initialize it at the start of the render pass. For example, a color attachment can be cleared to a specific RGBA color, while a depth attachment can be cleared to a specific depth value.

7.3 Step-by-Step Implementation

7.3.1 Step 1: Define the Clear Color

The clear color is defined as a `VkClearColor` structure. For a color attachment, this structure specifies the RGBA values to use when clearing.

```
1 // Define the clear color
2 VkClearColor clearColor = {0};
3 clearColor.color = {0.2f, 0.3f, 0.4f, 1.0f}; // Dark blue
```

Here, the clear color is set to a dark blue. You can change these values to customize the background color.

7.3.2 Step 2: Update the Render Pass Configuration

The render pass defines how attachments are cleared. During its creation, configure the `VkAttachmentDescription` structure to specify the clear operation.

```
1 // Configure the render pass attachment
2 VkAttachmentDescription colorAttachment = {0};
3 colorAttachment.format = swapchainImageFormat;
4 colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
5 colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR; // Clear at the start
6 colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE; // Store rendered data
7 colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
8 colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
```

Setting `VK_ATTACHMENT_LOAD_OP_CLEAR` ensures that the attachment is cleared to the specified color at the start of each render pass.

7.3.3 Step 3: Specify Clear Values in the Render Pass Begin Info

When starting a render pass, the clear values are passed through the `VkRenderPassBeginInfo` structure.

```
1 // Set up render pass begin info
2 VkRenderPassBeginInfo renderPassInfo = {0};
3 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
4 renderPassInfo.renderPass = renderPass;
5 renderPassInfo.framebuffer = framebuffer;
6 renderPassInfo.renderArea.offset = {0, 0};
7 renderPassInfo.renderArea.extent = swapchainExtent;
8
9 // Specify clear values
10 VkClearColor clearValues[1] = {clearColor};
11 renderPassInfo.clearValueCount = 1;
12 renderPassInfo.pClearValues = clearValues;
```

Here, the clear values are linked to the render pass, ensuring the color attachment is cleared at the start.

7.3.4 Step 4: Record and Execute the Command Buffer

The command buffer records the commands to begin and end the render pass. During the render pass, the attachment is automatically cleared to the specified color.

```
1 // Record commands to clear the screen
2 vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
3
4 // No drawing commands needed for clearing
5 vkCmdEndRenderPass(commandBuffer);
```

Submitting this command buffer to the graphics queue will clear the screen to the specified color.

7.4 Complete Example Program

To summarize, the steps to clear the screen are as follows:

1. Define the clear color using a `VkClearColor`.
2. Configure the render pass to perform the clear operation using `VK_ATTACHMENT_LOAD_OP_CLEAR`.
3. Specify the clear value in the render pass begin info.
4. Record and submit the command buffer with the render pass.

The following code snippet integrates these steps:

```
1 // Example of screen clearing
2 VkClearColorValue clearColor = {0};
3 clearColor.color = {0.2f, 0.3f, 0.4f, 1.0f};
4
5 VkRenderPassBeginInfo renderPassInfo = {0};
6 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
7 renderPassInfo.renderPass = renderPass;
8 renderPassInfo.framebuffer = framebuffer;
9 renderPassInfo.renderArea.offset = {0, 0};
10 renderPassInfo.renderArea.extent = swapchainExtent;
11 renderPassInfo.clearValueCount = 1;
12 renderPassInfo.pClearValues = &clearColor;
13
14 vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
15 vkCmdEndRenderPass(commandBuffer);
16
17 vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
18 vkQueuePresentKHR(presentQueue, &presentInfo);
```

7.5 Rendering Output

After implementing the steps, the output will display a screen cleared to the specified color. This is a significant milestone, as it forms the basis for more complex rendering operations in future chapters.

7.6 Summary

This chapter covered how to clear the screen to a specific color using Vulkan. You learned how to configure the render pass, define clear values, and execute the clearing operation efficiently. These steps ensure a clean and consistent rendering environment, paving the way for further graphics development.

DRAFT
version
0.681

8

Draw a Single Triangle (Solid Color): Render a triangle with a single color

8.1 Introduction

Having set up a logical device and retrieved queue handles, we are ready to render our first triangle using Vulkan. This chapter covers the steps required to render a single triangle with a solid color, introducing you to Vulkan's rendering pipeline, shaders, and basic drawing commands.

Rendering a triangle is often the first milestone in graphics programming. It encapsulates many fundamental Vulkan concepts, such as command buffers, pipelines, and render passes, while keeping the implementation manageable. The end of this chapter, you will have a triangle displayed on the screen.

8.2 The Rendering Pipeline Overview

Vulkan's rendering pipeline is a sequence of stages that process data to produce an image. For this example, we focus on a simplified pipeline consisting of:

1. **Vertex Input**: Supplies vertex data to the pipeline.
2. **Vertex Shader**: Processes vertices.
3. **Rasterization**: Converts vertices into fragments.
4. **Fragment Shader**: Colors fragments.
5. **Output**: Writes the final image to a framebuffer.

Each stage is highly configurable, allowing you to customize the pipeline to meet your needs.

8.3 Step-by-Step Implementation

8.3.1 Step 1: Prepare a Swapchain and Render Pass

Before rendering, you need to create a swapchain to manage the images to be displayed on the screen and a render pass to define how those images are rendered.

- **Swapchain:** A collection of images used for rendering and presenting to the screen.
- **Render Pass:** Defines the operations to be performed on the images, such as clearing and writing.

If you haven't set up a swapchain and render pass, refer to Vulkan documentation or future chapters in this book for detailed instructions.

8.3.2 Step 2: Define the Triangle's Vertex Data

A triangle is defined by three vertices. Each vertex has a position in normalized device coordinates (ranging from -1 to 1). In this example, the triangle is centered on the screen.

```

1 // Define vertex data
2 struct Vertex {
3     float position[2];
4 };
5
6 const std::vector<Vertex> vertices = {
7     {{ 0.0f, -0.5f}}, // Bottom
8     {{-0.5f,  0.5f}}, // Top left
9     {{ 0.5f,  0.5f}}, // Top right
10 };

```

8.3.3 Step 3: Create a Vertex Buffer

Vertex data is stored in a GPU-accessible buffer. The following steps outline how to create a vertex buffer:

1. Allocate memory for the buffer using `vkCreateBuffer`.
2. Copy the vertex data into the buffer.
3. Bind the buffer for use in the rendering pipeline.

Refer to the Vulkan documentation or future chapters for detailed instructions on creating buffers.

8.3.4 Step 4: Write the Shaders

Vulkan uses SPIR-V (Standard Portable Intermediate Representation) for shaders. For this example, we need a vertex shader and a fragment shader.

- **Vertex Shader:** Positions the vertices of the triangle.
- **Fragment Shader:** Assigns a color to each fragment of the triangle.

Here are example GLSL shaders:

Vertex Shader (triangle.vert)

```
1 #version 450
2 layout(location = 0) in vec2 inPosition;
3
4 void main() {
5     gl_Position = vec4(inPosition, 0.0, 1.0);
6 }
```

Fragment Shader (triangle.frag)

```
1 #version 450
2 layout(location = 0) out vec4 outColor;
3
4 void main() {
5     outColor = vec4(1.0, 0.0, 0.0, 1.0); // Red
6 }
```

Compile these shaders to SPIR-V format using a compiler such as `glslangValidator`.

8.3.5 Step 5: Create a Graphics Pipeline

The graphics pipeline combines all the components required for rendering, such as shaders, fixed-function stages, and the render pass. Here is an outline of the pipeline creation:

1. Create shader modules from the compiled SPIR-V shaders.
2. Specify the input assembly, vertex input layout, viewport, rasterization, and other configurations.
3. Link everything into a graphics pipeline.

For example:

```
1 // Create graphics pipeline (simplified)
2 VkPipeline pipeline;
3 VkGraphicsPipelineCreateInfo pipelineInfo{};
4 // Set shader stages, vertex input, input assembly, viewport, rasterization, etc.
5 // For detailed steps, refer to Vulkan documentation or future chapters.
6 vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &pipeline);
```

8.3.6 Step 6: Record Command Buffers

Command buffers store the rendering commands to be executed by the GPU. For drawing the triangle, the commands include binding the pipeline, setting the viewport, and issuing a draw call.

```
1 // Record commands for drawing the triangle
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
3
4 // Bind the vertex buffer
5 VkBuffer vertexBuffers[] = {vertexBuffer};
6 VkDeviceSize offsets[] = {0};
```



```

7 vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, offsets);
8
9 // Issue the draw call
10 vkCmdDraw(commandBuffer, 3, 1, 0, 0);

```

8.3.7 Step 7: Submit Commands and Present the Triangle

Finally, submit the command buffer to the graphics queue and present the rendered triangle to the screen using the swapchain.

```

1 // Submit the command buffer
2 VkSubmitInfo submitInfo{};
3 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
4 submitInfo.commandBufferCount = 1;
5 submitInfo.pCommandBuffers = &commandBuffer;
6 vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
7
8 // Present the rendered image
9 VkPresentInfoKHR presentInfo{};
10 presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
11 // Set swapchain, image index, etc.
12 vkQueuePresentKHR(presentQueue, &presentInfo);

```

8.4 Complete Example Program

Here is a high-level summary of the complete process to render a triangle:

1. Initialize Vulkan and create a swapchain.
2. Create a vertex buffer and load the triangle's vertex data.
3. Write and compile shaders.
4. Create a graphics pipeline.
5. Record and execute commands to draw the triangle.
6. Present the rendered image.

The detailed implementation requires combining all the steps described above.

8.5 Summary

Rendering a triangle with Vulkan is a significant achievement. It demonstrates your understanding of Vulkan's rendering pipeline, including shaders, pipelines, and command buffers. From here, you can build more complex scenes by adding textures, multiple objects, and advanced features.

9

Render a Quad: Create and render a basic quad using two triangles

9.1 Introduction

In this chapter, we extend our knowledge by rendering a quad (a four-sided polygon). Since Vulkan does not have built-in support for rendering quads, we approximate a quad using two triangles. This exercise introduces important concepts like using index buffers to optimize rendering and managing shared vertices. The end of this chapter, you will have a quad rendered on the screen, using efficient techniques.

Why Render a Quad? Quads are commonly used in graphics programming as building blocks for more complex scenes. They are the basis for 2D sprites, UI elements, and even 3D models. Learning to render a quad equips you with foundational skills for advanced Vulkan projects.

9.2 Vertex Data for a Quad

A quad is made up of four vertices arranged in a rectangular shape. To render the quad using two triangles, we define the vertices as follows:

```
1 // Define vertex data for a quad
2 struct Vertex {
3     float position[2];
4 };
5
6 const std::vector<Vertex> vertices = {
7     {{-0.5f, -0.5f}}, // Bottom-left
```

```

8    {{ 0.5f, -0.5f}}, // Bottom-right
9    {{ 0.5f,  0.5f}}, // Top-right
10   {{-0.5f,  0.5f}} // Top-left
11 };

```

These vertices form a rectangle in normalized device coordinates. We connect them into two triangles for rendering.

9.3 Using an Index Buffer

Instead of duplicating vertices for each triangle, we use an **index buffer** to reference shared vertices. This reduces memory usage and improves performance. The indices for the two triangles are as follows:

```

1 // Define index data for the quad
2 const std::vector<uint16_t> indices = {
3     0, 1, 2, // First triangle
4     2, 3, 0 // Second triangle
5 };

```

Here, each number refers to a vertex in the `vertices` array. The first triangle connects vertices 0, 1, and 2, while the second triangle connects vertices 2, 3, and 0.

9.4 Creating Buffers for Vertices and Indices

To use the vertex and index data in Vulkan, we need to create **vertex buffers** and **index buffers**. These buffers store the data in GPU-accessible memory.

1. **Create a Vertex Buffer:** Allocate memory for the vertex data and copy it to the buffer.
2. **Create an Index Buffer:** Similarly, allocate memory for the index data and copy it to the buffer.

Refer to the Vulkan documentation or prior chapters for details on buffer creation.

9.5 Updating the Graphics Pipeline

Rendering a quad requires modifying the graphics pipeline to support indexed drawing. Specifically, we update the vertex input configuration to include an index buffer.

```

1 // Update vertex input configuration
2 VkPipelineVertexInputStateCreateInfo vertexInputInfo{0};
3 vertexInputInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
4
5 // Specify vertex buffer bindings and attributes
6 vertexInputInfo.vertexBindingDescriptionCount = 1;
7 vertexInputInfo.pVertexBindingDescriptions = &bindingDescription;
8 vertexInputInfo.vertexAttributeDescriptionCount = 1;
9 vertexInputInfo.pVertexAttributeDescriptions = &attributeDescription;

```

The rest of the pipeline configuration remains similar to the previous chapter.

9.6 Recording Command Buffers

When recording command buffers for the quad, bind both the vertex and index buffers before issuing the draw call.

```
1 // Bind the vertex buffer
2 VkBuffer vertexBuffers[] = {vertexBuffer};
3 VkDeviceSize offsets[] = {0};
4 vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, offsets);
5
6 // Bind the index buffer
7 vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT16);
8
9 // Issue the draw call for indexed rendering
10 vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);
```

Here, `vkCmdDrawIndexed` replaces `vkCmdDraw`. It uses the index buffer to determine the order in which vertices are processed.

9.7 Complete Example Program

Below is a summary of the key steps required to render a quad:

1. Define the vertex and index data for the quad.
2. Create vertex and index buffers to store the data on the GPU.
3. Update the graphics pipeline to support indexed rendering.
4. Record command buffers to bind the buffers and issue the draw call.
5. Submit the command buffer and present the rendered quad.

The full implementation involves combining these steps with the Vulkan setup from previous chapters.

9.8 Rendering Output

Once completed, the rendered output will display a solid-colored quad in the center of the screen. You can modify the vertex positions to change its size and shape.

9.9 Summary

In this chapter, you learned how to render a quad using two triangles and an index buffer. This approach introduces efficient rendering techniques and prepares you for working with more complex geometry. Leveraging index buffers, you reduce redundancy and optimize your application.

10

Add a Depth Buffer: Integrate depth buffering for 3D scenes

10.1 Introduction

In 3D graphics, the depth buffer is a crucial component for ensuring proper rendering of objects based on their distance from the camera. Without a depth buffer, objects closer to the camera might incorrectly appear behind farther objects, breaking the illusion of depth. In this chapter, you will learn how to integrate a depth buffer into your Vulkan application, enabling accurate 3D scene rendering.

Why Use a Depth Buffer? The depth buffer, also called the z-buffer, stores the depth of every pixel in the scene. During rendering, Vulkan uses this buffer to determine whether a new fragment should overwrite an existing one based on its depth value. This process ensures that objects closer to the viewer occlude those behind them.

10.2 Understanding Depth Buffers

The depth buffer is an image resource similar to a color attachment. However, instead of storing color data, it stores depth values as floating-point numbers, where smaller values represent closer objects. To use a depth buffer in Vulkan, you need to:

1. Create a depth image with a depth-compatible format.
2. Allocate memory for the depth image.
3. Attach the depth image to the render pass.
4. Configure depth testing in the graphics pipeline.

10.3 Step-by-Step Implementation

10.3.1 Step 1: Create the Depth Image

Start by creating a depth image with a suitable format, such as `VK_FORMAT_D32_SFLOAT`.

```

1 // Choose a depth format
2 VkFormat findDepthFormat() {
3     return VK_FORMAT_D32_SFLOAT; // Commonly supported format
4 }
5
6 // Create depth image
7 VkImageCreateInfo imageInfo{};
8 imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
9 imageInfo.imageType = VK_IMAGE_TYPE_2D;
10 imageInfo.extent.width = swapchainExtent.width;
11 imageInfo.extent.height = swapchainExtent.height;
12 imageInfo.extent.depth = 1;
13 imageInfo.mipLevels = 1;
14 imageInfo.arrayLayers = 1;
15 imageInfo.format = findDepthFormat();
16 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
17 imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
18 imageInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
19 imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
20 imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
21
22 if (vkCreateImage(device, &imageInfo, nullptr, &depthImage) != VK_SUCCESS) {
23     throw std::runtime_error("Failed to create depth image!");
24 }

```

This image will serve as the depth buffer for your application.

10.3.2 Step 2: Allocate Memory for the Depth Image

Allocate memory for the depth image and bind it.

```

1 // Allocate memory for the depth image
2 VkMemoryRequirements memRequirements;
3 vkGetImageMemoryRequirements(device, depthImage, &memRequirements);
4
5 VkMemoryAllocateInfo allocInfo{};
6 allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
7 allocInfo.allocationSize = memRequirements.size;
8 allocInfo.memoryTypeIndex = findMemoryType(
9     memRequirements.memoryTypeBits, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
10 );
11
12 if (vkAllocateMemory(device, &allocInfo, nullptr, &depthImageMemory) != VK_SUCCESS) {
13     throw std::runtime_error("Failed to allocate depth image memory!");
14 }
15
16 vkBindImageMemory(device, depthImage, depthImageMemory, 0);

```

The `findMemoryType` function matches memory properties with Vulkan's memory requirements.

10.3.3 Step 3: Create a Depth Image View

An image view is required to access the depth image.

```

1  // Create depth image view
2  VkImageViewCreateInfo viewInfo{};
3  viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
4  viewInfo.image = depthImage;
5  viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
6  viewInfo.format = findDepthFormat();
7  viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
8  viewInfo.subresourceRange.baseMipLevel = 0;
9  viewInfo.subresourceRange.levelCount = 1;
10 viewInfo.subresourceRange.baseArrayLayer = 0;
11 viewInfo.subresourceRange.layerCount = 1;
12
13 if (vkCreateImageView(device, &viewInfo, nullptr, &depthImageView) != VK_SUCCESS) {
14     throw std::runtime_error("Failed to create depth image view!");
15 }

```

The depth image view allows the depth buffer to be attached to the render pass.

10.3.4 Step 4: Attach the Depth Buffer to the Render Pass

Modify the render pass to include a depth attachment.

```

1  // Define depth attachment description
2  VkAttachmentDescription depthAttachment{};
3  depthAttachment.format = findDepthFormat();
4  depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
5  depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6  depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
7  depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
8  depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
9  depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
10 depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
11
12 // Add depth attachment to the render pass
13 VkAttachmentReference depthAttachmentRef{};
14 depthAttachmentRef.attachment = 1; // Assuming 0 is the color attachment
15 depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
16
17 VkSubpassDescription subpass{};
18 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
19 subpass.pDepthStencilAttachment = &depthAttachmentRef;

```

Update the render pass creation to include the depth attachment.

10.3.5 Step 5: Enable Depth Testing in the Graphics Pipeline

Configure depth testing in the pipeline's rasterization state.

```

1  // Enable depth testing
2  VkPipelineDepthStencilStateCreateInfo depthStencil{};
3  depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
4  depthStencil.depthTestEnable = VK_TRUE;
5  depthStencil.depthWriteEnable = VK_TRUE;

```

```
6 depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;  
7 depthStencil.depthBoundsTestEnable = VK_FALSE;  
8 depthStencil.stencilTestEnable = VK_FALSE;
```

Pass this configuration when creating the graphics pipeline.

10.4 Complete Example

Adding a depth buffer involves creating a depth image, allocating memory, creating a view, and integrating it into the render pass and pipeline. This process ensures correct depth ordering in 3D scenes.

10.5 Summary

This chapter detailed the integration of a depth buffer into a Vulkan application. Depth buffering is essential for rendering realistic 3D scenes, as it ensures that objects closer to the camera correctly occlude those behind them. In the next chapter, we will explore how to load and render 3D models, combining vertex data with depth testing for fully realized 3D scenes.

Part II

Intermediate Vulkan Rendering

11

Use Descriptor Sets: Implement descriptor sets for texture binding

11.1 Introduction

Descriptors in Vulkan are a powerful mechanism for passing resources such as textures, buffers, and samplers to shaders. Descriptor sets bundle these descriptors, making it easy to manage and bind resources at runtime. In this chapter, we will explore how to use descriptor sets to bind a texture to a fragment shader, enabling the rendering of textured objects.

We will walk through creating descriptor sets, writing data to them, and binding them during rendering. By the end of this chapter, you will understand how to implement descriptor sets for texture binding in Vulkan.

11.2 Overview of Descriptor Sets

Descriptor sets in Vulkan are part of the pipeline's resource binding system. They are created based on a **descriptor set layout**, which specifies the types of resources the set will manage. These resources are then associated with the set through a **descriptor pool**, which allocates the required memory.

The primary steps for using descriptor sets include:

1. Creating a descriptor set layout.
2. Allocating and writing descriptor sets.
3. Binding the descriptor set during rendering.

11.3 Step 1: Create a Descriptor Set Layout

The descriptor set layout defines the types of resources (e.g., a uniform buffer, a sampler) that will be bound to the shader. For texture binding, we specify a combined image sampler.

```

1 // Define descriptor set layout for a combined image sampler
2 VkDescriptorSetLayoutBinding samplerLayoutBinding{};
3 samplerLayoutBinding.binding = 0;
4 samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
5 samplerLayoutBinding.descriptorCount = 1;
6 samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
7 samplerLayoutBinding.pImmutableSamplers = nullptr;
8
9 VkDescriptorSetLayoutCreateInfo layoutInfo{};
10 layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
11 layoutInfo.bindingCount = 1;
12 layoutInfo.pBindings = &samplerLayoutBinding;
13
14 VkDescriptorSetLayout descriptorSetLayout;
15 if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK_SUCCESS) {
16     throw std::runtime_error("Failed to create descriptor set layout!");
17 }

```

This layout ensures the shader can access a texture through a combined image sampler.

11.4 Step 2: Create a Descriptor Pool

The descriptor pool manages the allocation of descriptor sets. It must be created with enough capacity for all required descriptors.

```

1 // Create a descriptor pool
2 VkDescriptorPoolSize poolSize{};
3 poolSize.type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
4 poolSize.descriptorCount = 1;
5
6 VkDescriptorPoolCreateInfo poolInfo{};
7 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
8 poolInfo.poolSizeCount = 1;
9 poolInfo.pPoolSizes = &poolSize;
10 poolInfo.maxSets = 1;
11
12 VkDescriptorPool descriptorPool;
13 if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
14     throw std::runtime_error("Failed to create descriptor pool!");
15 }

```

This pool allows for the allocation of descriptor sets for textures.

11.5 Step 3: Allocate and Write to Descriptor Sets

Descriptor sets are allocated from the descriptor pool and then populated with the relevant resource information.

```

1 // Allocate descriptor set

```

```

2  VkDescriptorSetAllocateInfo allocInfo{};
3  allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
4  allocInfo.descriptorPool = descriptorPool;
5  allocInfo.descriptorSetCount = 1;
6  allocInfo.pSetLayouts = &descriptorSetLayout;
7
8  VkDescriptorSet descriptorSet;
9  if (vkAllocateDescriptorSets(device, &allocInfo, &descriptorSet) != VK_SUCCESS) {
10     throw std::runtime_error("Failed to allocate descriptor set!");
11 }
12
13 // Write data to the descriptor set
14 VkDescriptorImageInfo imageInfo{};
15 imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
16 imageInfo.imageView = textureImageView; // Created earlier during texture setup
17 imageInfo.sampler = textureSampler;     // Created earlier during sampler setup
18
19 VkWriteDescriptorSet descriptorWrite{};
20 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
21 descriptorWrite.dstSet = descriptorSet;
22 descriptorWrite.dstBinding = 0;
23 descriptorWrite.dstArrayElement = 0;
24 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
25 descriptorWrite.descriptorCount = 1;
26 descriptorWrite.pImageInfo = &imageInfo;
27
28 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);

```

This step associates the texture's image view and sampler with the descriptor set.

11.6 Step 4: Bind the Descriptor Set During Rendering

Before issuing draw commands, bind the descriptor set to the pipeline.

```

1  // Bind descriptor set during rendering
2  vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
3                          pipelineLayout, 0, 1, &descriptorSet, 0, nullptr);

```

This makes the texture accessible to the shader for rendering.

11.7 Using the Texture in Shaders

In the fragment shader, you can sample the texture using the descriptor set.

```

1  // Fragment shader code
2  #version 450
3
4  layout(binding = 0) uniform sampler2D texSampler;
5
6  layout(location = 0) in vec2 fragTexCoord;
7  layout(location = 0) out vec4 outColor;
8
9  void main() {
10     outColor = texture(texSampler, fragTexCoord);
11 }

```

The `sampler2D` object is bound to the combined image sampler in the descriptor set, enabling

texture sampling.

11.8 Summary

Descriptor sets provide a flexible and efficient mechanism for managing resources in Vulkan. In this chapter, you learned how to use descriptor sets for texture binding, from creating layouts to writing and binding descriptor sets. This approach allows you to render textured objects with Vulkan's powerful resource management system.

DRAFT
version
0.681

12

Draw a Textured Triangle: Add a basic texture to your triangle

12.1 Introduction

Rendering textured geometry is a cornerstone of graphics programming, allowing for detailed visual representations of objects. In this chapter, you will extend the basic triangle rendering from earlier chapters by applying a texture. This involves loading an image, creating a sampler, and passing texture coordinates to the vertex shader.

Why Use Textures? Textures enrich the visual detail of rendered objects without increasing geometric complexity. By mapping 2D image data onto 3D models, you can achieve realistic appearances, such as patterns, colors, and even simulated surface details.

This chapter guides you through the essential steps to add a texture to your triangle using Vulkan.

12.2 Steps to Add a Texture

Integrating textures involves the following steps:

1. Load a texture image into memory.
2. Create a Vulkan image and transfer the texture data.
3. Create an image view and a sampler.
4. Update shaders to handle texture coordinates and sampling.
5. Pass texture coordinates to the vertex shader.

12.2.1 Step 1: Load the Texture Image

The first step is to load a 2D texture image into CPU memory. For simplicity, we will use a third-party library like `stb_image.h`.

```

1 // Load image with stb_image
2 int texWidth, texHeight, texChannels;
3 stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
4
5 if (!pixels) {
6     throw std::runtime_error("Failed to load texture image!");
7 }
8
9 VkDeviceSize imageSize = texWidth * texHeight * 4;
```

This code loads the texture into memory and calculates its size. Ensure that the image file is in the same directory or provide the correct path.

12.2.2 Step 2: Create a Vulkan Image and Transfer the Data

Create a Vulkan image to hold the texture and transfer the image data from CPU memory.

```

1 // Create Vulkan image
2 VkImageCreateInfo imageInfo{};
3 imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
4 imageInfo.imageType = VK_IMAGE_TYPE_2D;
5 imageInfo.extent.width = texWidth;
6 imageInfo.extent.height = texHeight;
7 imageInfo.extent.depth = 1;
8 imageInfo.mipLevels = 1;
9 imageInfo.arrayLayers = 1;
10 imageInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
11 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
12 imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
13 imageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
14 imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
15 imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
16
17 if (vkCreateImage(device, &imageInfo, nullptr, &textureImage) != VK_SUCCESS) {
18     throw std::runtime_error("Failed to create texture image!");
19 }
20
21 // Allocate memory and bind it to the image (similar to the depth buffer steps)
```

After creating the image, transfer the texture data using a staging buffer, just as you would for vertex data.

12.2.3 Step 3: Create an Image View and Sampler

Create an image view for the texture, which allows shaders to access the image.

```

1 // Create image view
2 VkImageViewCreateInfo viewInfo{};
3 viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
4 viewInfo.image = textureImage;
5 viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
```

```

6 viewInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
7 viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
8 viewInfo.subresourceRange.baseMipLevel = 0;
9 viewInfo.subresourceRange.levelCount = 1;
10 viewInfo.subresourceRange.baseArrayLayer = 0;
11 viewInfo.subresourceRange.layerCount = 1;
12
13 if (vkCreateImageView(device, &viewInfo, nullptr, &textureImageView) != VK_SUCCESS) {
14     throw std::runtime_error("Failed to create texture image view!");
15 }

```

Next, create a sampler to control how the texture is sampled.

```

1 // Create texture sampler
2 VkSamplerCreateInfo samplerInfo{};
3 samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
4 samplerInfo.magFilter = VK_FILTER_LINEAR;
5 samplerInfo.minFilter = VK_FILTER_LINEAR;
6 samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
7 samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
8 samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
9 samplerInfo.anisotropyEnable = VK_TRUE;
10 samplerInfo.maxAnisotropy = 16;
11 samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
12 samplerInfo.unnormalizedCoordinates = VK_FALSE;
13
14 if (vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler) != VK_SUCCESS) {
15     throw std::runtime_error("Failed to create texture sampler!");
16 }

```

12.2.4 Step 4: Update Shaders to Handle Textures

Modify the fragment shader to include a sampler and texture coordinates.

```

1 // Fragment shader code
2 #version 450
3
4 layout(binding = 0) uniform sampler2D texSampler;
5
6 layout(location = 0) in vec2 fragTexCoord;
7 layout(location = 0) out vec4 outColor;
8
9 void main() {
10     outColor = texture(texSampler, fragTexCoord);
11 }

```

Similarly, update the vertex shader to pass texture coordinates to the fragment shader.

12.2.5 Step 5: Update Vertex Data and Descriptor Sets

Define texture coordinates for the vertices of your triangle and update the vertex buffer.

```

1 struct Vertex {
2     glm::vec2 pos;
3     glm::vec2 texCoord;
4 };
5
6 // Example vertex data

```



```

7  std::vector<Vertex> vertices = {
8      {{0.0f, -0.5f}, {0.5f, 0.0f}}, // Bottom vertex
9      {{0.5f, 0.5f}, {1.0f, 1.0f}}, // Top-right vertex
10     {{-0.5f, 0.5f}, {0.0f, 1.0f}}, // Top-left vertex
11 };

```

Update descriptor sets to bind the texture image and sampler.

```

1  // Bind the image view and sampler to descriptor sets
2  VkDescriptorImageInfo imageInfo{};
3  imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
4  imageInfo.imageView = textureImageView;
5  imageInfo.sampler = textureSampler;

```

12.3 Complete Example

By combining the above steps, you can render a textured triangle in Vulkan. Ensure all pipeline and descriptor updates are synchronized with the new texture resources.

12.4 Summary

This chapter demonstrated how to add a texture to a triangle using Vulkan. Texturing enhances the visual appeal of your rendered scenes and lays the foundation for more complex techniques, such as multi-texturing and normal mapping.

13

Animate Triangle Movement: Add dynamic transformation (e.g., rotate, scale)

13.1 Introduction

Static scenes often lack the dynamism needed for engaging graphics applications. Adding animation to your rendering pipeline is a natural next step. In this chapter, you will learn how to animate a triangle by applying dynamic transformations, such as rotation, scaling, or translation, using a uniform buffer updated on each frame.

Why Animate Objects? Animation is critical for creating interactive and visually compelling applications. It introduces motion and life to rendered objects, providing a foundation for real-time applications like games and simulations.

This chapter will cover the use of uniform buffers to pass transformation matrices to shaders and the mechanics of updating these matrices for animation.

13.2 Steps to Animate a Triangle

Animating a triangle involves three main steps:

1. Create and update a transformation matrix.
2. Pass the matrix to the vertex shader using a uniform buffer.
3. Apply the transformation in the vertex shader.

13.2.1 Step 1: Create and Update a Transformation Matrix

Transformations in Vulkan are typically represented by 4x4 matrices, which can be used to perform operations like rotation, scaling, and translation. The `glm` library is commonly used for matrix manipulation.

```

1 // Include glm for matrix calculations
2 #include <glm/glm.hpp>
3 #include <glm/gtc/matrix_transform.hpp>
4
5 // Initialize transformation variables
6 float time = 0.0f; // Elapsed time for animation
7
8 // Create a dynamic transformation matrix
9 glm::mat4 createTransformationMatrix(float time) {
10     glm::mat4 model = glm::mat4(1.0f);
11     model = glm::rotate(model, glm::radians(time * 50.0f), glm::vec3(0.0f, 0.0f, 1.0f)); // Rotate
12     model = glm::scale(model, glm::vec3(0.5f + 0.5f * sin(time), 0.5f + 0.5f * sin(time), 1.0f)); // Scale
13     return model;
14 }
```

The `createTransformationMatrix` function generates a matrix that combines rotation and scaling based on the elapsed time, creating a dynamic animation.

13.2.2 Step 2: Pass the Matrix to the Vertex Shader

To pass the transformation matrix to the vertex shader, use a uniform buffer.

```

1 // Create a uniform buffer for the transformation matrix
2 VkBuffer uniformBuffer;
3 VkDeviceMemory uniformBufferMemory;
4
5 // Allocate and map memory for the uniform buffer
6 void createUniformBuffer() {
7     VkDeviceSize bufferSize = sizeof(glm::mat4);
8
9     createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
10                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
11                 uniformBuffer, uniformBufferMemory);
12 }
13
14 // Update the uniform buffer with the transformation matrix
15 void updateUniformBuffer(float time) {
16     glm::mat4 modelMatrix = createTransformationMatrix(time);
17
18     void* data;
19     vkMapMemory(device, uniformBufferMemory, 0, sizeof(modelMatrix), 0, &data);
20     memcpy(data, &modelMatrix, sizeof(modelMatrix));
21     vkUnmapMemory(device, uniformBufferMemory);
22 }
```

The uniform buffer is updated every frame with the current transformation matrix. The `vkMapMemory` and `vkUnmapMemory` calls ensure that the data is properly written to GPU memory.

13.2.3 Step 3: Apply the Transformation in the Vertex Shader

The vertex shader retrieves the transformation matrix from the uniform buffer and applies it to each vertex.

```

1  // Vertex shader code
2  #version 450
3
4  layout(location = 0) in vec2 inPos;
5  layout(location = 1) in vec2 inTexCoord;
6
7  layout(location = 0) out vec2 fragTexCoord;
8
9  layout(set = 0, binding = 0) uniform UniformBufferObject {
10     mat4 model;
11 } ubo;
12
13 void main() {
14     gl_Position = ubo.model * vec4(inPos, 0.0, 1.0);
15     fragTexCoord = inTexCoord;
16 }
```

The matrix `ubo.model` is used to transform the vertex position before it is output to the graphics pipeline.

13.2.4 Integrate Animation into the Render Loop

In the rendering loop, calculate the elapsed time and update the uniform buffer.

```

1  // Render loop
2  auto startTime = std::chrono::high_resolution_clock::now();
3
4  while (!windowShouldClose()) {
5      auto currentTime = std::chrono::high_resolution_clock::now();
6      float time = std::chrono::duration<float, std::chrono::seconds::period>(currentTime - startTime).count();
7
8      updateUniformBuffer(time);
9
10     drawFrame(); // Render the frame
11 }
```

This code computes the time elapsed since the application started and uses it to drive the animation.

13.3 Summary

In this chapter, you learned how to animate a triangle using dynamic transformations. By leveraging uniform buffers and matrix operations, you can bring motion and life to your graphics. These principles extend to animating more complex scenes and objects.

14

Add Push Constants: Use Push Constants for Efficient Data Passing to Shaders

14.1 Introduction

As your Vulkan application grows more complex, you'll need to pass dynamic data to shaders frequently. While uniform buffers are well-suited for passing large or persistent data, they can introduce latency and require careful synchronization. Push constants, on the other hand, allow small amounts of data to be passed to shaders with minimal overhead, making them ideal for frequently updated or lightweight parameters.

What Are Push Constants? Push constants are a small, fixed-size region of memory that is directly accessible to shaders. They provide a highly efficient way to pass per-frame or per-draw information, such as transformation matrices or material properties.

This chapter will show you how to set up push constants, define them in shaders, and use them in a practical example to control the color of a rendered object dynamically.

14.2 Setting Up Push Constants in Vulkan

To use push constants, you need to define their layout in both the Vulkan pipeline and the shader. Vulkan requires explicit declaration of the size and layout of push constants.

14.2.1 Step 1: Define a Push Constant Range

The push constant range specifies the size and shader stage(s) that will use the data. For this example, we'll define a push constant range that holds a color vector.

```
1 // Define push constant range
2 VkPushConstantRange pushConstantRange{};
3 pushConstantRange.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT; // Used in fragment shader
4 pushConstantRange.offset = 0;
5 pushConstantRange.size = sizeof(glm::vec4); // For a vec4 color
```

14.2.2 Step 2: Update the Pipeline Layout

Integrate the push constant range into the pipeline layout during its creation.

```
1 // Create pipeline layout with push constants
2 VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
3 pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
4 pipelineLayoutInfo.setLayoutCount = 0; // No descriptor sets for this example
5 pipelineLayoutInfo.pSetLayouts = nullptr;
6 pipelineLayoutInfo.pushConstantRangeCount = 1;
7 pipelineLayoutInfo.pPushConstantRanges = &pushConstantRange;
8
9 if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &pipelineLayout) != VK_SUCCESS) {
10     throw std::runtime_error("Failed to create pipeline layout!");
11 }
```

The pipeline layout now includes the push constant range, enabling shaders to access the data.

14.3 Using Push Constants in Shaders

Shaders must declare the layout of push constants to match the range defined in Vulkan.

```
1 // Fragment shader with push constant
2 #version 450
3
4 layout(push_constant) uniform PushConstants {
5     vec4 color;
6 } pc;
7
8 layout(location = 0) out vec4 outColor;
9
10 void main() {
11     outColor = pc.color; // Use push constant color
12 }
```

The `PushConstants` structure in the shader corresponds to the push constant range defined in Vulkan, allowing the shader to access the passed color.

14.4 Passing Push Constants During Rendering

Push constants are updated during the command recording phase using `vkCmdPushConstants`. For example, you can dynamically set the color of an object for each draw call.

```

1 // Example of passing a push constant during rendering
2 void recordCommandBuffer(VkCommandBuffer commandBuffer, VkPipelineLayout pipelineLayout) {
3     glm::vec4 dynamicColor = glm::vec4(1.0f, 0.5f, 0.3f, 1.0f); // Example color
4
5     vkCmdPushConstants(
6         commandBuffer,
7         pipelineLayout,
8         VK_SHADER_STAGE_FRAGMENT_BIT,
9         0,
10        sizeof(dynamicColor),
11        &dynamicColor
12    );
13
14    // Continue with binding pipeline and issuing draw commands
15    vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
16    vkCmdDraw(commandBuffer, 3, 1, 0, 0); // Draw a triangle
17 }
```

This example shows how to bind a dynamic color to the fragment shader using push constants.

14.5 Practical Example: Dynamic Object Color

Let's use push constants to dynamically change the color of a triangle during rendering. By updating the color before each draw call, you can create effects like flashing, color cycling, or representing per-object states.

```

1 // Dynamic color update in render loop
2 float time = 0.0f;
3
4 while (!windowShouldClose()) {
5     time += deltaTime; // Calculate elapsed time
6     glm::vec4 color = glm::vec4(sin(time) * 0.5f + 0.5f, cos(time) * 0.5f + 0.5f, 0.5f, 1.0f);
7
8     vkCmdPushConstants(commandBuffer, pipelineLayout, VK_SHADER_STAGE_FRAGMENT_BIT, 0, sizeof(color), &color);
9
10    drawFrame(); // Record commands and present the frame
11 }
```

The color varies over time, demonstrating how push constants can efficiently control dynamic behavior in shaders.

14.6 Summary

In this chapter, you learned how to use push constants to pass lightweight, dynamic data to shaders. Push constants provide a low-latency alternative to uniform buffers for small amounts of data, making them ideal for per-draw or per-object parameters.

With this knowledge, you can now experiment with advanced per-object shading effects, ma-

terial parameters, or even control game logic directly from the rendering pipeline.

DRAFT
version
0.681

15

Create a Uniform Buffer: Pass data (e.g., transformations) to the shader

15.1 Introduction

While push constants are great for small, frequently updated data, uniform buffers are designed to handle larger, persistent data shared across multiple frames or draw calls. Uniform buffers are commonly used to pass transformation matrices, lighting parameters, and other global data to shaders.

In this chapter, you will learn how to create and manage a uniform buffer in Vulkan, bind it to a shader, and update it during runtime. By the end, you'll have the foundation for implementing features such as camera transformations and per-object transformations.

15.2 Understanding Uniform Buffers

A uniform buffer is a memory region accessible to shaders via descriptor sets. Unlike push constants, uniform buffers are suitable for larger data sets and are particularly useful when the same data needs to be shared among multiple shaders or objects.

15.3 Steps to Create and Use a Uniform Buffer

To implement a uniform buffer, you will:

1. Define the uniform buffer structure and create it in Vulkan.

2. Update the buffer with transformation data.
3. Bind the buffer to shaders using a descriptor set.

15.3.1 Step 1: Define and Create a Uniform Buffer

Let's start by defining the data structure for the uniform buffer and creating it in Vulkan.

```

1 // Define the Uniform Buffer Object (UBO) structure
2 struct UniformBufferObject {
3     glm::mat4 model;
4     glm::mat4 view;
5     glm::mat4 proj;
6 };

```

The UBO contains transformation matrices for the model, view (camera), and projection stages.

```

1 // Create the uniform buffer
2 VkBuffer uniformBuffer;
3 VkDeviceMemory uniformBufferMemory;
4
5 void createUniformBuffer() {
6     VkDeviceSize bufferSize = sizeof(UniformBufferObject);
7
8     createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
9                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
10                uniformBuffer, uniformBufferMemory);
11 }

```

The `createBuffer` function initializes the uniform buffer with appropriate usage and memory properties, making it accessible for updates on the CPU.

15.3.2 Step 2: Update the Uniform Buffer with Data

Uniform buffers are typically updated with per-frame data, such as camera transformations. The following example computes the model, view, and projection matrices and uploads them to the uniform buffer.

```

1 // Update the uniform buffer with transformation data
2 void updateUniformBuffer(float aspectRatio, float time) {
3     UniformBufferObject ubo{};
4     ubo.model = glm::rotate(glm::mat4(1.0f), time * glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));
5     ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f)
6     ↪ );
7     ubo.proj = glm::perspective(glm::radians(45.0f), aspectRatio, 0.1f, 10.0f);
8     ubo.proj[1][1] *= -1; // Vulkan clip space correction
9
10    void* data;
11    vkMapMemory(device, uniformBufferMemory, 0, sizeof(ubo), 0, &data);
12    memcpy(data, &ubo, sizeof(ubo));
13    vkUnmapMemory(device, uniformBufferMemory);
14 }

```

The matrices are computed using `glm` and copied to the uniform buffer memory.

15.3.3 Step 3: Bind the Uniform Buffer to Shaders

Uniform buffers are accessed by shaders through descriptor sets. To use a uniform buffer, you must:

1. Define a descriptor set layout.
2. Allocate and update descriptor sets.

```

1 // Define the descriptor set layout
2 VkDescriptorSetLayoutBinding uboLayoutBinding{};
3 uboLayoutBinding.binding = 0;
4 uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
5 uboLayoutBinding.descriptorCount = 1;
6 uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
7 uboLayoutBinding.pImmutableSamplers = nullptr; // Not used
8
9 VkDescriptorSetLayoutCreateInfo layoutInfo{};
10 layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
11 layoutInfo.bindingCount = 1;
12 layoutInfo.pBindings = &uboLayoutBinding;
13
14 if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK_SUCCESS) {
15     throw std::runtime_error("Failed to create descriptor set layout!");
16 }
```

The descriptor set layout specifies the binding for the uniform buffer.

```

1 // Allocate and update descriptor set
2 VkDescriptorBufferInfo bufferInfo{};
3 bufferInfo.buffer = uniformBuffer;
4 bufferInfo.offset = 0;
5 bufferInfo.range = sizeof(UniformBufferObject);
6
7 VkWriteDescriptorSet descriptorWrite{};
8 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9 descriptorWrite.dstSet = descriptorSet;
10 descriptorWrite.dstBinding = 0;
11 descriptorWrite.dstArrayElement = 0;
12 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
13 descriptorWrite.descriptorCount = 1;
14 descriptorWrite.pBufferInfo = &bufferInfo;
15
16 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

The uniform buffer is now bound to the descriptor set and ready for use in shaders.

15.3.4 Access the Uniform Buffer in Shaders

The shader must declare the uniform buffer to match the Vulkan descriptor set.

```

1 // Vertex shader code
2 #version 450
3
4 layout(binding = 0) uniform UniformBufferObject {
5     mat4 model;
6     mat4 view;
7     mat4 proj;
8 } ubo;
9
```



```
10 layout(location = 0) in vec2 inPos;
11
12 layout(location = 0) out vec3 fragPos;
13
14 void main() {
15     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPos, 0.0, 1.0);
16     fragPos = vec3(ubo.model * vec4(inPos, 0.0, 1.0));
17 }
```

The uniform buffer provides the transformation matrices to the vertex shader, enabling transformations of vertex positions.

15.4 Summary

In this chapter, you learned how to create and use a uniform buffer in Vulkan to pass transformation data to shaders. Uniform buffers are essential for managing data shared across frames or objects, such as camera parameters or lighting information.

This knowledge sets the stage for rendering more complex scenes with dynamic transformations.

16

Use Descriptor Sets for Multiple Objects: Manage Data for Multiple Objects

16.1 Introduction

As your Vulkan application grows, you'll often need to render multiple objects, each with its own set of parameters, such as transformation matrices, material properties, or textures. Managing data for multiple objects efficiently requires using descriptor sets to associate each object with its unique data.

In this chapter, you will learn how to use descriptor sets for rendering multiple objects. This approach scales well for complex scenes, ensuring that each object has access to its specific data in a structured and efficient manner.

16.2 Understanding Descriptor Sets for Multiple Objects

Descriptor sets in Vulkan act as bridges between application-side resources and shaders. By using one descriptor set per object, you can associate unique data (e.g., a uniform buffer or texture) with each object. This provides a flexible and modular approach for rendering scenes with many objects.

16.3 Steps to Use Descriptor Sets for Multiple Objects

To implement descriptor sets for multiple objects, follow these steps:

1. Create a uniform buffer for each object.
2. Allocate a descriptor set for each object and bind its unique data.
3. Update and bind descriptor sets during rendering.

16.3.1 Step 1: Create a Uniform Buffer for Each Object

Each object requires its own uniform buffer to hold unique data, such as transformation matrices.

```

1 // Define data for each object
2 struct ObjectData {
3     glm::mat4 model; // Model transformation matrix
4 };
5
6 // Create uniform buffers for multiple objects
7 std::vector<VkBuffer> uniformBuffers(numObjects);
8 std::vector<VkDeviceMemory> uniformBufferMemories(numObjects);
9
10 for (size_t i = 0; i < numObjects; i++) {
11     createBuffer(sizeof(ObjectData), VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
12                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
13                 uniformBuffers[i], uniformBufferMemories[i]);
14 }
```

Each object gets its own uniform buffer, ensuring that transformations or other unique data can be updated independently.

16.3.2 Step 2: Allocate Descriptor Sets for Each Object

Descriptor sets link the uniform buffers to the shaders. Allocate and configure a descriptor set for each object.

```

1 // Allocate descriptor sets for each object
2 std::vector<VkDescriptorSet> descriptorSets(numObjects);
3
4 for (size_t i = 0; i < numObjects; i++) {
5     VkDescriptorSetAllocateInfo allocInfo{};
6     allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
7     allocInfo.descriptorPool = descriptorPool;
8     allocInfo.descriptorSetCount = 1;
9     allocInfo.pSetLayouts = &descriptorSetLayout;
10
11     if (vkAllocateDescriptorSets(device, &allocInfo, &descriptorSets[i]) != VK_SUCCESS) {
12         throw std::runtime_error("Failed to allocate descriptor sets!");
13     }
14
15     // Configure descriptor set for the current object
16     VkDescriptorBufferInfo bufferInfo{};
17     bufferInfo.buffer = uniformBuffers[i];
18     bufferInfo.offset = 0;
19     bufferInfo.range = sizeof(ObjectData);
20
21     VkWriteDescriptorSet descriptorWrite{};
22     descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
23     descriptorWrite.dstSet = descriptorSets[i];
24     descriptorWrite.dstBinding = 0;
25     descriptorWrite.dstArrayElement = 0;
26     descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
```

```

27 descriptorWrite.descriptorCount = 1;
28 descriptorWrite.pBufferInfo = &bufferInfo;
29
30 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
31 }

```

This process ensures that each descriptor set is linked to a unique uniform buffer.

16.3.3 Step 3: Update Descriptor Sets During Rendering

Before rendering each object, update its uniform buffer and bind the corresponding descriptor set.

```

1 // Update and bind descriptor sets for each object
2 for (size_t i = 0; i < numObjects; i++) {
3     // Update the uniform buffer with object-specific data
4     ObjectData objectData{};
5     objectData.model = glm::translate(glm::mat4(1.0f), objectPositions[i]);
6
7     void* data;
8     vkMapMemory(device, uniformBufferMemories[i], 0, sizeof(objectData), 0, &data);
9     memcpy(data, &objectData, sizeof(objectData));
10    vkUnmapMemory(device, uniformBufferMemories[i]);
11
12    // Bind the descriptor set for the current object
13    vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
14        pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
15
16    // Issue draw command
17    vkCmdDraw(commandBuffer, 6, 1, 0, 0); // Assuming a quad with 6 vertices
18 }

```

This approach ensures that each object is rendered with its unique transformation data.

16.4 Accessing Data in the Shader

The shader retrieves data from the uniform buffer associated with the currently bound descriptor set.

```

1 // Vertex shader code
2 #version 450
3
4 layout(binding = 0) uniform ObjectData {
5     mat4 model;
6 } objectData;
7
8 layout(location = 0) in vec2 inPos;
9
10 layout(location = 0) out vec3 fragPos;
11
12 void main() {
13     gl_Position = objectData.model * vec4(inPos, 0.0, 1.0);
14     fragPos = vec3(objectData.model * vec4(inPos, 0.0, 1.0));
15 }

```

The `model` matrix is fetched from the uniform buffer for each object, enabling object-specific

transformations.

16.5 Summary

In this chapter, you learned how to use descriptor sets to manage data for multiple objects. This approach scales well for complex scenes, allowing each object to have its own transformation, material properties, or other unique data.

By leveraging descriptor sets, you've taken an important step towards rendering dynamic and realistic scenes in Vulkan.

DRAFT
version
0.681

17

Render Multiple Objects: Draw multiple triangles or quads in a single pass

17.1 Introduction

Rendering multiple objects in Vulkan requires efficient management of resources and a well-structured rendering pipeline. Unlike rendering a single object, rendering multiple objects in a single pass involves batching draw calls, leveraging instance data, or dynamically binding resources such as descriptor sets.

In this chapter, you will learn how to draw multiple objects, such as triangles or quads, in a single render pass. We will focus on two approaches: issuing separate draw calls for each object and using instanced rendering for improved performance. By the end of this chapter, you will have the skills to render complex scenes efficiently.

17.2 Drawing Multiple Objects with Separate Draw Calls

The simplest approach to render multiple objects is to issue separate draw calls for each object. Each draw call specifies the object's transformation and resources.

17.2.1 Step 1: Create Per-Object Resources

Each object requires a unique set of transformation data, as shown in the previous chapter. This involves creating a uniform buffer or a push constant for each object.

```
1 // Define per-object data
```



```

2 struct ObjectData {
3     glm::mat4 model;
4 };
5
6 // Create uniform buffers and descriptor sets for multiple objects
7 std::vector<VkBuffer> uniformBuffers(numObjects);
8 std::vector<VkDeviceMemory> uniformBufferMemories(numObjects);
9 std::vector<VkDescriptorSet> descriptorSets(numObjects);
10
11 for (size_t i = 0; i < numObjects; i++) {
12     createBuffer(sizeof(ObjectData), VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
13                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
14                 uniformBuffers[i], uniformBufferMemories[i]);
15
16     allocateAndUpdateDescriptorSet(descriptorSets[i], uniformBuffers[i]);
17 }

```

This setup ensures that each object has unique transformation data accessible to the shaders.

17.2.2 Step 2: Render Objects in a Loop

In the render loop, iterate over all objects and issue separate draw commands for each one. Update the uniform buffer with the object's transformation before each draw call.

```

1 // Render multiple objects
2 for (size_t i = 0; i < numObjects; i++) {
3     // Update the uniform buffer with per-object data
4     ObjectData objectData{};
5     objectData.model = glm::translate(glm::mat4(1.0f), objectPositions[i]);
6
7     void* data;
8     vkMapMemory(device, uniformBufferMemories[i], 0, sizeof(objectData), 0, &data);
9     memcpy(data, &objectData, sizeof(objectData));
10    vkUnmapMemory(device, uniformBufferMemories[i]);
11
12    // Bind the descriptor set for the current object
13    vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
14                           pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);
15
16    // Issue draw command
17    vkCmdDraw(commandBuffer, 6, 1, 0, 0); // Draw a quad with 6 vertices
18 }

```

This method is straightforward but may result in performance bottlenecks due to frequent state changes and draw calls.

17.3 Using Instanced Rendering for Multiple Objects

Instanced rendering allows you to render multiple objects with a single draw call by providing per-instance data. This approach reduces CPU overhead and improves rendering performance.

17.3.1 Step 1: Create an Instance Buffer

The instance buffer contains per-instance data, such as transformation matrices, for all objects. This data is accessed by shaders during rendering.

```

1 // Define instance data
2 struct InstanceData {
3     glm::mat4 model;
4 };
5
6 // Create the instance buffer
7 VkBuffer instanceBuffer;
8 VkDeviceMemory instanceBufferMemory;
9
10 void createInstanceBuffer() {
11     VkDeviceSize bufferSize = sizeof(InstanceData) * numObjects;
12
13     createBuffer(bufferSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
14                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
15                 instanceBuffer, instanceBufferMemory);
16 }

```

17.3.2 Step 2: Update the Instance Buffer

Before rendering, populate the instance buffer with transformation data for all objects.

```

1 // Update instance buffer
2 void updateInstanceBuffer(const std::vector<glm::mat4>& objectTransforms) {
3     void* data;
4     vkMapMemory(device, instanceBufferMemory, 0, sizeof(InstanceData) * numObjects, 0, &data);
5     memcpy(data, objectTransforms.data(), sizeof(InstanceData) * numObjects);
6     vkUnmapMemory(device, instanceBufferMemory);
7 }

```

17.3.3 Step 3: Bind the Instance Buffer and Issue a Single Draw Call

Bind the instance buffer as an additional vertex buffer and issue a single draw call for all objects.

```

1 // Bind vertex and instance buffers
2 VkBuffer buffers[] = {vertexBuffer, instanceBuffer};
3 VkDeviceSize offsets[] = {0, 0};
4 vkCmdBindVertexBuffers(commandBuffer, 0, 2, buffers, offsets);
5
6 // Draw all instances in a single call
7 vkCmdDraw(commandBuffer, 6, numObjects, 0, 0);

```

This method significantly improves performance by minimizing draw calls and state changes.

17.4 Accessing Instance Data in the Shader

The vertex shader retrieves per-instance data from the instance buffer, enabling object-specific transformations.

```

1 // Vertex shader code
2 #version 450
3
4 layout(location = 0) in vec2 inPos; // Vertex position
5 layout(location = 1) in mat4 instanceModel; // Per-instance model matrix

```

```
6
7 layout(location = 0) out vec3 fragPos;
8
9 void main() {
10     gl_Position = instanceModel * vec4(inPos, 0.0, 1.0);
11     fragPos = vec3(instanceModel * vec4(inPos, 0.0, 1.0));
12 }
```

The shader applies the transformation matrix for each instance, enabling efficient rendering of multiple objects.

17.5 Summary

In this chapter, you explored two methods for rendering multiple objects: issuing separate draw calls and using instanced rendering. While separate draw calls are simple to implement, instanced rendering offers significant performance benefits for scenes with a large number of objects.

18

Add Staging Buffers: Optimize data transfer between CPU and GPU

18.1 Introduction

Efficient data transfer between the CPU and GPU is crucial for high-performance Vulkan applications. Directly uploading data from the CPU to a GPU-visible buffer is often suboptimal because GPU-visible memory is typically not host-coherent or host-visible. Instead, Vulkan provides a better solution: **staging buffers**, which act as intermediaries for transferring data.

In this chapter, you will learn how to use staging buffers to optimize data transfer, focusing on vertex data as an example. This approach minimizes overhead and improves performance by leveraging Vulkan's memory management features.

18.2 What Are Staging Buffers?

A staging buffer is a temporary, host-visible buffer located in memory optimized for CPU access. Data is first written to the staging buffer and then copied to a GPU-visible buffer. This allows you to benefit from the fast GPU memory while avoiding the limitations of direct CPU-to-GPU memory access.

18.3 Steps to Use Staging Buffers

To implement staging buffers effectively, follow these steps:

1. Create a host-visible staging buffer and copy data into it.
2. Create a GPU-visible buffer to store the final data.
3. Copy data from the staging buffer to the GPU buffer using a command.

18.3.1 Step 1: Create a Host-Visible Staging Buffer

The staging buffer is created with `VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT` and `VK_MEMORY_PROPERTY_HOST_COHERENT_BIT`, making it accessible for CPU writes.

```

1 // Create a staging buffer
2 VkBuffer stagingBuffer;
3 VkDeviceMemory stagingBufferMemory;
4
5 void createStagingBuffer(VkDeviceSize size, VkBuffer& buffer, VkDeviceMemory& memory) {
6     createBuffer(size, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
7                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
8                 buffer, memory);
9 }
```

This staging buffer serves as an intermediary for the data transfer process.

18.3.2 Step 2: Create a GPU-Visible Buffer

The GPU buffer is created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` and located in device-local memory. This memory is optimized for GPU access but not directly accessible by the CPU.

```

1 // Create a GPU-visible buffer
2 VkBuffer gpuBuffer;
3 VkDeviceMemory gpuBufferMemory;
4
5 void createGPUBuffer(VkDeviceSize size, VkBuffer& buffer, VkDeviceMemory& memory) {
6     createBuffer(size, VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
7                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
8                 buffer, memory);
9 }
```

This buffer will hold the vertex data for use by the GPU.

18.3.3 Step 3: Copy Data to the Staging Buffer

Write the vertex data to the staging buffer using a mapped memory region.

```

1 // Copy data to the staging buffer
2 void* data;
3 vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
4 memcpy(data, vertexData.data(), static_cast<size_t>(bufferSize));
5 vkUnmapMemory(device, stagingBufferMemory);
```

18.3.4 Step 4: Transfer Data to the GPU Buffer

To transfer data from the staging buffer to the GPU buffer, use a command buffer with a copy operation.

```

1  // Copy data from staging buffer to GPU buffer
2  void copyBuffer(VkBuffer srcBuffer, VkBuffer dstBuffer, VkDeviceSize size) {
3      VkCommandBuffer commandBuffer = beginSingleTimeCommands();
4
5      VkBufferCopy copyRegion{};
6      copyRegion.srcOffset = 0;
7      copyRegion.dstOffset = 0;
8      copyRegion.size = size;
9      vkCmdCopyBuffer(commandBuffer, srcBuffer, dstBuffer, 1, &copyRegion);
10
11     endSingleTimeCommands(commandBuffer);
12 }

```

Here, the `beginSingleTimeCommands` and `endSingleTimeCommands` functions ensure the copy operation is recorded and executed as a one-time command.

18.4 Full Implementation Example

Bringing all the steps together, here's an example of uploading vertex data using a staging buffer:

```

1  // Vertex data
2  std::vector<Vertex> vertices = {
3      {{0.0f, -0.5f}, {1.0f, 0.0f, 0.0f}},
4      {{0.5f, 0.5f}, {0.0f, 1.0f, 0.0f}},
5      {{-0.5f, 0.5f}, {0.0f, 0.0f, 1.0f}}
6  };
7
8  VkDeviceSize bufferSize = sizeof(vertices[0]) * vertices.size();
9
10 // Step 1: Create a staging buffer
11 VkBuffer stagingBuffer;
12 VkDeviceMemory stagingBufferMemory;
13 createStagingBuffer(bufferSize, stagingBuffer, stagingBufferMemory);
14
15 // Step 2: Create a GPU-visible buffer
16 VkBuffer vertexBuffer;
17 VkDeviceMemory vertexBufferMemory;
18 createGPUBuffer(bufferSize, vertexBuffer, vertexBufferMemory);
19
20 // Step 3: Copy data to the staging buffer
21 void* data;
22 vkMapMemory(device, stagingBufferMemory, 0, bufferSize, 0, &data);
23 memcpy(data, vertices.data(), (size_t) bufferSize);
24 vkUnmapMemory(device, stagingBufferMemory);
25
26 // Step 4: Transfer data to the GPU buffer
27 copyBuffer(stagingBuffer, vertexBuffer, bufferSize);
28
29 // Cleanup
30 vkDestroyBuffer(device, stagingBuffer, nullptr);
31 vkFreeMemory(device, stagingBufferMemory, nullptr);

```


18.5 Performance Benefits of Staging Buffers

By using staging buffers, you can:

- **Leverage fast GPU memory:** Data resides in device-local memory optimized for GPU access.
- **Minimize latency:** Transfers occur in batches rather than individual writes.
- **Avoid CPU-GPU synchronization issues:** Use command buffers to ensure efficient data transfer.

18.6 Summary

In this chapter, you learned how to optimize data transfer between the CPU and GPU using staging buffers. This approach improves performance by separating the stages of data upload and leveraging Vulkan's memory management capabilities.

19

Generate a Procedural Gradient: Render a triangle with a procedurally generated gradient

19.1 Introduction

Procedural generation allows graphics applications to dynamically create patterns, textures, or visuals directly in shaders, without relying on precomputed assets. In this chapter, we will render a triangle with a procedural color gradient generated in the fragment shader. This example illustrates how Vulkan shaders can be leveraged for efficient, real-time visual effects.

19.2 What is a Procedural Gradient?

A procedural gradient is a smooth transition of colors that is calculated mathematically, often based on spatial coordinates. In our example, the gradient will transition colors across the triangle's surface based on its fragment positions in normalized device coordinates (NDC).

This task demonstrates the following Vulkan concepts:

- Setting up vertex attributes for positional data.
- Using shaders to compute fragment colors dynamically.
- Linking Vulkan resources like pipelines and shaders to achieve the effect.

19.3 Shader Implementation for Gradient

The gradient logic is implemented in the fragment shader. We calculate the color of each pixel based on its position relative to the triangle's area.

19.3.1 Vertex Shader

The vertex shader passes position and color interpolation data to the fragment shader. We add basic transformations to prepare the input.

```
1 // Vertex Shader (GLSL)
2 #version 450
3
4 layout(location = 0) in vec2 inPosition;
5
6 layout(location = 0) out vec2 fragCoord;
7
8 void main() {
9     gl_Position = vec4(inPosition, 0.0, 1.0);
10    fragCoord = inPosition * 0.5 + 0.5; // Normalize to [0, 1]
11 }
```

The normalized `fragCoord` is passed to the fragment shader to compute the gradient.

19.3.2 Fragment Shader

The fragment shader generates the gradient based on the interpolated position data.

```
1 // Fragment Shader (GLSL)
2 #version 450
3
4 layout(location = 0) in vec2 fragCoord;
5 layout(location = 0) out vec4 outColor;
6
7 void main() {
8     vec3 color = vec3(fragCoord.x, fragCoord.y, 1.0 - fragCoord.x);
9     outColor = vec4(color, 1.0);
10 }
```

The color is calculated using `fragCoord.x` and `fragCoord.y`, creating a smooth blend of red, green, and blue.

19.4 Vulkan Pipeline Setup

To render the procedural gradient, configure the Vulkan pipeline with the gradient shaders and appropriate vertex attributes.

19.4.1 Vertex Data

Define the vertices for the triangle.

```

1 // Triangle vertex data
2 std::vector<Vertex> vertices = {
3     {{0.0f, -0.5f}}, // Bottom
4     {{0.5f, 0.5f}}, // Top right
5     {{-0.5f, 0.5f}}, // Top left
6 };

```

The vertex buffer holds only position data, which the vertex shader normalizes to generate gradient inputs.

19.4.2 Graphics Pipeline

Attach the shaders to the Vulkan pipeline. Ensure the fragment shader is designed to process the interpolated position data.

```

1 // Load shaders
2 VkShaderModule vertShaderModule = createShaderModule("gradient_vert.spv");
3 VkShaderModule fragShaderModule = createShaderModule("gradient_frag.spv");
4
5 // Pipeline configuration
6 pipelineInfo.stageCount = 2;
7 VkPipelineShaderStageCreateInfo shaderStages[] = {vertShaderStageInfo, fragShaderStageInfo};
8 pipelineInfo.pStages = shaderStages;

```

The `gradient_vert.spv` and `gradient_frag.spv` are the SPIR-V compiled versions of the GLSL vertex and fragment shaders.

19.4.3 Rendering Loop

Render the triangle as a single draw call. Ensure the pipeline and vertex buffer are correctly bound.

```

1 // Bind the pipeline
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
3
4 // Bind vertex buffer
5 VkDeviceSize offsets[] = {0};
6 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
7
8 // Draw the triangle
9 vkCmdDraw(commandBuffer, static_cast<uint32_t>(vertices.size()), 1, 0, 0);

```

This sequence ensures the triangle is drawn with the procedural gradient calculated in the shaders.

19.5 Result

When executed, this code renders a triangle with a gradient smoothly transitioning between red, green, and blue. The gradient is entirely computed in the fragment shader, showcasing the power of procedural techniques for real-time effects.

19.6 Summary

This chapter demonstrated how to generate a procedural gradient using Vulkan and GLSL. By leveraging shader programming, you can dynamically compute visual effects, reducing dependency on precomputed assets and enabling flexibility in rendering. Procedural generation is a cornerstone of modern graphics, and this example provides a foundational understanding to build upon.

DRAFT
version
0.681

20

Add Anti-Aliasing: Use multisample anti-aliasing (MSAA)

20.1 Introduction

Rendering sharp edges often introduces visual artifacts, such as jagged lines, known as aliasing. **Multisample Anti-Aliasing (MSAA)** is a powerful technique in Vulkan that reduces these artifacts by sampling multiple points per pixel and averaging the results. MSAA improves the visual quality of rendered images while maintaining performance efficiency compared to other anti-aliasing methods.

In this chapter, you will learn how to integrate MSAA into your Vulkan application, including configuring the swapchain, setting up multisampled attachments, and modifying the render pipeline.

20.2 Overview of MSAA in Vulkan

MSAA works by sampling a pixel multiple times at different locations and averaging the sampled values. Vulkan supports MSAA by using:

1. A multisampled render target (color attachment).
2. A resolve operation that resolves the multisampled data into a single-sample image for display.

The steps to implement MSAA are:

- Choose a supported sample count.

- Create multisampled images and views.
- Update the render pass and pipeline to use MSAA.
- Resolve the multisampled image to a single-sample image.

20.3 Step 1: Determine Supported Sample Counts

First, query the GPU for supported MSAA sample counts. Vulkan provides this information in the `VkPhysicalDeviceProperties` structure.

```

1 // Query supported MSAA sample counts
2 VkSampleCountFlags getMaxUsableSampleCount() {
3     VkPhysicalDeviceProperties physicalDeviceProperties;
4     vkGetPhysicalDeviceProperties(physicalDevice, &physicalDeviceProperties);
5
6     VkSampleCountFlags counts = physicalDeviceProperties.limits.framebufferColorSampleCounts &
7                               physicalDeviceProperties.limits.framebufferDepthSampleCounts;
8     if (counts & VK_SAMPLE_COUNT_8_BIT) return VK_SAMPLE_COUNT_8_BIT;
9     if (counts & VK_SAMPLE_COUNT_4_BIT) return VK_SAMPLE_COUNT_4_BIT;
10    if (counts & VK_SAMPLE_COUNT_2_BIT) return VK_SAMPLE_COUNT_2_BIT;
11
12    return VK_SAMPLE_COUNT_1_BIT; // Default to no MSAA
13 }
14 VkSampleCountFlagBits msaaSamples = getMaxUsableSampleCount();

```

In this example, the highest supported sample count is selected, with a fallback to `VK_SAMPLE_COUNT_1_BIT` (no MSAA).

20.4 Step 2: Create Multisampled Image Attachments

Create a multisampled color attachment to store the intermediate MSAA results. This attachment should match the format of the swapchain images and use the selected sample count.

```

1 // Create a multisampled color attachment
2 void createColorAttachment() {
3     VkImageCreateInfo imageInfo{};
4     imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
5     imageInfo.imageType = VK_IMAGE_TYPE_2D;
6     imageInfo.extent.width = swapchainExtent.width;
7     imageInfo.extent.height = swapchainExtent.height;
8     imageInfo.extent.depth = 1;
9     imageInfo.mipLevels = 1;
10    imageInfo.arrayLayers = 1;
11    imageInfo.format = swapchainImageFormat;
12    imageInfo.samples = msaaSamples;
13    imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
14    imageInfo.usage = VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT | VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
15
16    vkCreateImage(device, &imageInfo, nullptr, &colorImage);
17    allocateImageMemory(colorImage, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, &colorImageMemory);
18
19    // Create an image view for the multisampled attachment
20    colorImageView = createImageView(colorImage, swapchainImageFormat, VK_IMAGE_ASPECT_COLOR_BIT);
21 }

```

This attachment will store the multisampled color data before resolving it to the final

swapchain image.

20.5 Step 3: Update the Render Pass

The render pass must include a multisampled color attachment and a resolve attachment. The multisampled image will store intermediate data, while the resolve attachment outputs the final single-sampled image.

```
1  // Render pass attachments
2  VkAttachmentDescription colorAttachment{};
3  colorAttachment.format = swapchainImageFormat;
4  colorAttachment.samples = msaaSamples;
5  colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6  colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
7  colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
8  colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
9  colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
10 colorAttachment.finalLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
11
12 VkAttachmentDescription resolveAttachment{};
13 resolveAttachment.format = swapchainImageFormat;
14 resolveAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
15 resolveAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
16 resolveAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
17 resolveAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
18 resolveAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
19 resolveAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
20 resolveAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
21
22 // Subpass attachment references
23 VkAttachmentReference colorAttachmentRef{};
24 colorAttachmentRef.attachment = 0;
25 colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
26
27 VkAttachmentReference resolveAttachmentRef{};
28 resolveAttachmentRef.attachment = 1;
29 resolveAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
30
31 // Subpass description
32 VkSubpassDescription subpass{};
33 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
34 subpass.colorAttachmentCount = 1;
35 subpass.pColorAttachments = &colorAttachmentRef;
36 subpass.pResolveAttachments = &resolveAttachmentRef;
37
38 // Render pass creation
39 std::array<VkAttachmentDescription, 2> attachments = {colorAttachment, resolveAttachment};
40 VkRenderPassCreateInfo renderPassInfo{};
41 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
42 renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.size());
43 renderPassInfo.pAttachments = attachments.data();
44 renderPassInfo.subpassCount = 1;
45 renderPassInfo.pSubpasses = &subpass;
46
47 vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass);
```

This configuration ensures MSAA is performed during rendering, and the result is resolved for presentation.

20.6 Step 4: Modify the Graphics Pipeline

Update the pipeline configuration to enable MSAA by setting the rasterization and multisample state.

```
1 // Configure multisampling in the pipeline
2 VkPipelineMultisampleStateCreateInfo multisampling{};
3 multisampling.sType = VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
4 multisampling.sampleShadingEnable = VK_FALSE;
5 multisampling.rasterizationSamples = msaaSamples;
6
7 pipelineInfo.pMultisampleState = &multisampling;
```

This ensures the pipeline uses the selected MSAA sample count during rendering.

20.7 Rendering with MSAA

The MSAA process is now integrated into the Vulkan rendering loop. As you render objects, jagged edges are smoothed through multisampling, resulting in higher-quality visuals.

20.8 Summary

In this chapter, you implemented Multisample Anti-Aliasing (MSAA) in Vulkan. By using multisampled attachments and a resolve operation, you achieved smoother visuals and reduced aliasing artifacts. MSAA is a highly effective technique for improving rendering quality with minimal performance cost.

Part III

Advanced Rendering Techniques

21

Render to a Texture: Create an off-screen framebuffer

21.1 Introduction

Rendering to a texture is a versatile technique that enables advanced effects like post-processing, shadow mapping, and dynamic reflections. Instead of rendering directly to the screen, Vulkan allows you to render to an off-screen framebuffer, which can then be used as a texture in subsequent rendering passes. This chapter demonstrates how to set up an off-screen framebuffer, render a scene to it, and use the resulting texture in a second pass.

21.2 Concept Overview

In Vulkan, rendering to a texture involves creating a framebuffer with an image attachment that serves as the rendering target. The rendered output can then be sampled as a texture in shaders or copied for further processing. The main steps are:

- Create an off-screen image and image view.
- Set up a framebuffer to use the image.
- Render to the off-screen framebuffer.
- Sample the resulting texture in a second render pass.

21.3 Step 1: Create an Off-Screen Image

The off-screen image will act as the render target. It should be compatible with the desired texture format and support rendering operations.

```

1  // Create an off-screen image
2  void createOffscreenImage() {
3      VkImageCreateInfo imageInfo{};
4      imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
5      imageInfo.imageType = VK_IMAGE_TYPE_2D;
6      imageInfo.extent.width = offscreenWidth;
7      imageInfo.extent.height = offscreenHeight;
8      imageInfo.extent.depth = 1;
9      imageInfo.mipLevels = 1;
10     imageInfo.arrayLayers = 1;
11     imageInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
12     imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
13     imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
14     imageInfo.usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
15     imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
16
17     vkCreateImage(device, &imageInfo, nullptr, &offscreenImage);
18     allocateImageMemory(offscreenImage, VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, &offscreenImageMemory);
19
20     // Create an image view for the off-screen texture
21     offscreenImageView = createImageView(offscreenImage, VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT);
22 }

```

This image is the target for rendering and can later be sampled as a texture.

21.4 Step 2: Create an Off-Screen Framebuffer

Next, create a framebuffer that uses the off-screen image as its color attachment.

```

1  // Create an off-screen framebuffer
2  void createOffscreenFramebuffer() {
3      VkFramebufferCreateInfo framebufferInfo{};
4      framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
5      framebufferInfo.renderPass = offscreenRenderPass;
6      framebufferInfo.attachmentCount = 1;
7      framebufferInfo.pAttachments = &offscreenImageView;
8      framebufferInfo.width = offscreenWidth;
9      framebufferInfo.height = offscreenHeight;
10     framebufferInfo.layers = 1;
11
12     vkCreateFramebuffer(device, &framebufferInfo, nullptr, &offscreenFramebuffer);
13 }

```

The `offscreenRenderPass` is a dedicated render pass for off-screen rendering, which we will configure next.

21.5 Step 3: Set Up an Off-Screen Render Pass

Define a render pass that outputs to the off-screen image. The color attachment is configured similarly to the swapchain attachments but targets the off-screen image.

```

1 // Define the off-screen render pass
2 VkAttachmentDescription colorAttachment{};
3 colorAttachment.format = VK_FORMAT_R8G8B8A8_UNORM;
4 colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
5 colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6 colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
7 colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
8 colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
9 colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
10 colorAttachment.finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
11
12 VkAttachmentReference colorAttachmentRef{};
13 colorAttachmentRef.attachment = 0;
14 colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
15
16 VkSubpassDescription subpass{};
17 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
18 subpass.colorAttachmentCount = 1;
19 subpass.pColorAttachments = &colorAttachmentRef;
20
21 VkRenderPassCreateInfo renderPassInfo{};
22 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
23 renderPassInfo.attachmentCount = 1;
24 renderPassInfo.pAttachments = &colorAttachment;
25 renderPassInfo.subpassCount = 1;
26 renderPassInfo.pSubpasses = &subpass;
27
28 vkCreateRenderPass(device, &renderPassInfo, nullptr, &offscreenRenderPass);

```

The final layout of the attachment is set to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL`, allowing it to be sampled as a texture in subsequent operations.

21.6 Step 4: Render to the Off-Screen Framebuffer

Render to the off-screen framebuffer by binding it in the rendering loop.

```

1 // Begin the off-screen render pass
2 VkClearColorValue clearColor = {0.0f, 0.0f, 0.0f, 1.0f};
3 VkRenderPassBeginInfo renderPassInfo{};
4 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
5 renderPassInfo.renderPass = offscreenRenderPass;
6 renderPassInfo.framebuffer = offscreenFramebuffer;
7 renderPassInfo.renderArea.offset = {0, 0};
8 renderPassInfo.renderArea.extent = {offscreenWidth, offscreenHeight};
9 renderPassInfo.clearValueCount = 1;
10 renderPassInfo.pClearValues = &clearColor;
11
12 vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
13
14 // Render scene here...
15
16 vkCmdEndRenderPass(commandBuffer);

```

After this step, the off-screen image contains the rendered scene.

21.7 Step 5: Sample the Off-Screen Texture

The resulting off-screen image can now be used as a texture in a subsequent render pass. Bind it as a sampled image in a shader.

```
1 // Fragment shader to sample the texture
2 #version 450
3
4 layout(set = 0, binding = 0) uniform sampler2D offscreenTexture;
5 layout(location = 0) in vec2 fragUV;
6 layout(location = 0) out vec4 outColor;
7
8 void main() {
9     outColor = texture(offscreenTexture, fragUV);
10 }
```

Ensure that the descriptor sets are properly updated with the off-screen texture's image view and sampler.

21.8 Summary

This chapter demonstrated how to render to a texture using an off-screen framebuffer in Vulkan. This technique is fundamental for implementing many advanced rendering effects and workflows, such as post-processing and shadow mapping.

22

Implement a Frame Graph: Manage frame dependencies

22.1 Introduction

Modern graphics applications involve complex rendering pipelines with multiple passes that produce intermediate results. Managing these dependencies manually can quickly become error-prone and difficult to scale. A **frame graph** is an abstraction that automates the management of frame dependencies, ensuring that resources like images, buffers, and render passes are created, scheduled, and executed efficiently.

In this chapter, you will learn how to implement a basic frame graph for Vulkan. This includes defining resources and render passes, managing their dependencies, and ensuring that the execution order respects the dependencies.

22.2 Understanding a Frame Graph

A frame graph organizes rendering operations into a directed acyclic graph (DAG), where:

- Nodes represent rendering operations or passes (e.g., geometry pass, lighting pass).
- Edges represent dependencies between nodes, such as one pass requiring the output of another as input.

The frame graph resolves these dependencies by scheduling passes in the correct order, managing transient resources, and handling resource transitions. This improves both the maintainability and performance of your Vulkan application.

22.3 Step 1: Define Frame Graph Resources

Frame graph resources represent the inputs and outputs of render passes. These could be images, buffers, or other Vulkan objects. Start by defining a structure for resources:

```

1 // Frame graph resource structure
2 struct FrameGraphResource {
3     std::string name;
4     VkImage image;
5     VkImageView imageView;
6     VkFormat format;
7     VkExtent2D extent;
8     bool isTransient; // True for temporary resources
9 };

```

Resources can be persistent (e.g., swapchain images) or transient (e.g., intermediate images used only within a frame).

22.4 Step 2: Define Frame Graph Passes

Next, define the structure for a render pass in the frame graph. Each pass specifies its input and output resources and a callback for execution.

```

1 // Frame graph pass structure
2 struct FrameGraphPass {
3     std::string name;
4     std::vector<std::string> inputResources;
5     std::vector<std::string> outputResources;
6     std::function<void(VkCommandBuffer)> execute; // Pass execution callback
7 };

```

The `execute` function is a callback that records commands for the render pass.

22.5 Step 3: Build the Frame Graph

To create a frame graph, add resources and passes to a central structure. Resolve dependencies by analyzing resource usage.

```

1 // Frame graph structure
2 struct FrameGraph {
3     std::unordered_map<std::string, FrameGraphResource> resources;
4     std::vector<FrameGraphPass> passes;
5
6     // Add a resource to the graph
7     void addResource(const std::string& name, const FrameGraphResource& resource) {
8         resources[name] = resource;
9     }
10
11     // Add a pass to the graph
12     void addPass(const FrameGraphPass& pass) {
13         passes.push_back(pass);
14     }
15 }

```

```

16 // Execute the frame graph
17 void execute(VkCommandBuffer commandBuffer) {
18     for (const auto& pass : passes) {
19         pass.execute(commandBuffer);
20     }
21 }
22 };

```

This implementation processes passes in the order they are added. For more complex graphs, a topological sort can be performed based on dependencies.

22.6 Step 4: Create Resources and Passes

Now, define some resources and passes to populate the frame graph. For example, a basic geometry and lighting pass:

```

1 // Create resources
2 FrameGraphResource gBuffer = {
3     "GBuffer",
4     createImage(VK_FORMAT_R8G8B8A8_UNORM, {width, height}),
5     createImageView(gBufferImage, VK_FORMAT_R8G8B8A8_UNORM),
6     VK_FORMAT_R8G8B8A8_UNORM,
7     {width, height},
8     true
9 };
10
11 FrameGraphResource lightingResult = {
12     "LightingResult",
13     createImage(VK_FORMAT_R8G8B8A8_UNORM, {width, height}),
14     createImageView(lightingResultImage, VK_FORMAT_R8G8B8A8_UNORM),
15     VK_FORMAT_R8G8B8A8_UNORM,
16     {width, height},
17     true
18 };
19
20 // Add resources to the frame graph
21 frameGraph.addResource("GBuffer", gBuffer);
22 frameGraph.addResource("LightingResult", lightingResult);
23
24 // Create passes
25 FrameGraphPass geometryPass = {
26     "GeometryPass",
27     {},
28     {"GBuffer"},
29     [](VkCommandBuffer commandBuffer) {
30         // Record geometry rendering commands
31     }
32 };
33
34 FrameGraphPass lightingPass = {
35     "LightingPass",
36     {"GBuffer"},
37     {"LightingResult"},
38     [](VkCommandBuffer commandBuffer) {
39         // Record lighting commands using GBuffer as input
40     }
41 };
42
43 // Add passes to the frame graph

```



```
44 frameGraph.addPass(geometryPass);  
45 frameGraph.addPass(lightningPass);
```

These passes and resources represent a minimal example of a deferred rendering pipeline.

22.7 Step 5: Execute the Frame Graph

Execute the frame graph in the rendering loop. Resources and passes are managed automatically.

```
1 // Execute frame graph  
2 vkBeginCommandBuffer(commandBuffer, &beginInfo);  
3 frameGraph.execute(commandBuffer);  
4 vkEndCommandBuffer(commandBuffer);
```

This approach simplifies resource management and ensures that dependencies are resolved correctly.

22.8 Advantages of Using a Frame Graph

Using a frame graph offers several benefits:

- **Automatic Dependency Resolution:** The graph ensures passes are executed in the correct order.
- **Efficient Resource Management:** Transient resources are automatically created and reused as needed.
- **Scalability:** Adding new passes or resources is straightforward, even in complex pipelines.
- **Code Organization:** The frame graph provides a clear structure for rendering logic.

22.9 Summary

In this chapter, you implemented a basic frame graph to manage render passes and their dependencies. This abstraction simplifies complex pipelines, improves maintainability, and allows for efficient resource management.

23

Deferred Rendering: Render multiple passes for better lighting quality

23.1 Introduction

Deferred rendering is a popular technique in real-time graphics that separates the geometry and lighting calculations into distinct passes. This approach enables more efficient lighting computations, especially in scenes with numerous light sources. Unlike forward rendering, where lighting calculations are performed during the geometry pass, deferred rendering stores intermediate data (like positions, normals, and material properties) in a **G-buffer** for subsequent lighting calculations.

This chapter demonstrates how to implement deferred rendering in Vulkan. You will create a G-buffer, render geometry into it, and perform a lighting pass to produce the final image.

23.2 Overview of Deferred Rendering

Deferred rendering involves the following key steps:

1. **Geometry Pass:** Render the scene geometry and store per-pixel data, such as positions, normals, and albedo, in the G-buffer.
2. **Lighting Pass:** Use the G-buffer data to compute lighting for each pixel.
3. **Final Composition:** Combine the results into the final image.

This approach decouples the geometry and lighting stages, allowing for scalable lighting computations.

23.3 Step 1: Create the G-buffer

The G-buffer stores intermediate data required for lighting calculations. It typically consists of multiple images:

- **Position:** Stores per-pixel world-space positions.
- **Normal:** Stores per-pixel world-space normals.
- **Albedo:** Stores the base color of the material.

```

1 // Create G-buffer attachments
2 void createGBuffer() {
3     // Position attachment
4     gBuffer.positionImage = createImage(width, height, VK_FORMAT_R16G16B16A16_SFLOAT,
5                                         VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT);
6     gBuffer.positionImageView = createImageView(gBuffer.positionImage, VK_FORMAT_R16G16B16A16_SFLOAT,
7                                                VK_IMAGE_ASPECT_COLOR_BIT);
8
9     // Normal attachment
10    gBuffer.normalImage = createImage(width, height, VK_FORMAT_R16G16B16A16_SFLOAT,
11                                       VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT);
12    gBuffer.normalImageView = createImageView(gBuffer.normalImage, VK_FORMAT_R16G16B16A16_SFLOAT,
13                                             VK_IMAGE_ASPECT_COLOR_BIT);
14
15    // Albedo attachment
16    gBuffer.albedoImage = createImage(width, height, VK_FORMAT_R8G8B8A8_UNORM,
17                                       VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT);
18    gBuffer.albedoImageView = createImageView(gBuffer.albedoImage, VK_FORMAT_R8G8B8A8_UNORM,
19                                             VK_IMAGE_ASPECT_COLOR_BIT);
20 }

```

Each image is bound as an attachment during the geometry pass.

23.4 Step 2: Create the Geometry Pass

The geometry pass writes to the G-buffer. The render pass uses multiple color attachments, one for each G-buffer component.

```

1 // Define the geometry render pass
2 void createGeometryRenderPass() {
3     VkAttachmentDescription attachments[3];
4
5     // Position attachment
6     attachments[0].format = VK_FORMAT_R16G16B16A16_SFLOAT;
7     attachments[0].samples = VK_SAMPLE_COUNT_1_BIT;
8     attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
9     attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
10    attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
11    attachments[0].finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
12
13    // Normal attachment
14    attachments[1].format = VK_FORMAT_R16G16B16A16_SFLOAT;
15    attachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
16    attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
17    attachments[1].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
18    attachments[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
19    attachments[1].finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
20
21    // Albedo attachment
22    attachments[2].format = VK_FORMAT_R8G8B8A8_UNORM;

```

```

23 attachments[2].samples = VK_SAMPLE_COUNT_1_BIT;
24 attachments[2].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
25 attachments[2].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
26 attachments[2].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
27 attachments[2].finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
28
29 // Subpass
30 VkSubpassDescription subpass{};
31 VkAttachmentReference colorRefs[3] = {
32     {0, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL},
33     {1, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL},
34     {2, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL}
35 };
36
37 subpass.colorAttachmentCount = 3;
38 subpass.pColorAttachments = colorRefs;
39
40 // Render pass info
41 VkRenderPassCreateInfo renderPassInfo{};
42 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
43 renderPassInfo.attachmentCount = 3;
44 renderPassInfo.pAttachments = attachments;
45 renderPassInfo.subpassCount = 1;
46 renderPassInfo.pSubpasses = &subpass;
47
48 vkCreateRenderPass(device, &renderPassInfo, nullptr, &geometryRenderPass);
49 }

```

This render pass outputs to all G-buffer attachments simultaneously.

23.5 Step 3: Perform the Lighting Pass

In the lighting pass, sample the G-buffer to calculate lighting for each pixel. The fragment shader reads data from the G-buffer and applies lighting equations.

```

1 // Fragment shader for lighting pass
2 #version 450
3
4 layout(set = 0, binding = 0) uniform sampler2D gPosition;
5 layout(set = 0, binding = 1) uniform sampler2D gNormal;
6 layout(set = 0, binding = 2) uniform sampler2D gAlbedo;
7
8 layout(location = 0) in vec2 fragUV;
9 layout(location = 0) out vec4 outColor;
10
11 void main() {
12     vec3 position = texture(gPosition, fragUV).xyz;
13     vec3 normal = normalize(texture(gNormal, fragUV).xyz);
14     vec3 albedo = texture(gAlbedo, fragUV).rgb;
15
16     vec3 lightDir = normalize(vec3(0.5, 1.0, 0.7));
17     vec3 lightColor = vec3(1.0);
18
19     // Calculate diffuse lighting
20     float diffuse = max(dot(normal, lightDir), 0.0);
21     vec3 result = albedo * lightColor * diffuse;
22
23     outColor = vec4(result, 1.0);
24 }

```

The lighting pass uses a simple fullscreen quad to process each pixel of the G-buffer.

23.6 Step 4: Combine Passes in the Frame Graph

Combine the geometry and lighting passes using the frame graph introduced in the previous chapter. Ensure the G-buffer images are correctly transitioned between passes.

```

1 // Add passes to the frame graph
2 FrameGraphPass geometryPass = {
3     "GeometryPass",
4     {},
5     {"GBuffer_Position", "GBuffer_Normal", "GBuffer_Albedo"},
6     [](VkCommandBuffer commandBuffer) {
7         // Record geometry pass commands
8     }
9 };
10
11 FrameGraphPass lightingPass = {
12     "LightingPass",
13     {"GBuffer_Position", "GBuffer_Normal", "GBuffer_Albedo"},
14     {"FinalOutput"},
15     [](VkCommandBuffer commandBuffer) {
16         // Record lighting pass commands
17     }
18 };
19
20 frameGraph.addPass(geometryPass);
21 frameGraph.addPass(lightingPass);

```

The frame graph ensures that resources are correctly synchronized and transitions are applied automatically.

23.7 Summary

In this chapter, you implemented deferred rendering using Vulkan. This technique separates geometry and lighting calculations into distinct passes, enabling efficient and scalable lighting. By leveraging the G-buffer and frame graph, you created a robust pipeline for managing complex rendering workflows.

24

Add Multiple Light Sources: Implement dynamic lighting in the scene

24.1 Introduction

In many modern graphics applications, rendering realistic scenes requires support for multiple light sources. Deferred rendering is particularly well-suited for this, as the lighting pass can aggregate contributions from all light sources without recalculating the geometry. In this chapter, you will extend the lighting pass of the deferred renderer to include support for multiple dynamic light sources. This includes designing a structure for light data, passing it to the shaders, and efficiently iterating through the lights in the lighting pass.

24.2 Understanding Multiple Light Sources

Adding multiple light sources involves defining and managing the properties of each light, such as:

- **Position:** The world-space location of the light.
- **Color:** The RGB color of the light.
- **Intensity:** The strength of the light's contribution.
- **Radius:** The range within which the light affects the scene (for point lights).

The deferred renderer processes these lights during the lighting pass by iterating over the light list and aggregating their contributions for each pixel.

24.3 Step 1: Define a Light Structure

Start by defining a structure to represent individual light sources in your application. This structure will be used both on the CPU and in the shader.

```

1 // Light structure
2 struct Light {
3     glm::vec3 position; // World-space position
4     float radius;       // Radius of influence
5     glm::vec3 color;    // RGB color
6     float intensity;    // Light intensity
7 };

```

Store the lights in a list and update their properties dynamically to simulate movement or interaction.

24.4 Step 2: Pass Light Data to the Shader

Use a uniform buffer or a storage buffer to pass the list of lights to the lighting shader. A storage buffer is preferable for dynamic arrays of lights, as Vulkan allows for flexible buffer sizes with `VK_EXT_descriptor_indexing`.

```

1 // Create a buffer for light data
2 void createLightBuffer() {
3     VkDeviceSize bufferSize = sizeof(Light) * MAX_LIGHTS;
4
5     lightBuffer = createBuffer(bufferSize, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
6                                 VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT);
7
8     // Map the buffer for CPU updates
9     vkMapMemory(device, lightBuffer.memory, 0, bufferSize, 0, &mappedLightData);
10 }

```

Update the buffer each frame with the current light data:

```

1 // Update light data
2 void updateLights(const std::vector<Light>& lights) {
3     memcpy(mappedLightData, lights.data(), sizeof(Light) * lights.size());
4 }

```

Bind the buffer to the lighting pass descriptor set:

```

1 // Bind the light buffer
2 VkDescriptorBufferInfo bufferInfo{};
3 bufferInfo.buffer = lightBuffer.buffer;
4 bufferInfo.offset = 0;
5 bufferInfo.range = sizeof(Light) * MAX_LIGHTS;
6
7 VkWriteDescriptorSet descriptorWrite{};
8 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9 descriptorWrite.dstSet = descriptorSet;
10 descriptorWrite.dstBinding = 0; // Binding for light buffer
11 descriptorWrite.dstArrayElement = 0;
12 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
13 descriptorWrite.descriptorCount = 1;
14 descriptorWrite.pBufferInfo = &bufferInfo;

```

```

15
16 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);

```

24.5 Step 3: Modify the Lighting Shader

Update the lighting shader to handle multiple lights. Use a storage buffer to access the light data and iterate through the list to compute the lighting contributions.

```

1  // GLSL fragment shader for lighting
2  #version 450
3
4  layout(set = 0, binding = 0) buffer LightBuffer {
5      struct Light {
6          vec3 position;
7          float radius;
8          vec3 color;
9          float intensity;
10     } lights[];
11 };
12
13 layout(set = 0, binding = 1) uniform sampler2D gPosition;
14 layout(set = 0, binding = 2) uniform sampler2D gNormal;
15 layout(set = 0, binding = 3) uniform sampler2D gAlbedo;
16
17 layout(location = 0) in vec2 fragUV;
18 layout(location = 0) out vec4 outColor;
19
20 void main() {
21     vec3 position = texture(gPosition, fragUV).xyz;
22     vec3 normal = normalize(texture(gNormal, fragUV).xyz);
23     vec3 albedo = texture(gAlbedo, fragUV).rgb;
24
25     vec3 finalColor = vec3(0.0);
26
27     for (int i = 0; i < MAX_LIGHTS; i++) {
28         Light light = lights[i];
29
30         // Calculate light direction and attenuation
31         vec3 lightDir = light.position - position;
32         float distance = length(lightDir);
33         lightDir = normalize(lightDir);
34         float attenuation = max(0.0, 1.0 - distance / light.radius);
35
36         // Diffuse lighting
37         float diffuse = max(dot(normal, lightDir), 0.0);
38
39         // Combine light contribution
40         finalColor += albedo * light.color * light.intensity * diffuse * attenuation;
41     }
42
43     outColor = vec4(finalColor, 1.0);
44 }

```

This shader calculates the contribution of each light for every pixel and aggregates the results.

24.6 Step 4: Add Lights to the Scene

In your application, define a list of lights and update their positions or properties each frame. For example:

```

1 // Create light instances
2 std::vector<Light> lights = {
3     {{0.0f, 5.0f, 0.0f}, 10.0f, {1.0f, 0.8f, 0.6f}, 1.5f},
4     {{-5.0f, 3.0f, -2.0f}, 8.0f, {0.6f, 1.0f, 0.8f}, 1.0f},
5     {{2.0f, 4.0f, 3.0f}, 7.0f, {0.8f, 0.6f, 1.0f}, 1.2f}
6 };
7
8 // Update light buffer each frame
9 updateLights(lights);

```

This allows you to dynamically add, remove, or move lights in the scene.

24.7 Performance Optimization

Rendering multiple lights can be computationally expensive. Consider the following optimizations:

- **Culling**: Exclude lights that do not affect the current screen region.
- **Tiled Deferred Rendering**: Divide the screen into tiles and process lights only for tiles they overlap.
- **Clustered Lighting**: Extend tiled rendering to handle volumetric light clusters.

24.8 Summary

In this chapter, you extended the deferred renderer to support multiple light sources. By leveraging Vulkan's flexible descriptor sets and storage buffers, you implemented dynamic lighting that can scale to a large number of lights. This feature significantly enhances the visual realism of your scenes.

25

Physically Based Rendering (PBR): Start with simple PBR shaders

25.1 Introduction

Physically Based Rendering (PBR) aims to simulate the interaction of light with materials in a way that is consistent with the laws of physics. PBR enables more realistic and visually compelling images by using models that approximate real-world materials. In this chapter, you will implement simple PBR shaders using the metallic-roughness workflow. This method models materials with two main properties: **metalness** and **roughness**, along with albedo (base color).

By the end of this chapter, you will have a working implementation of a PBR shader that combines these properties to calculate realistic lighting effects using Vulkan.

25.2 Overview of PBR Concepts

The metallic-roughness workflow uses the following key parameters:

- **Albedo**: The base color of the material.
- **Metalness**: A factor (0 to 1) determining if a surface behaves like a metal (1) or a non-metal/dielectric (0).
- **Roughness**: A factor (0 to 1) that determines how smooth (0) or rough (1) the surface is. Smooth surfaces produce sharp reflections, while rough surfaces scatter light.
- **Normal**: The direction perpendicular to the surface, influencing how light reflects.
- **Ambient Occlusion** (optional): A factor that darkens areas occluded from ambient light.

PBR shaders use the microfacet BRDF (Bidirectional Reflectance Distribution Function) to calculate light interaction with these properties.

25.3 Step 1: Extend the G-buffer for PBR

To support PBR, add extra components to your G-buffer for metalness, roughness, and ambient occlusion. Update the G-buffer creation process:

```

1 // Extend G-buffer for PBR
2 void createGBuffer() {
3     // Existing attachments (Position, Normal, Albedo)
4     // ...
5
6     // Metallic-Roughness-Ambient Occlusion attachment
7     gBuffer.metallicRoughnessAOImage = createImage(width, height, VK_FORMAT_R8G8B8A8_UNORM,
8                                                     VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT |
9                                                     VK_IMAGE_USAGE_SAMPLED_BIT);
10    gBuffer.metallicRoughnessAOImageView = createImageView(gBuffer.metallicRoughnessAOImage,
11                                                            VK_FORMAT_R8G8B8A8_UNORM, VK_IMAGE_ASPECT_COLOR_BIT);
12 }

```

This attachment will store:

- **R (red)**: Metalness
- **G (green)**: Roughness
- **B (blue)**: Ambient Occlusion (optional)

25.4 Step 2: Update Geometry Pass Shaders

Modify the geometry pass shaders to output metalness, roughness, and ambient occlusion into the extended G-buffer.

```

1 // GLSL Fragment Shader - Geometry Pass
2 #version 450
3
4 layout(location = 0) in vec3 fragPosition;
5 layout(location = 1) in vec3 fragNormal;
6 layout(location = 2) in vec2 fragUV;
7
8 layout(set = 0, binding = 0) uniform sampler2D albedoMap;
9 layout(set = 0, binding = 1) uniform sampler2D metalnessMap;
10 layout(set = 0, binding = 2) uniform sampler2D roughnessMap;
11 layout(set = 0, binding = 3) uniform sampler2D aoMap;
12
13 layout(location = 0) out vec4 outPosition;
14 layout(location = 1) out vec4 outNormal;
15 layout(location = 2) out vec4 outAlbedo;
16 layout(location = 3) out vec4 outMetallicRoughnessAO;
17
18 void main() {
19     outPosition = vec4(fragPosition, 1.0);
20     outNormal = vec4(normalize(fragNormal), 0.0);
21     outAlbedo = texture(albedoMap, fragUV);
22
23     float metalness = texture(metalnessMap, fragUV).r;

```

```

24     float roughness = texture(roughnessMap, fragUV).r;
25     float ao = texture(aoMap, fragUV).r;
26
27     outMetallicRoughnessAO = vec4(metalness, roughness, ao, 0.0);
28 }

```

This shader writes PBR-specific data into the G-buffer.

25.5 Step 3: Implement the PBR Lighting Shader

The lighting shader calculates light contributions using the PBR equations. Implement the following key calculations:

- **Fresnel**: Determines reflectance at grazing angles.
- **Normal Distribution Function (NDF)**: Models the surface's microfacet distribution.
- **Geometry Function**: Accounts for occlusion and shadowing effects between microfacets.

```

1  // GLSL Fragment Shader - Lighting Pass
2  #version 450
3
4  layout(set = 0, binding = 0) uniform sampler2D gPosition;
5  layout(set = 0, binding = 1) uniform sampler2D gNormal;
6  layout(set = 0, binding = 2) uniform sampler2D gAlbedo;
7  layout(set = 0, binding = 3) uniform sampler2D gMetallicRoughnessAO;
8
9  layout(set = 0, binding = 4) buffer LightBuffer {
10     struct Light {
11         vec3 position;
12         vec3 color;
13         float intensity;
14     } lights[];
15 };
16
17 layout(location = 0) in vec2 fragUV;
18 layout(location = 0) out vec4 outColor;
19
20 // Fresnel equation
21 vec3 fresnelSchlick(float cosTheta, vec3 F0) {
22     return F0 + (1.0 - F0) * pow(1.0 - cosTheta, 5.0);
23 }
24
25 // Geometry function (Smith's Schlick-GGX)
26 float geometrySchlickGGX(float NdotV, float roughness) {
27     float k = pow(roughness + 1.0, 2.0) / 8.0;
28     return NdotV / (NdotV * (1.0 - k) + k);
29 }
30
31 // Normal Distribution Function (NDF - GGX)
32 float distributionGGX(vec3 N, vec3 H, float roughness) {
33     float a = roughness * roughness;
34     float a2 = a * a;
35     float NdotH = max(dot(N, H), 0.0);
36     float denom = (NdotH * a2 - NdotH) * NdotH + 1.0;
37     return a2 / (PI * denom * denom);
38 }
39
40 void main() {
41     vec3 position = texture(gPosition, fragUV).xyz;
42     vec3 normal = normalize(texture(gNormal, fragUV).xyz);
43     vec3 albedo = texture(gAlbedo, fragUV).rgb;

```



```

44     vec4 mra = texture(gMetallicRoughnessAO, fragUV);
45
46     float metalness = mra.r;
47     float roughness = mra.g;
48     float ao = mra.b;
49
50     vec3 F0 = mix(vec3(0.04), albedo, metalness);
51     vec3 N = normalize(normal);
52     vec3 V = normalize(cameraPos - position);
53     vec3 Lo = vec3(0.0);
54
55     for (int i = 0; i < lightCount; i++) {
56         Light light = lights[i];
57         vec3 L = normalize(light.position - position);
58         vec3 H = normalize(V + L);
59
60         float distance = length(light.position - position);
61         float attenuation = 1.0 / (distance * distance);
62         vec3 radiance = light.color * light.intensity * attenuation;
63
64         float NDF = distributionGGX(N, H, roughness);
65         float G = geometrySchlickGGX(max(dot(N, V), 0.0), roughness) *
66             geometrySchlickGGX(max(dot(N, L), 0.0), roughness);
67         vec3 F = fresnelSchlick(max(dot(H, V), 0.0), F0);
68
69         vec3 numerator = NDF * G * F;
70         float denominator = 4.0 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
71         vec3 specular = numerator / denominator;
72
73         vec3 kS = F;
74         vec3 kD = vec3(1.0) - kS;
75         kD *= 1.0 - metalness;
76
77         float NdotL = max(dot(N, L), 0.0);
78         Lo += (kD * albedo / PI + specular) * radiance * NdotL;
79     }
80
81     vec3 ambient = vec3(0.03) * albedo * ao;
82     vec3 color = ambient + Lo;
83
84     outColor = vec4(color, 1.0);
85 }

```

25.6 Step 4: Integrate and Test

Add the PBR shader to your Vulkan pipeline and test it with a range of materials and light configurations. Use textures for metalness and roughness to achieve diverse visual effects.

25.7 Summary

In this chapter, you implemented a basic PBR shader using the metallic-roughness workflow. This shader models realistic material behavior, significantly enhancing the visual quality of rendered scenes.

26

Implement Shadow Mapping: Add basic shadow rendering

26.1 Introduction

Shadows add depth and realism to rendered scenes by simulating light occlusion. Shadow mapping is a widely used technique in graphics programming to generate shadows by rendering a depth map from the perspective of the light source. This chapter will guide you through implementing basic shadow mapping in Vulkan, including creating a depth map, sampling it in the lighting pass, and combining shadows with lighting.

26.2 Overview of Shadow Mapping

Shadow mapping works as follows:

1. Render the scene from the light's perspective to a depth texture, creating a **shadow map**.
2. In the lighting pass, sample the shadow map to determine whether a fragment is in shadow.
3. Modulate the light's contribution based on the shadow status.

26.3 Step 1: Create a Shadow Map Render Pass

The shadow map is a depth-only texture rendered from the light's perspective. Start by creating a shadow map render pass.

```

1 // Create shadow map render pass
2 VkAttachmentDescription depthAttachment{};
3 depthAttachment.format = findDepthFormat();
4 depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
5 depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6 depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
7 depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
8 depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
9 depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
10 depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL;
11
12 VkAttachmentReference depthAttachmentRef{};
13 depthAttachmentRef.attachment = 0;
14 depthAttachmentRef.layout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
15
16 VkSubpassDescription subpass{};
17 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
18 subpass.pDepthStencilAttachment = &depthAttachmentRef;
19
20 VkRenderPassCreateInfo renderPassInfo{};
21 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
22 renderPassInfo.attachmentCount = 1;
23 renderPassInfo.pAttachments = &depthAttachment;
24 renderPassInfo.subpassCount = 1;
25 renderPassInfo.pSubpasses = &subpass;
26
27 vkCreateRenderPass(device, &renderPassInfo, nullptr, &shadowRenderPass);

```

This render pass is optimized for writing depth values to the shadow map.

26.4 Step 2: Create the Shadow Map Framebuffer

Allocate a depth texture to store the shadow map and create a framebuffer for the shadow render pass.

```

1 // Create depth texture for shadow map
2 VkImageCreateInfo imageInfo{};
3 imageInfo.imageType = VK_IMAGE_TYPE_2D;
4 imageInfo.extent.width = shadowMapWidth;
5 imageInfo.extent.height = shadowMapHeight;
6 imageInfo.extent.depth = 1;
7 imageInfo.mipLevels = 1;
8 imageInfo.arrayLayers = 1;
9 imageInfo.format = findDepthFormat();
10 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
11 imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
12 imageInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
13 imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
14
15 vkCreateImage(device, &imageInfo, nullptr, &shadowMapImage);
16
17 // Create framebuffer
18 VkFramebufferCreateInfo framebufferInfo{};
19 framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
20 framebufferInfo.renderPass = shadowRenderPass;
21 framebufferInfo.attachmentCount = 1;
22 framebufferInfo.pAttachments = &shadowMapImageView;
23 framebufferInfo.width = shadowMapWidth;
24 framebufferInfo.height = shadowMapHeight;
25 framebufferInfo.layers = 1;
26
27 vkCreateFramebuffer(device, &framebufferInfo, nullptr, &shadowFramebuffer);

```

26.5 Step 3: Render the Scene to the Shadow Map

Use a lightweight shader to render the scene's depth from the light's perspective.

```

1  // GLSL Vertex Shader - Shadow Map
2  #version 450
3
4  layout(location = 0) in vec3 inPosition;
5
6  layout(push_constant) uniform LightViewProjection {
7      mat4 lightViewProjection;
8  } pc;
9
10 void main() {
11     gl_Position = pc.lightViewProjection * vec4(inPosition, 1.0);
12 }
```

The vertex shader projects the scene geometry into light-space coordinates to produce the depth map.

Set up a pipeline for the shadow map pass, binding this shader and disabling color writes.

```

1  // Configure shadow map pipeline
2  VkPipelineDepthStencilStateCreateInfo depthStencil{};
3  depthStencil.depthTestEnable = VK_TRUE;
4  depthStencil.depthWriteEnable = VK_TRUE;
5  depthStencil.depthCompareOp = VK_COMPARE_OP_LESS;
6  depthStencil.stencilTestEnable = VK_FALSE;
7
8  // Disable color output
9  VkPipelineColorBlendStateCreateInfo colorBlendState{};
10 colorBlendState.attachmentCount = 0;
```

26.6 Step 4: Sample the Shadow Map in the Lighting Pass

In the lighting shader, use the shadow map to determine whether a fragment is in shadow. Perform the following checks:

1. Transform the fragment into light space.
2. Compare the fragment's depth with the value in the shadow map.

```

1  // GLSL Fragment Shader - Lighting with Shadows
2  #version 450
3
4  layout(set = 0, binding = 4) uniform sampler2D shadowMap;
5
6  vec3 calculateShadows(vec3 position, vec3 normal, vec3 lightPos) {
7      vec4 lightSpacePos = lightViewProjection * vec4(position, 1.0);
8      vec3 shadowCoord = lightSpacePos.xyz / lightSpacePos.w;
9
10     // Sample shadow map
11     float shadowDepth = texture(shadowMap, shadowCoord.xy).r;
12     float currentDepth = shadowCoord.z;
13
14     // Shadow comparison with bias to avoid artifacts
15     float shadow = currentDepth - 0.005 > shadowDepth ? 0.0 : 1.0;
16 }
```

```

17     return shadow;
18 }
19
20 void main() {
21     vec3 position = texture(gPosition, fragUV).xyz;
22     vec3 normal = normalize(texture(gNormal, fragUV).xyz);
23
24     vec3 lightPos = lights[0].position;
25     vec3 lightColor = lights[0].color;
26
27     vec3 shadow = calculateShadows(position, normal, lightPos);
28
29     // Combine lighting and shadows
30     vec3 finalColor = shadow * lightColor * max(dot(normal, normalize(lightPos - position)), 0.0);
31     outColor = vec4(finalColor, 1.0);
32 }

```

26.7 Step 5: Handle Shadow Artifacts

Shadow mapping introduces common artifacts such as **shadow acne** and **peter panning**. Address these by:

- Adding a depth bias to the shadow comparison.
- Using PCF (Percentage Closer Filtering) to smooth shadows by averaging neighboring samples.

26.8 Step 6: Integrate and Test

Add the shadow map pipeline to your rendering loop. Render the shadow map before the lighting pass and ensure the shadow map is available as an input texture in the lighting pass.

```

1 // Render shadow map
2 beginRenderPass(shadowCommandBuffer, shadowRenderPass, shadowFramebuffer);
3 bindShadowPipeline(shadowPipeline);
4 updatePushConstants(lightViewProjectionMatrix);
5 drawScene();
6 endRenderPass();

```

26.9 Summary

In this chapter, you implemented basic shadow mapping using Vulkan. By rendering a depth map from the light's perspective and sampling it during the lighting pass, you created realistic shadows in your scene.

27

Omnidirectional Shadows: Implement cube map shadows for point lights

27.1 Introduction

Omnidirectional shadows allow point lights to cast shadows in all directions, creating a realistic effect in 3D scenes. This is achieved by rendering a shadow map in the form of a cube map, where each face represents the light's view in a specific direction. In this chapter, you will implement omnidirectional shadows using Vulkan, leveraging cube maps and multiple render passes.

27.2 Overview of Omnidirectional Shadow Mapping

Omnidirectional shadow mapping involves:

1. Rendering the scene six times, once for each face of the cube map, from the perspective of the point light.
2. Sampling the cube map during the lighting pass to determine shadowed fragments.

The technique requires efficient use of Vulkan's features, such as cube map textures, frame-buffers, and dynamic uniform updates.

27.3 Step 1: Create a Cube Map for Shadows

Create a cube map texture to store the depth information for all six faces.


```

1 // Create cube map image for shadow map
2 VkImageCreateInfo imageInfo{};
3 imageInfo.imageType = VK_IMAGE_TYPE_2D;
4 imageInfo.extent.width = shadowMapResolution;
5 imageInfo.extent.height = shadowMapResolution;
6 imageInfo.extent.depth = 1;
7 imageInfo.mipLevels = 1;
8 imageInfo.arrayLayers = 6; // Six faces for cube map
9 imageInfo.format = findDepthFormat();
10 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
11 imageInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
12 imageInfo.flags = VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT;
13
14 vkCreateImage(device, &imageInfo, nullptr, &shadowCubeMapImage);

```

Create an image view for the cube map with the `VK_IMAGE_VIEW_TYPE_CUBE` type.

```

1 // Create image view for cube map
2 VkImageViewCreateInfo viewInfo{};
3 viewInfo.image = shadowCubeMapImage;
4 viewInfo.viewType = VK_IMAGE_VIEW_TYPE_CUBE;
5 viewInfo.format = findDepthFormat();
6 viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
7 viewInfo.subresourceRange.baseMipLevel = 0;
8 viewInfo.subresourceRange.levelCount = 1;
9 viewInfo.subresourceRange.baseArrayLayer = 0;
10 viewInfo.subresourceRange.layerCount = 6;
11
12 vkCreateImageView(device, &viewInfo, nullptr, &shadowCubeMapView);

```

27.4 Step 2: Set Up a Shadow Map Render Pass and Framebuffer

Create a render pass for depth-only rendering, as described in the previous chapter. Then, create framebuffers for each face of the cube map.

```

1 // Create framebuffer for each cube map face
2 VkFramebufferCreateInfo framebufferInfo{};
3 framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
4 framebufferInfo.renderPass = shadowRenderPass;
5 framebufferInfo.attachmentCount = 1;
6 framebufferInfo.pAttachments = &shadowCubeMapView;
7 framebufferInfo.width = shadowMapResolution;
8 framebufferInfo.height = shadowMapResolution;
9 framebufferInfo.layers = 1;
10
11 std::vector<VkFramebuffer> shadowFramebuffers(6);
12 for (int i = 0; i < 6; ++i) {
13     framebufferInfo.pAttachments = &shadowCubeMapView[i];
14     vkCreateFramebuffer(device, &framebufferInfo, nullptr, &shadowFramebuffers[i]);
15 }

```

27.5 Step 3: Render the Scene to Each Cube Map Face

For each face of the cube map, render the scene from the light's perspective. Define the view and projection matrices for all six directions (positive and negative X, Y, Z).

```

1 // Define light's view matrices for cube map faces
2 std::vector<glm::mat4> lightViews = {
3     glm::lookAt(lightPosition, lightPosition + glm::vec3(1, 0, 0), glm::vec3(0, -1, 0)), // +X
4     glm::lookAt(lightPosition, lightPosition + glm::vec3(-1, 0, 0), glm::vec3(0, -1, 0)), // -X
5     glm::lookAt(lightPosition, lightPosition + glm::vec3(0, 1, 0), glm::vec3(0, 0, 1)), // +Y
6     glm::lookAt(lightPosition, lightPosition + glm::vec3(0, -1, 0), glm::vec3(0, 0, -1)), // -Y
7     glm::lookAt(lightPosition, lightPosition + glm::vec3(0, 0, 1), glm::vec3(0, -1, 0)), // +Z
8     glm::lookAt(lightPosition, lightPosition + glm::vec3(0, 0, -1), glm::vec3(0, -1, 0)) // -Z
9 };
10
11 // Perspective projection for the light
12 glm::mat4 lightProjection = glm::perspective(glm::radians(90.0f), 1.0f, nearPlane, farPlane);
13
14 // Render each face
15 for (int face = 0; face < 6; ++face) {
16     beginRenderPass(commandBuffer, shadowRenderPass, shadowFramebuffers[face]);
17     updatePushConstants(lightProjection * lightViews[face]);
18     drawScene();
19     endRenderPass(commandBuffer);
20 }

```

27.6 Step 4: Sample the Cube Map in the Lighting Pass

In the lighting shader, use a cube sampler to determine whether a fragment is in shadow.

```

1 // GLSL Fragment Shader - Lighting with Omnidirectional Shadows
2 #version 450
3
4 layout(set = 0, binding = 4) uniform samplerCube shadowCubeMap;
5
6 float calculateOmnidirectionalShadow(vec3 fragPos, vec3 lightPos) {
7     vec3 lightToFragment = fragPos - lightPos;
8     float currentDepth = length(lightToFragment);
9
10    // Sample cube map
11    float closestDepth = texture(shadowCubeMap, lightToFragment).r * farPlane;
12
13    // Compare depths
14    float shadow = currentDepth - 0.005 > closestDepth ? 0.0 : 1.0;
15    return shadow;
16 }
17
18 void main() {
19     vec3 fragPos = texture(gPosition, fragUV).xyz;
20     vec3 normal = normalize(texture(gNormal, fragUV).xyz);
21
22     vec3 lightPos = lights[0].position;
23     vec3 lightColor = lights[0].color;
24
25     float shadow = calculateOmnidirectionalShadow(fragPos, lightPos);
26     vec3 lighting = shadow * lightColor * max(dot(normal, normalize(lightPos - fragPos)), 0.0);
27
28     outColor = vec4(lighting, 1.0);
29 }

```

27.7 Step 5: Optimize and Test

Omnidirectional shadow mapping is computationally expensive. Optimize it by:

- Reducing shadow map resolution.
- Using cascaded rendering for larger scenes.
- Implementing PCF or VSM for smoother shadows.

Integrate the shadow map into your lighting pipeline and test the effect in various scenes with dynamic point lights.

27.8 Summary

In this chapter, you implemented omnidirectional shadows using Vulkan, allowing point lights to cast realistic shadows in all directions. This was achieved by rendering a cube map as a shadow map and sampling it in the lighting pass.

28

Create a Skybox: Render a cubemap as the background

28.1 Introduction

A skybox is a large cube surrounding the scene, textured with a cubemap to simulate an infinite environment such as a sky, mountains, or space. This chapter will guide you through rendering a skybox in Vulkan using a cubemap texture, providing a visually appealing background for your 3D scenes.

28.2 Overview of Skybox Rendering

Skybox rendering involves:

1. Creating and loading a cubemap texture.
2. Rendering the cubemap in a pass that ensures it appears in the background.
3. Disabling depth writes so the skybox doesn't interfere with other objects in the scene.

28.3 Step 1: Load the Cubemap Texture

A cubemap texture contains six images representing the six faces of a cube. These images must be loaded and combined into a Vulkan `VkImage`.

```
1 // Load images for cubemap faces
2 std::vector<std::string> cubemapFaces = {
```

```

3     "right.jpg", "left.jpg", "top.jpg",
4     "bottom.jpg", "front.jpg", "back.jpg"
5 };
6
7 // Create cubemap image
8 VkImageCreateInfo imageInfo{};
9 imageInfo.imageType = VK_IMAGE_TYPE_2D;
10 imageInfo.extent.width = cubemapWidth;
11 imageInfo.extent.height = cubemapHeight;
12 imageInfo.extent.depth = 1;
13 imageInfo.mipLevels = 1;
14 imageInfo.arrayLayers = 6; // Six faces
15 imageInfo.format = VK_FORMAT_R8G8B8A8_SRGB;
16 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
17 imageInfo.usage = VK_IMAGE_USAGE_SAMPLED_BIT | VK_IMAGE_USAGE_TRANSFER_DST_BIT;
18 imageInfo.flags = VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT;
19
20 vkCreateImage(device, &imageInfo, nullptr, &cubemapImage);

```

Load the image data into the cubemap and transition its layout to `VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL` for use in shaders.

28.4 Step 2: Create the Skybox Pipeline

To render the skybox, set up a pipeline with the following characteristics:

- Use a vertex and fragment shader specifically designed for cubemap sampling.
- Enable depth testing but disable depth writes to ensure the skybox is always in the background.
- Bind the cubemap as a shader resource.

```

1 // Create depth and rasterization states
2 VkPipelineDepthStencilStateCreateInfo depthStencil{};
3 depthStencil.depthTestEnable = VK_TRUE;
4 depthStencil.depthWriteEnable = VK_FALSE; // Disable depth writes
5 depthStencil.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
6
7 VkPipelineRasterizationStateCreateInfo rasterizer{};
8 rasterizer.cullMode = VK_CULL_MODE_FRONT_BIT; // Cull front faces
9 rasterizer.frontFace = VK_FRONT_FACE_CLOCKWISE;

```

28.5 Step 3: Write the Skybox Shaders

The vertex shader transforms the cube vertices into clip space, and the fragment shader samples the cubemap to produce the skybox color.

```

1 // GLSL Vertex Shader - Skybox
2 #version 450
3
4 layout(location = 0) in vec3 inPosition;
5 layout(push_constant) uniform PushConstants {
6     mat4 viewProjection;
7 } pc;
8
9 out vec3 texCoords;

```

```

10
11 void main() {
12     texCoords = inPosition;
13     gl_Position = (pc.viewProjection * vec4(inPosition, 1.0)).xyzw; // Infinite depth
14 }

1 // GLSL Fragment Shader - Skybox
2 #version 450
3
4 layout(set = 0, binding = 0) uniform samplerCube skybox;
5
6 in vec3 texCoords;
7 out vec4 outColor;
8
9 void main() {
10     outColor = texture(skybox, texCoords);
11 }

```

28.6 Step 4: Render the Skybox

In your rendering loop, render the skybox before other objects to ensure it serves as the background.

```

1 // Bind skybox pipeline and descriptor set
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, skyboxPipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, skyboxPipelineLayout, 0, 1, &
    ↳ skyboxDescriptorSet, 0, nullptr);
4
5 // Push constants for view-projection matrix (without translation)
6 glm::mat4 viewProjection = projectionMatrix * glm::mat4(glm::mat3(viewMatrix));
7 vkCmdPushConstants(commandBuffer, skyboxPipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, sizeof(glm::mat4), &
    ↳ viewProjection);
8
9 // Draw the skybox
10 vkCmdDraw(commandBuffer, 36, 1, 0, 0);

```

The view matrix is stripped of its translation component to keep the skybox centered around the camera.

28.7 Step 5: Optimize the Skybox Rendering

To optimize skybox rendering:

- Use lower-resolution cubemaps to save memory.
- Compress cubemap textures with formats like BC6 or BC7 for HDR content.
- Generate mipmaps for smoother transitions at a distance.

28.8 Summary

In this chapter, you implemented a skybox using Vulkan, enhancing the visual appeal of your 3D scenes with a realistic background. By leveraging cubemap textures and a specialized rendering pipeline, you created an efficient and visually striking backdrop.

29

Add Post-Processing Effects: Add effects like bloom or vignette

29.1 Introduction

Post-processing effects enhance the visual quality of your rendered scenes, adding realism or stylistic flair. Effects like bloom and vignette are applied after rendering the main scene, making them flexible for a variety of artistic goals. In this chapter, you will learn how to implement post-processing effects in Vulkan by rendering to an off-screen framebuffer and processing the results in a separate rendering pass.

29.2 Overview of Post-Processing in Vulkan

Post-processing involves the following steps:

1. Render the scene to an off-screen framebuffer.
2. Pass the framebuffer image to a full-screen quad.
3. Apply shaders that modify the image to achieve the desired effect.

This approach decouples the scene rendering from image processing, enabling a wide range of effects.

29.3 Step 1: Create an Off-Screen Framebuffer

Set up an off-screen framebuffer to capture the rendered scene. Use a color attachment with the desired format and resolution.

```

1 // Create color attachment for off-screen framebuffer
2 VkImageCreateInfo colorImageInfo{};
3 colorImageInfo.imageType = VK_IMAGE_TYPE_2D;
4 colorImageInfo.extent.width = swapchainExtent.width;
5 colorImageInfo.extent.height = swapchainExtent.height;
6 colorImageInfo.extent.depth = 1;
7 colorImageInfo.mipLevels = 1;
8 colorImageInfo.arrayLayers = 1;
9 colorImageInfo.format = swapchainImageFormat;
10 colorImageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
11 colorImageInfo.usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
12
13 vkCreateImage(device, &colorImageInfo, nullptr, &offscreenColorImage);

```

Create an image view and framebuffer for this attachment.

```

1 // Create image view
2 VkImageViewCreateInfo viewInfo{};
3 viewInfo.image = offscreenColorImage;
4 viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
5 viewInfo.format = swapchainImageFormat;
6 viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
7 viewInfo.subresourceRange.baseMipLevel = 0;
8 viewInfo.subresourceRange.levelCount = 1;
9 viewInfo.subresourceRange.baseArrayLayer = 0;
10 viewInfo.subresourceRange.layerCount = 1;
11
12 vkCreateImageView(device, &viewInfo, nullptr, &offscreenColorImageView);
13
14 // Create framebuffer
15 VkFramebufferCreateInfo framebufferInfo{};
16 framebufferInfo.renderPass = offscreenRenderPass;
17 framebufferInfo.attachmentCount = 1;
18 framebufferInfo.pAttachments = &offscreenColorImageView;
19 framebufferInfo.width = swapchainExtent.width;
20 framebufferInfo.height = swapchainExtent.height;
21 framebufferInfo.layers = 1;
22
23 vkCreateFramebuffer(device, &framebufferInfo, nullptr, &offscreenFramebuffer);

```

29.4 Step 2: Render to the Off-Screen Framebuffer

In the primary rendering pass, bind the off-screen framebuffer as the render target instead of the swapchain.

```

1 // Begin rendering to off-screen framebuffer
2 VkRenderPassBeginInfo renderPassInfo{};
3 renderPassInfo.renderPass = offscreenRenderPass;
4 renderPassInfo.framebuffer = offscreenFramebuffer;
5 renderPassInfo.renderArea.extent = swapchainExtent;
6 renderPassInfo.clearValueCount = 1;
7 renderPassInfo.pClearValues = &clearValue;
8
9 vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);

```

```

10
11 // Render the scene here
12 drawScene();
13
14 vkCmdEndRenderPass(commandBuffer);

```

29.5 Step 3: Set Up Post-Processing Pipeline

Create a pipeline for post-processing effects. This involves rendering a full-screen quad and applying a fragment shader that processes the off-screen image.

```

1 // Create fullscreen quad pipeline
2 VkPipelineRasterizationStateCreateInfo rasterizer{};
3 rasterizer.cullMode = VK_CULL_MODE_NONE;
4
5 VkPipelineDepthStencilStateCreateInfo depthStencil{};
6 depthStencil.depthTestEnable = VK_FALSE;
7 depthStencil.depthWriteEnable = VK_FALSE;
8
9 VkPipelineShaderStageCreateInfo vertShaderStage = loadShader("postprocess.vert.spv", VK_SHADER_STAGE_VERTEX_BIT
    ↳ );
10 VkPipelineShaderStageCreateInfo fragShaderStage = loadShader("postprocess.frag.spv",
    ↳ VK_SHADER_STAGE_FRAGMENT_BIT);
11
12 VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStage, fragShaderStage };
13
14 // Configure pipeline for fullscreen quad rendering
15 VkGraphicsPipelineCreateInfo pipelineInfo{};
16 pipelineInfo.stageCount = 2;
17 pipelineInfo.pStages = shaderStages;
18 // Attach other states such as rasterizer, depthStencil, etc.
19 vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &postProcessPipeline);

```

29.6 Step 4: Write Post-Processing Shaders

Write a fragment shader to implement a bloom or vignette effect. For bloom, highlight bright regions and blur them. For vignette, darken the edges of the screen.

```

1 // GLSL Fragment Shader - Bloom
2 #version 450
3
4 layout(set = 0, binding = 0) uniform sampler2D sceneTexture;
5 layout(location = 0) in vec2 fragUV;
6 layout(location = 0) out vec4 outColor;
7
8 void main() {
9     vec4 color = texture(sceneTexture, fragUV);
10    float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722)); // Luminance
11    vec4 bloom = brightness > 1.0 ? color * 0.5 : vec4(0.0); // Simple bloom
12    outColor = color + bloom;
13 }

```

29.7 Step 5: Render the Post-Processed Image

Render the full-screen quad with the processed image.

```

1  // Begin post-processing render pass
2  VkRenderPassBeginInfo postRenderPassInfo{};
3  postRenderPassInfo.renderPass = postProcessRenderPass;
4  postRenderPassInfo.framebuffer = swapchainFramebuffers[currentFrame];
5  postRenderPassInfo.renderArea.extent = swapchainExtent;
6  postRenderPassInfo.clearValueCount = 1;
7  postRenderPassInfo.pClearValues = &clearValue;
8
9  vkCmdBeginRenderPass(commandBuffer, &postRenderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
10
11 // Bind post-processing pipeline and descriptor set
12 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, postProcessPipeline);
13 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, postProcessPipelineLayout, 0, 1, &
    ↳ postProcessDescriptorSet, 0, nullptr);
14
15 // Draw full-screen quad
16 vkCmdDraw(commandBuffer, 6, 1, 0, 0);
17
18 vkCmdEndRenderPass(commandBuffer);

```

29.8 Summary

In this chapter, you implemented a post-processing pipeline using Vulkan, rendering a full-screen quad to apply effects like bloom or vignette. These techniques open the door to advanced visual enhancements, including HDR, tone mapping, and screen-space ambient occlusion.

30

Create a Particle System: Simulate particles using a vertex shader

30.1 Introduction

Particle systems are used to simulate dynamic effects such as smoke, fire, explosions, and magic spells. By leveraging the GPU for computation, Vulkan allows you to simulate and render thousands of particles efficiently. In this chapter, you will learn to create a basic particle system using a vertex shader to handle particle behavior and movement.

30.2 Overview of Particle Systems in Vulkan

A particle system typically involves:

1. Storing particle properties such as position, velocity, and color.
2. Updating particle positions and other attributes in the vertex shader.
3. Rendering particles as points or quads.

Using the vertex shader simplifies the simulation by offloading particle updates directly to the GPU.

30.3 Step 1: Define the Particle Structure

Define a structure to represent each particle and initialize a buffer to store particle data.


```

1 // Particle structure
2 struct Particle {
3     glm::vec2 position;
4     glm::vec2 velocity;
5     glm::vec4 color;
6     float lifetime;
7     float age;
8 };
9
10 // Initialize particle buffer
11 std::vector<Particle> particles(maxParticles);
12 for (auto& particle : particles) {
13     particle.position = glm::vec2(0.0f);
14     particle.velocity = glm::vec2(
15         random(-1.0f, 1.0f),
16         random(-1.0f, 1.0f)
17     ) * initialSpeed;
18     particle.color = glm::vec4(1.0f, 1.0f, 1.0f, 1.0f);
19     particle.lifetime = random(1.0f, 3.0f);
20     particle.age = 0.0f;
21 }
22
23 // Create Vulkan buffer for particles
24 VkBufferCreateInfo bufferInfo{};
25 bufferInfo.size = sizeof(Particle) * maxParticles;
26 bufferInfo.usage = VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
27
28 vkCreateBuffer(device, &bufferInfo, nullptr, &particleBuffer);

```

Upload the initialized particle data to the GPU.

30.4 Step 2: Set Up the Graphics Pipeline

Create a graphics pipeline to render the particles. Use a vertex shader for particle movement and a fragment shader to color the particles.

```

1 // Configure pipeline for particle rendering
2 VkPipelineRasterizationStateCreateInfo rasterizer{};
3 rasterizer.polygonMode = VK_POLYGON_MODE_FILL;
4 rasterizer.cullMode = VK_CULL_MODE_NONE;
5
6 VkPipelineInputAssemblyStateCreateInfo inputAssembly{};
7 inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
8
9 // Load shaders
10 VkPipelineShaderStageCreateInfo vertShaderStage = loadShader("particle.vert.spv", VK_SHADER_STAGE_VERTEX_BIT);
11 VkPipelineShaderStageCreateInfo fragShaderStage = loadShader("particle.frag.spv", VK_SHADER_STAGE_FRAGMENT_BIT);
12
13 VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStage, fragShaderStage };
14
15 // Create the graphics pipeline
16 VkGraphicsPipelineCreateInfo pipelineInfo{};
17 pipelineInfo.stageCount = 2;
18 pipelineInfo.pStages = shaderStages;
19 // Attach other pipeline states like rasterizer and input assembly
20 vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &particlePipeline);

```

30.5 Step 3: Write the Vertex Shader

The vertex shader updates particle positions based on their velocity and age. It also calculates whether a particle should still be visible.

```

1  // GLSL Vertex Shader - Particle
2  #version 450
3
4  layout(location = 0) in vec2 inPosition;
5  layout(location = 1) in vec2 inVelocity;
6  layout(location = 2) in vec4 inColor;
7  layout(location = 3) in float inLifetime;
8  layout(location = 4) in float inAge;
9
10 layout(location = 0) out vec4 fragColor;
11
12 layout(push_constant) uniform PushConstants {
13     float deltaTime;
14 } pc;
15
16 void main() {
17     float newAge = inAge + pc.deltaTime;
18     if (newAge > inLifetime) {
19         gl_Position = vec4(-9999.0, -9999.0, 0.0, 1.0); // Move off-screen
20     } else {
21         vec2 newPosition = inPosition + inVelocity * pc.deltaTime;
22         gl_Position = vec4(newPosition, 0.0, 1.0);
23         fragColor = inColor * (1.0 - newAge / inLifetime); // Fade over time
24     }
25 }
```

30.6 Step 4: Write the Fragment Shader

The fragment shader colors the particles based on the data passed from the vertex shader.

```

1  // GLSL Fragment Shader - Particle
2  #version 450
3
4  layout(location = 0) in vec4 fragColor;
5  layout(location = 0) out vec4 outColor;
6
7  void main() {
8      outColor = fragColor;
9  }
```

30.7 Step 5: Render the Particle System

In the rendering loop, bind the particle pipeline and buffer, and draw the particles.

```

1  // Bind the particle pipeline and buffer
2  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, particlePipeline);
3  VkDeviceSize offsets[] = { 0 };
4  vkCmdBindVertexBuffers(commandBuffer, 0, 1, &particleBuffer, offsets);
5
6  // Push the delta time constant
7  float deltaTime = computeDeltaTime();
```

```
8 vkCmdPushConstants(commandBuffer, particlePipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, sizeof(float), &  
    ↪ deltaTime);  
9  
10 // Draw particles  
11 vkCmdDraw(commandBuffer, maxParticles, 1, 0, 0);
```

30.8 Summary

In this chapter, you built a particle system using Vulkan and a vertex shader for simulation. This system dynamically updates particle attributes, rendering effects such as fire, smoke, or explosions. By optimizing and extending this approach, you can create complex and visually appealing particle effects.

Part IV

Compute Shader Projects

31

Hello, Compute Shader: Write your first compute shader

31.1 Introduction

Compute shaders in Vulkan provide a powerful way to perform general-purpose computations on the GPU. Unlike graphics shaders, compute shaders are not tied to the graphics pipeline and can be used for tasks like physics simulations, image processing, or even AI inference. In this chapter, we will implement a simple compute shader that performs basic vector addition, introducing the concepts of compute pipelines, workgroups, and data synchronization.

31.2 Overview of Compute Shaders in Vulkan

Compute shaders execute in **workgroups**, which are groups of threads operating in parallel. These threads access shared data and are synchronized at specific points. Vulkan's compute pipeline provides the flexibility to dispatch compute workloads directly to the GPU.

The steps involved are:

1. Creating a compute pipeline.
2. Setting up input and output buffers.
3. Writing and executing the compute shader.

31.3 Step 1: Define the Input and Output Buffers

We'll use two input buffers containing vectors and an output buffer to store the results.

```

1 // Define input and output data
2 std::vector<float> inputA = { 1.0f, 2.0f, 3.0f, 4.0f };
3 std::vector<float> inputB = { 5.0f, 6.0f, 7.0f, 8.0f };
4 std::vector<float> output(4, 0.0f);
5
6 // Create Vulkan buffers for inputA, inputB, and output
7 VkBufferCreateInfo bufferInfo{};
8 bufferInfo.size = sizeof(float) * inputA.size();
9 bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
10
11 VkBuffer inputABuffer, inputBBuffer, outputBuffer;
12 vkCreateBuffer(device, &bufferInfo, nullptr, &inputABuffer);
13 vkCreateBuffer(device, &bufferInfo, nullptr, &inputBBuffer);
14 vkCreateBuffer(device, &bufferInfo, nullptr, &outputBuffer);
15
16 // Allocate and bind memory to buffers
17 allocateAndBindMemory(inputABuffer, inputA.data());
18 allocateAndBindMemory(inputBBuffer, inputB.data());
19 allocateAndBindMemory(outputBuffer, output.data());

```

31.4 Step 2: Write the Compute Shader

The compute shader will read two vectors, add their elements, and write the results to an output buffer.

```

1 // GLSL Compute Shader
2 #version 450
3
4 layout(set = 0, binding = 0) buffer InputA {
5     float data[];
6 } inputA;
7
8 layout(set = 0, binding = 1) buffer InputB {
9     float data[];
10 } inputB;
11
12 layout(set = 0, binding = 2) buffer Output {
13     float data[];
14 } outputBuffer;
15
16 layout(local_size_x = 1) in; // One thread per vector element
17
18 void main() {
19     uint idx = gl_GlobalInvocationID.x;
20     outputBuffer.data[idx] = inputA.data[idx] + inputB.data[idx];
21 }

```

31.5 Step 3: Create the Compute Pipeline

Create a compute pipeline to execute the shader. Load the compiled SPIR-V shader, and set up the pipeline.

```

1 // Load the compute shader
2 VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("vector_add.comp.spv",
    ↳ VK_SHADER_STAGE_COMPUTE_BIT);
3
4 // Create the compute pipeline
5 VkComputePipelineCreateInfo pipelineInfo{};
6 pipelineInfo.stage = computeShaderStage;
7 pipelineInfo.layout = computePipelineLayout;
8
9 VkPipeline computePipeline;
10 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

31.6 Step 4: Set Up Descriptor Sets

Bind the input and output buffers to the compute shader using descriptor sets.

```

1 // Create descriptor set layout
2 VkDescriptorSetLayoutBinding inputABinding{};
3 inputABinding.binding = 0;
4 inputABinding.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
5 inputABinding.descriptorCount = 1;
6 inputABinding.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
7
8 VkDescriptorSetLayoutBinding inputBBinding = inputABinding;
9 inputBBinding.binding = 1;
10
11 VkDescriptorSetLayoutBinding outputBinding = inputABinding;
12 outputBinding.binding = 2;
13
14 VkDescriptorSetLayoutBinding bindings[] = { inputABinding, inputBBinding, outputBinding };
15
16 VkDescriptorSetLayoutCreateInfo layoutInfo{};
17 layoutInfo.bindingCount = 3;
18 layoutInfo.pBindings = bindings;
19
20 VkDescriptorSetLayout descriptorSetLayout;
21 vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout);
22
23 // Allocate and write descriptor sets
24 VkDescriptorSet descriptorSet;
25 // Assume descriptor pool is already created
26 allocateDescriptorSet(descriptorSet, descriptorSetLayout);
27
28 VkDescriptorBufferInfo inputABufferInfo = { inputABuffer, 0, VK_WHOLE_SIZE };
29 VkDescriptorBufferInfo inputBBufferInfo = { inputBBuffer, 0, VK_WHOLE_SIZE };
30 VkDescriptorBufferInfo outputBufferInfo = { outputBuffer, 0, VK_WHOLE_SIZE };
31
32 VkWriteDescriptorSet descriptorWrites[3] = {};
33 descriptorWrites[0] = createWriteDescriptorSet(descriptorSet, 0, &inputABufferInfo);
34 descriptorWrites[1] = createWriteDescriptorSet(descriptorSet, 1, &inputBBufferInfo);
35 descriptorWrites[2] = createWriteDescriptorSet(descriptorSet, 2, &outputBufferInfo);
36
37 vkUpdateDescriptorSets(device, 3, descriptorWrites, 0, nullptr);

```

31.7 Step 5: Dispatch the Compute Shader

To execute the compute shader, bind the pipeline, descriptor sets, and issue a dispatch command.

```

1 // Bind the compute pipeline and descriptor sets
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
  ↳ descriptorSet, 0, nullptr);
4
5 // Dispatch the compute shader
6 uint32_t workgroupCount = (uint32_t)inputA.size();
7 vkCmdDispatch(commandBuffer, workgroupCount, 1, 1);

```

Ensure that the buffer memory is synchronized after computation to retrieve the results.

31.8 Step 6: Retrieve and Verify Results

Map the output buffer and verify the results.

```

1 // Map and read output buffer
2 float* mappedData;
3 vkMapMemory(device, outputBufferMemory, 0, VK_WHOLE_SIZE, 0, (void**)&mappedData);
4
5 // Verify results
6 for (size_t i = 0; i < output.size(); i++) {
7     std::cout << "Result[" << i << "] = " << mappedData[i] << std::endl;
8 }
9
10 vkUnmapMemory(device, outputBufferMemory);

```

31.9 Summary

In this chapter, you implemented your first compute shader in Vulkan to perform basic vector addition. This introduction provides the foundation for more advanced compute operations, such as particle simulations, physics, or machine learning tasks.

32

Compute a Mandelbrot Set: Render fractals using compute shaders

32.1 Introduction

The Mandelbrot set is a classic example of a fractal, showcasing intricate, self-repeating patterns. Rendering a Mandelbrot set is computationally intensive but an excellent use case for compute shaders. In this chapter, we will use Vulkan compute shaders to generate the Mandelbrot set, storing the results in a texture for visualization.

32.2 Overview of Mandelbrot Set Computation

The Mandelbrot set is defined in the complex plane. For each point c , an iterative function $z_{n+1} = z_n^2 + c$ is evaluated. If the magnitude of z_n exceeds a threshold (e.g., 2) after a certain number of iterations, c is considered outside the set. Otherwise, it belongs to the set.

This process is highly parallelizable, as each pixel in the output image can be calculated independently.

32.3 Step 1: Set Up a Compute Shader for the Mandelbrot Set

The compute shader calculates the Mandelbrot set for a region of the complex plane. The results are written to a storage image, which is later rendered to the screen.

```

2 #version 450
3
4 layout(local_size_x = 16, local_size_y = 16) in;
5
6 layout(set = 0, binding = 0, rgba32f) uniform image2D outImage;
7
8 layout(push_constant) uniform PushConstants {
9     vec2 center;
10    float scale;
11    int maxIterations;
12 } pc;
13
14 void main() {
15     ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
16     vec2 uv = vec2(pixelCoords) / vec2(imageSize(outImage));
17     vec2 c = pc.center + (uv - 0.5) * pc.scale;
18
19     vec2 z = vec2(0.0);
20     int iterations = 0;
21     while (length(z) < 2.0 && iterations < pc.maxIterations) {
22         z = vec2(
23             z.x * z.x - z.y * z.y + c.x,
24             2.0 * z.x * z.y + c.y
25         );
26         iterations++;
27     }
28
29     float intensity = float(iterations) / float(pc.maxIterations);
30     vec4 color = vec4(intensity, intensity * 0.5, intensity * 0.25, 1.0);
31     imageStore(outImage, pixelCoords, color);
32 }

```

32.4 Step 2: Create a Storage Image

Allocate a Vulkan image to store the compute shader's output.

```

1 // Create storage image
2 VkImageCreateInfo imageInfo{};
3 imageInfo.imageType = VK_IMAGE_TYPE_2D;
4 imageInfo.format = VK_FORMAT_R32G32B32A32_SFLOAT;
5 imageInfo.extent = { width, height, 1 };
6 imageInfo.mipLevels = 1;
7 imageInfo.arrayLayers = 1;
8 imageInfo.usage = VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
9
10 VkImage storageImage;
11 vkCreateImage(device, &imageInfo, nullptr, &storageImage);
12
13 // Allocate and bind memory
14 VkMemoryRequirements memRequirements;
15 vkGetImageMemoryRequirements(device, storageImage, &memRequirements);
16 allocateAndBindImageMemory(storageImage, memRequirements);

```

32.5 Step 3: Set Up the Compute Pipeline

Create a compute pipeline for the Mandelbrot shader.

```

1 // Load compute shader

```

```

2  VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("mandelbrot.comp.spv",
    ↪ VK_SHADER_STAGE_COMPUTE_BIT);
3
4  // Create compute pipeline
5  VkComputePipelineCreateInfo pipelineInfo{};
6  pipelineInfo.stage = computeShaderStage;
7  pipelineInfo.layout = computePipelineLayout;
8
9  VkPipeline computePipeline;
10 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

32.6 Step 4: Dispatch the Compute Shader

Configure push constants to define the region of the complex plane to render and dispatch the compute shader.

```

1  // Define push constants
2  struct PushConstants {
3      glm::vec2 center = glm::vec2(-0.75f, 0.0f);
4      float scale = 3.0f;
5      int maxIterations = 500;
6  } pushConstants;
7
8  // Bind pipeline and descriptor sets
9  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
10 vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(PushConstants),
    ↪ &pushConstants);
11
12 // Dispatch compute shader
13 uint32_t workgroupX = (width + 15) / 16;
14 uint32_t workgroupY = (height + 15) / 16;
15 vkCmdDispatch(commandBuffer, workgroupX, workgroupY, 1);

```

32.7 Step 5: Visualize the Result

The storage image must be transitioned to a sampled image layout for visualization. Render it to the screen using a simple fullscreen quad.

```

1  // Transition image layout to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
2  transitionImageLayout(storageImage, VK_IMAGE_LAYOUT_GENERAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
3
4  // Render the image as a fullscreen quad
5  renderFullscreenQuad(storageImage);

```

32.8 Interactive Exploration

To make the Mandelbrot set interactive, you can modify the push constants dynamically based on user input. For example:

- Zoom in/out by adjusting the `scale`.
- Pan across the fractal by modifying the `center`.
- Increase `maxIterations` for more detail.

32.9 Summary

This chapter demonstrated how to compute and render a Mandelbrot set using Vulkan compute shaders. This project introduced key concepts like storage images, compute pipelines, and workgroup execution. By extending this framework, you can explore other fractals or perform complex mathematical visualizations.

DRAFT
version
0.681

33

Add Image Blur Effect: Apply a Gaussian blur using compute shaders

33.1 Introduction

Blurring is a common image-processing effect used in graphics for creating soft transitions, simulating depth of field, or applying post-processing effects. A Gaussian blur is a weighted average of pixel values, where closer pixels contribute more to the output. In this chapter, we will use Vulkan compute shaders to apply a Gaussian blur to an image, leveraging GPU parallelism for efficient processing.

33.2 Overview of Gaussian Blur

The Gaussian blur algorithm operates by convolving an image with a Gaussian kernel. The kernel is a matrix of weights that define how surrounding pixels influence the center pixel. For a 2D image, the Gaussian kernel is defined as:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where σ controls the spread of the blur. The algorithm involves:

1. A horizontal pass, blurring each row.
2. A vertical pass, blurring each column of the horizontally blurred result.

33.3 Step 1: Prepare Input and Output Images

Load an image as input, and create a storage image for the blurred result.

```

1  // Load input image and create storage image
2  VkImage inputImage = loadImage("input.jpg");
3
4  VkImageCreateInfo imageInfo{};
5  imageInfo.imageType = VK_IMAGE_TYPE_2D;
6  imageInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
7  imageInfo.extent = { width, height, 1 };
8  imageInfo.mipLevels = 1;
9  imageInfo.arrayLayers = 1;
10 imageInfo.usage = VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
11
12 VkImage outputImage;
13 vkCreateImage(device, &imageInfo, nullptr, &outputImage);
14
15 // Allocate and bind memory
16 VkMemoryRequirements memRequirements;
17 vkGetImageMemoryRequirements(device, outputImage, &memRequirements);
18 allocateAndBindImageMemory(outputImage, memRequirements);

```

33.4 Step 2: Write the Compute Shader for Gaussian Blur

The compute shader performs a horizontal and vertical blur in two passes. First, it reads from the input image and writes horizontally blurred results to an intermediate image. Then, it reads from the intermediate image to apply the vertical blur.

```

1  // GLSL Compute Shader - Gaussian Blur
2  #version 450
3
4  layout(local_size_x = 16, local_size_y = 16) in;
5
6  layout(set = 0, binding = 0) uniform sampler2D inputImage;
7  layout(set = 0, binding = 1, rgba32f) uniform image2D outputImage;
8
9  layout(push_constant) uniform PushConstants {
10     vec2 direction; // (1.0, 0.0) for horizontal, (0.0, 1.0) for vertical
11     float sigma;
12     int kernelRadius;
13 } pc;
14
15 void main() {
16     ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
17     vec2 texCoords = vec2(pixelCoords) / vec2(imageSize(outputImage));
18
19     vec3 color = vec3(0.0);
20     float weightSum = 0.0;
21
22     for (int i = -pc.kernelRadius; i <= pc.kernelRadius; ++i) {
23         float weight = exp(-0.5 * (i / pc.sigma) * (i / pc.sigma));
24         vec2 offset = pc.direction * float(i) / vec2(imageSize(outputImage));
25         color += texture(inputImage, texCoords + offset).rgb * weight;
26         weightSum += weight;
27     }
28
29     color /= weightSum;
30     imageStore(outputImage, pixelCoords, vec4(color, 1.0));
31 }

```

33.5 Step 3: Set Up the Compute Pipeline

Compile the shader and create the compute pipeline.

```

1 // Load compute shader
2 VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("gaussian_blur.comp.spv",
    ↳ VK_SHADER_STAGE_COMPUTE_BIT);
3
4 // Create compute pipeline
5 VkComputePipelineCreateInfo pipelineInfo{};
6 pipelineInfo.stage = computeShaderStage;
7 pipelineInfo.layout = computePipelineLayout;
8
9 VkPipeline computePipeline;
10 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

33.6 Step 4: Dispatch Horizontal and Vertical Blur Passes

Perform two passes: first horizontally, then vertically. Push constants are used to configure the blur direction and kernel radius.

```

1 // Horizontal blur pass
2 PushConstants pushConstants = {};
3 pushConstants.direction = glm::vec2(1.0f, 0.0f); // Horizontal blur
4 pushConstants.sigma = 2.0f;
5 pushConstants.kernelRadius = 5;
6
7 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
8 vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(PushConstants),
    ↳ &pushConstants);
9 vkCmdDispatch(commandBuffer, (width + 15) / 16, (height + 15) / 16, 1);
10
11 // Transition output image to general layout for the next pass
12 transitionImageLayout(outputImage, VK_IMAGE_LAYOUT_GENERAL, VK_IMAGE_LAYOUT_GENERAL);
13
14 // Vertical blur pass
15 pushConstants.direction = glm::vec2(0.0f, 1.0f); // Vertical blur
16 vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(PushConstants),
    ↳ &pushConstants);
17 vkCmdDispatch(commandBuffer, (width + 15) / 16, (height + 15) / 16, 1);

```

33.7 Step 5: Visualize the Result

Transition the blurred output image to a sampled layout, and render it to the screen using a fullscreen quad.

```

1 // Transition to shader-read-only layout
2 transitionImageLayout(outputImage, VK_IMAGE_LAYOUT_GENERAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
3
4 // Render the blurred image
5 renderFullscreenQuad(outputImage);

```

33.8 Tuning the Blur Effect

To control the blur intensity:

- Adjust the `sigma` value to increase or decrease the spread of the blur.
- Change the `kernelRadius` to include more or fewer pixels in the calculation.

33.9 Summary

In this chapter, you implemented a Gaussian blur using Vulkan compute shaders. By leveraging two-pass processing with horizontal and vertical passes, the shader efficiently generates a high-quality blur. This project introduces fundamental image-processing concepts and highlights the power of compute shaders for handling parallel workloads.

34

Simulate Physics: Implement a simple physics engine with compute shaders

34.1 Introduction

Physics simulations are fundamental in games, animations, and scientific visualizations. Implementing a physics engine using Vulkan compute shaders allows you to leverage GPU parallelism for efficient calculations. In this chapter, we will create a simple physics engine that simulates a particle system, complete with gravity, collisions, and basic interactions.

34.2 Overview of the Physics Simulation

The simulation models particles as objects with position, velocity, and mass. Each particle's movement is governed by Newtonian mechanics, with forces such as gravity and collision responses applied during each simulation step. Key concepts include:

- **Particle integration:** Update particle positions and velocities over time.
- **Collision detection and response:** Handle interactions between particles and boundaries.
- **Parallel processing:** Each particle's behavior is calculated independently, making compute shaders an ideal tool.

34.3 Step 1: Define the Particle Data Structure

We define a data structure for the particles, storing their positions, velocities, and other properties.

```

1 // Particle structure
2 struct Particle {
3     vec2 position;
4     vec2 velocity;
5     float mass;
6     float radius;
7 };

```

In Vulkan, this data will be stored in a shader storage buffer object (SSBO), accessible to the compute shader.

34.4 Step 2: Write the Physics Compute Shader

The compute shader performs the physics simulation for each particle. It applies gravity, updates positions, and handles collisions.

```

1 // GLSL Compute Shader - Physics Simulation
2 #version 450
3
4 layout(local_size_x = 256) in;
5
6 struct Particle {
7     vec2 position;
8     vec2 velocity;
9     float mass;
10    float radius;
11 };
12
13 layout(std430, binding = 0) buffer Particles {
14     Particle particles[];
15 };
16
17 layout(push_constant) uniform PushConstants {
18     vec2 gravity; // Gravity force (e.g., vec2(0.0, -9.8))
19     float deltaTime; // Time step
20     vec2 bounds; // Simulation bounds (width, height)
21 } pc;
22
23 void main() {
24     uint idx = gl_GlobalInvocationID.x;
25
26     if (idx >= particles.length()) return;
27
28     Particle p = particles[idx];
29
30     // Apply gravity
31     p.velocity += pc.gravity * pc.deltaTime;
32
33     // Update position
34     p.position += p.velocity * pc.deltaTime;
35
36     // Handle boundary collisions
37     if (p.position.x - p.radius < 0.0 || p.position.x + p.radius > pc.bounds.x) {
38         p.velocity.x *= -0.8; // Dampen velocity

```

```

39     p.position.x = clamp(p.position.x, p.radius, pc.bounds.x - p.radius);
40 }
41 if (p.position.y - p.radius < 0.0 || p.position.y + p.radius > pc.bounds.y) {
42     p.velocity.y *= -0.8; // Dampen velocity
43     p.position.y = clamp(p.position.y, p.radius, pc.bounds.y - p.radius);
44 }
45
46 particles[idx] = p;
47 }

```

34.5 Step 3: Create the SSBO and Initialize Particles

Allocate a buffer to hold the particle data and initialize it with random positions, velocities, and masses.

```

1  // Create particle buffer
2  VkBufferCreateInfo bufferInfo{};
3  bufferInfo.size = sizeof(Particle) * particleCount;
4  bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
5  bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
6
7  VkBuffer particleBuffer;
8  vkCreateBuffer(device, &bufferInfo, nullptr, &particleBuffer);
9
10 // Allocate and bind memory
11 VkMemoryRequirements memRequirements;
12 vkGetBufferMemoryRequirements(device, particleBuffer, &memRequirements);
13 allocateAndBindBufferMemory(particleBuffer, memRequirements);
14
15 // Initialize particle data
16 std::vector<Particle> particles(particleCount);
17 for (Particle& p : particles) {
18     p.position = randomVec2(bounds);
19     p.velocity = randomVec2(vec2(-1.0f, 1.0f));
20     p.mass = 1.0f;
21     p.radius = 5.0f;
22 }
23 uploadDataToBuffer(particleBuffer, particles);

```

34.6 Step 4: Configure the Compute Pipeline

Compile the shader and create a compute pipeline for the physics simulation.

```

1  // Load compute shader
2  VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("physics_simulation.comp.spv",
3  // ↳ VK_SHADER_STAGE_COMPUTE_BIT);
4
5  // Create compute pipeline
6  VkComputePipelineCreateInfo pipelineInfo{};
7  pipelineInfo.stage = computeShaderStage;
8  pipelineInfo.layout = computePipelineLayout;
9
10 VkPipeline computePipeline;
11 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

34.7 Step 5: Dispatch the Compute Shader

Execute the compute shader, updating particle positions in each frame.

```

1  // Push constants for simulation parameters
2  PushConstants pushConstants = 0;
3  pushConstants.gravity = glm::vec2(0.0f, -9.8f);
4  pushConstants.deltaTime = 0.016f; // 16 ms per frame
5  pushConstants.bounds = glm::vec2(width, height);
6
7  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
8  vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(PushConstants),
   →   &pushConstants);
9
10 // Dispatch compute shader
11 uint32_t workgroupCount = (particleCount + 255) / 256;
12 vkCmdDispatch(commandBuffer, workgroupCount, 1, 1);

```

34.8 Step 6: Visualize the Particles

Use a vertex shader and a simple point rendering pipeline to visualize the particle positions.

```

1  // Update particle buffer to read-only for rendering
2  transitionBufferToReadOnly(particleBuffer);
3
4  // Render particles as points
5  renderParticles(particleBuffer);

```

34.9 Interactive Features

You can extend this physics engine with additional features:

- Add mouse or keyboard interaction to apply external forces.
- Simulate collisions between particles.
- Introduce additional forces, such as wind or magnetic fields.

34.10 Summary

This chapter demonstrated how to simulate basic physics with Vulkan compute shaders. The project highlighted the power of parallel processing for physics calculations and introduced SSBOs for managing particle data.

35

Game of Life Simulation: Use compute shaders for cellular automata

35.1 Introduction

The Game of Life, devised by mathematician John Conway, is a classic example of cellular automata. It operates on a grid where each cell evolves over time according to simple rules based on the states of its neighbors. In this chapter, we will use Vulkan compute shaders to simulate the Game of Life efficiently, leveraging the parallelism of GPUs to update all cells simultaneously.

35.2 Overview of the Rules

The Game of Life grid consists of cells that are either alive or dead. The rules for evolving each cell are:

1. **Survival:** A living cell remains alive if it has 2 or 3 living neighbors.
2. **Death:** A living cell dies if it has fewer than 2 or more than 3 neighbors (underpopulation or overpopulation).
3. **Birth:** A dead cell becomes alive if it has exactly 3 living neighbors.

Each cell's state is updated simultaneously based on the states of its neighbors, making this problem ideal for compute shaders.

35.3 Step 1: Define the Grid Data Structure

The grid will be stored in a buffer as a 2D array, where each cell is represented by a single integer (1 for alive, 0 for dead).

```

1 // Define the grid dimensions
2 const uint32_t gridWidth = 1024;
3 const uint32_t gridHeight = 1024;
4
5 // Buffer to store the grid
6 VkBuffer gridBuffer;
7 VkBufferCreateInfo bufferInfo{};
8 bufferInfo.size = gridWidth * gridHeight * sizeof(uint32_t);
9 bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
10 bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
11
12 vkCreateBuffer(device, &bufferInfo, nullptr, &gridBuffer);
13
14 // Allocate and bind memory
15 VkMemoryRequirements memRequirements;
16 vkGetBufferMemoryRequirements(device, gridBuffer, &memRequirements);
17 allocateAndBindBufferMemory(gridBuffer, memRequirements);
18
19 // Initialize the grid with random states
20 std::vector<uint32_t> grid(gridWidth * gridHeight);
21 for (uint32_t &cell : grid) {
22     cell = rand() % 2; // Randomly set cell state to 0 or 1
23 }
24 uploadDataToBuffer(gridBuffer, grid);

```

35.4 Step 2: Write the Compute Shader for the Game of Life

The compute shader reads the current grid state, calculates the number of living neighbors for each cell, and updates the state of each cell in a new grid.

```

1 // GLSL Compute Shader - Game of Life
2 #version 450
3
4 layout(local_size_x = 16, local_size_y = 16) in;
5
6 layout(std430, binding = 0) buffer InputGrid {
7     uint currentGrid[];
8 };
9
10 layout(std430, binding = 1) buffer OutputGrid {
11     uint nextGrid[];
12 };
13
14 layout(push_constant) uniform PushConstants {
15     ivec2 gridSize; // Grid dimensions (width, height)
16 } pc;
17
18 void main() {
19     ivec2 coords = ivec2(gl_GlobalInvocationID.xy);
20     int index = coords.y * pc.gridSize.x + coords.x;
21
22     if (coords.x >= pc.gridSize.x || coords.y >= pc.gridSize.y) return;
23
24     // Count living neighbors
25     int livingNeighbors = 0;

```

```

26     for (int dy = -1; dy <= 1; ++dy) {
27         for (int dx = -1; dx <= 1; ++dx) {
28             if (dx == 0 && dy == 0) continue; // Skip the cell itself
29
30             ivec2 neighborCoords = coords + ivec2(dx, dy);
31             if (neighborCoords.x >= 0 && neighborCoords.x < pc.gridSize.x &&
32                 neighborCoords.y >= 0 && neighborCoords.y < pc.gridSize.y) {
33                 int neighborIndex = neighborCoords.y * pc.gridSize.x + neighborCoords.x;
34                 livingNeighbors += int(currentGrid[neighborIndex]);
35             }
36         }
37     }
38
39     // Apply the Game of Life rules
40     uint cellState = currentGrid[index];
41     if (cellState == 1) {
42         nextGrid[index] = (livingNeighbors == 2 || livingNeighbors == 3) ? 1 : 0;
43     } else {
44         nextGrid[index] = (livingNeighbors == 3) ? 1 : 0;
45     }
46 }

```

35.5 Step 3: Create and Configure the Compute Pipeline

Compile the shader and set up the compute pipeline to execute the simulation.

```

1 // Load compute shader
2 VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("game_of_life.comp.spv",
3     ↳ VK_SHADER_STAGE_COMPUTE_BIT);
4
5 // Create compute pipeline
6 VkComputePipelineCreateInfo pipelineInfo{};
7 pipelineInfo.stage = computeShaderStage;
8 pipelineInfo.layout = computePipelineLayout;
9
10 VkPipeline computePipeline;
11 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

35.6 Step 4: Execute the Compute Shader

Each frame, execute the compute shader to update the grid. Alternate between two buffers to store the current and next grid states.

```

1 // Push constants for grid size
2 PushConstants pushConstants = {};
3 pushConstants.gridSize = glm::ivec2(gridWidth, gridHeight);
4
5 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
6 vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(PushConstants),
7     ↳ &pushConstants);
8
9 // Bind input and output buffers
10 VkDescriptorSet descriptorSet = createDescriptorSet(gridBuffer, nextGridBuffer);
11 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
12     ↳ descriptorSet, 0, nullptr);
13
14 // Dispatch compute shader

```



```
13 vkCmdDispatch(commandBuffer, (gridWidth + 15) / 16, (gridHeight + 15) / 16, 1);
```

After the compute pass, swap the buffers so that the output becomes the input for the next frame.

```
1 // Swap buffers
2 std::swap(gridBuffer, nextGridBuffer);
```

35.7 Step 5: Visualize the Grid

Render the grid using a fullscreen quad, coloring cells based on their state (e.g., black for dead, white for alive).

```
1 // Transition grid buffer for visualization
2 transitionBufferToReadOnly(gridBuffer);
3
4 // Render the grid as a texture
5 renderGrid(gridBuffer, gridWidth, gridHeight);
```

35.8 Interactive Features

To make the simulation more engaging, consider adding:

- **Mouse input:** Allow users to toggle cell states by clicking on the grid.
- **Dynamic parameters:** Adjust grid size or simulation speed interactively.
- **Custom patterns:** Enable users to load or define initial patterns.

35.9 Summary

In this chapter, you implemented the Game of Life simulation using Vulkan compute shaders. This project demonstrated how cellular automata can be efficiently simulated with GPU parallelism and introduced techniques for managing grid-based data. The Game of Life serves as a foundation for more complex simulations, including fluid dynamics or advanced pattern generation.

36

Generate Perlin Noise: Use compute shaders to create noise textures

36.1 Introduction

Perlin noise is a gradient noise function widely used in computer graphics to generate textures, terrain, and other natural-looking patterns. Unlike simple random noise, Perlin noise produces smooth transitions, making it ideal for procedural content. In this chapter, we will use Vulkan compute shaders to generate a 2D Perlin noise texture efficiently.

36.2 Understanding Perlin Noise

Perlin noise works by:

1. Dividing the 2D space into a grid of gradients.
2. Assigning a random gradient vector to each grid point.
3. Interpolating the gradients smoothly between grid points using a fade function.

The resulting texture has coherent noise patterns with a natural appearance.

36.3 Step 1: Define the Compute Shader for Perlin Noise

The compute shader calculates the Perlin noise value for each pixel in the output texture. The following code implements the basic Perlin noise algorithm.

```

1  // GLSL Compute Shader - Perlin Noise
2  #version 450
3
4  layout(local_size_x = 16, local_size_y = 16) in;
5
6  layout(binding = 0, rgba32f) uniform image2D outputImage;
7
8  // Fade function for smooth interpolation
9  float fade(float t) {
10     return t * t * t * (t * (t * 6.0 - 15.0) + 10.0);
11 }
12
13 // Linear interpolation
14 float lerp(float a, float b, float t) {
15     return a + t * (b - a);
16 }
17
18 // Gradient function
19 float grad(int hash, float x, float y) {
20     int h = hash & 3;
21     float u = h < 2 ? x : y;
22     float v = h < 2 ? y : x;
23     return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
24 }
25
26 // Hash function
27 int permute(int x) {
28     return (x * 34 + 1) % 289;
29 }
30
31 // Perlin noise function
32 float perlinNoise(vec2 pos) {
33     ivec2 gridPos = ivec2(floor(pos));
34     vec2 relPos = pos - vec2(gridPos);
35
36     // Gradient hashes for grid points
37     int g00 = permute(permute(gridPos.x) + gridPos.y);
38     int g01 = permute(permute(gridPos.x) + gridPos.y + 1);
39     int g10 = permute(permute(gridPos.x + 1) + gridPos.y);
40     int g11 = permute(permute(gridPos.x + 1) + gridPos.y + 1);
41
42     // Gradients
43     float n00 = grad(g00, relPos.x, relPos.y);
44     float n01 = grad(g01, relPos.x, relPos.y - 1.0);
45     float n10 = grad(g10, relPos.x - 1.0, relPos.y);
46     float n11 = grad(g11, relPos.x - 1.0, relPos.y - 1.0);
47
48     // Fade and interpolate
49     vec2 fadeRelPos = vec2(fade(relPos.x), fade(relPos.y));
50     float nx0 = lerp(n00, n10, fadeRelPos.x);
51     float nx1 = lerp(n01, n11, fadeRelPos.x);
52     return lerp(nx0, nx1, fadeRelPos.y);
53 }
54
55 void main() {
56     ivec2 pixelCoord = ivec2(gl_GlobalInvocationID.xy);
57     ivec2 imageSize = imageSize(outputImage);
58
59     if (pixelCoord.x >= imageSize.x || pixelCoord.y >= imageSize.y) return;
60
61     // Map pixel coordinates to noise space
62     vec2 noisePos = vec2(pixelCoord) / vec2(imageSize) * 10.0;
63
64     // Generate Perlin noise
65     float noiseValue = perlinNoise(noisePos);
66
67     // Output grayscale color

```

```

68     vec4 color = vec4(vec3(noiseValue * 0.5 + 0.5), 1.0);
69     imageStore(outputImage, pixelCoord, color);
70 }

```

36.4 Step 2: Create the Compute Pipeline

Compile the shader and create a Vulkan compute pipeline to execute it.

```

1  // Load compute shader
2  VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("perlin_noise.comp.spv",
    ↳ VK_SHADER_STAGE_COMPUTE_BIT);
3
4  // Create compute pipeline
5  VkComputePipelineCreateInfo pipelineInfo{};
6  pipelineInfo.stage = computeShaderStage;
7  pipelineInfo.layout = computePipelineLayout;
8
9  VkPipeline computePipeline;
10 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

36.5 Step 3: Allocate and Bind an Image for Output

Create an image to store the generated Perlin noise texture.

```

1  // Create an output image
2  VkImageCreateInfo imageInfo{};
3  imageInfo.imageType = VK_IMAGE_TYPE_2D;
4  imageInfo.extent.width = 1024;
5  imageInfo.extent.height = 1024;
6  imageInfo.extent.depth = 1;
7  imageInfo.mipLevels = 1;
8  imageInfo.arrayLayers = 1;
9  imageInfo.format = VK_FORMAT_R32G32B32A32_SFLOAT;
10 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
11 imageInfo.usage = VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
12
13 VkImage outputImage;
14 vkCreateImage(device, &imageInfo, nullptr, &outputImage);
15
16 // Allocate memory for the image
17 VkMemoryRequirements memRequirements;
18 vkGetImageMemoryRequirements(device, outputImage, &memRequirements);
19 allocateAndBindImageMemory(outputImage, memRequirements);

```

36.6 Step 4: Dispatch the Compute Shader

Execute the compute shader to generate the Perlin noise texture.

```

1  // Bind the output image to a descriptor set
2  VkDescriptorSet descriptorSet = createDescriptorSet(outputImage);
3
4  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
5  vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
    ↳ descriptorSet, 0, nullptr);

```

```
6
7 // Dispatch compute shader
8 vkCmdDispatch(commandBuffer, (1024 + 15) / 16, (1024 + 15) / 16, 1);
```

36.7 Step 5: Visualize the Noise Texture

Render the generated Perlin noise texture onto a fullscreen quad using a simple fragment shader.

```
1 // Render the Perlin noise texture
2 renderTexture(outputImage);
```

36.8 Extending the Perlin Noise Implementation

This implementation can be extended in several ways:

- **Octaves:** Combine multiple layers of Perlin noise at different scales for more complex patterns.
- **Tiling:** Modify the algorithm to create seamless, tiling textures.
- **3D Noise:** Extend the implementation to generate 3D Perlin noise for volumetric effects.

36.9 Summary

In this chapter, you learned how to generate 2D Perlin noise textures using Vulkan compute shaders. This project demonstrated the versatility of compute shaders for procedural texture generation and introduced key concepts like interpolation and gradient-based algorithms.

37

Sort Data with Compute Shaders: Implement a parallel sorting algorithm

37.1 Introduction

Sorting is a fundamental operation in many computational tasks, and efficient sorting is critical for large datasets. Leveraging Vulkan compute shaders, we can implement parallel sorting algorithms, such as the Bitonic Sort or Radix Sort, to take advantage of the GPU's parallel processing capabilities. In this chapter, we will focus on implementing Bitonic Sort—a highly parallelizable sorting algorithm.

37.2 Overview of Bitonic Sort

Bitonic Sort works by:

1. Dividing the input data into small sequences.
2. Merging these sequences into bitonic sequences (sequences that monotonically increase and then decrease, or vice versa).
3. Sorting the bitonic sequences into fully sorted sequences.

This process repeats in a hierarchical manner until the entire dataset is sorted.

37.3 Step 1: Prepare the Input Data

First, we prepare a buffer to hold the data to be sorted.


```

1 // Number of elements to sort
2 const uint32_t elementCount = 1024;
3
4 // Create a buffer to store the data
5 VkBuffer dataBuffer;
6 VkBufferCreateInfo bufferInfo{};
7 bufferInfo.size = elementCount * sizeof(float);
8 bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
9 bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
10
11 vkCreateBuffer(device, &bufferInfo, nullptr, &dataBuffer);
12
13 // Allocate and bind memory
14 VkMemoryRequirements memRequirements;
15 vkGetBufferMemoryRequirements(device, dataBuffer, &memRequirements);
16 allocateAndBindBufferMemory(dataBuffer, memRequirements);
17
18 // Initialize the buffer with random data
19 std::vector<float> data(elementCount);
20 for (float &value : data) {
21     value = static_cast<float>(rand()) / RAND_MAX; // Random float between 0 and 1
22 }
23 uploadDataToBuffer(dataBuffer, data);

```

37.4 Step 2: Write the Compute Shader for Bitonic Sort

The compute shader performs the sorting in multiple passes, dividing the work among work-groups and threads.

```

1 // GLSL Compute Shader - Bitonic Sort
2 #version 450
3
4 layout(local_size_x = 256) in;
5
6 layout(std430, binding = 0) buffer Data {
7     float values[];
8 };
9
10 layout(push_constant) uniform PushConstants {
11     uint stage; // Current stage of the sort
12     uint pass; // Current pass within the stage
13 } pc;
14
15 void bitonicSort(uint idx, uint stage, uint pass) {
16     uint pairDistance = 1 << pass;
17     uint blockSize = 1 << (stage + 1);
18
19     // Determine indices to compare and swap
20     uint blockStart = (idx / blockSize) * blockSize;
21     uint pairIdx = idx ^ pairDistance;
22
23     if (pairIdx > idx && idx - blockStart < blockSize / 2) {
24         bool ascending = ((idx & blockSize) == 0);
25         if ((ascending && values[idx] > values[pairIdx]) ||
26             (!ascending && values[idx] < values[pairIdx])) {
27             // Swap values
28             float temp = values[idx];
29             values[idx] = values[pairIdx];
30             values[pairIdx] = temp;
31         }
32     }
33 }

```

```

34
35 void main() {
36     uint idx = gl_GlobalInvocationID.x;
37     if (idx >= values.length()) return;
38
39     bitonicSort(idx, pc.stage, pc.pass);
40 }

```

37.5 Step 3: Create and Configure the Compute Pipeline

Compile the shader and create the Vulkan compute pipeline.

```

1 // Load compute shader
2 VkPipelineShaderStageCreateInfo computeShaderStage = loadShader("bitonic_sort.comp.spv",
    ↳ VK_SHADER_STAGE_COMPUTE_BIT);
3
4 // Create compute pipeline
5 VkComputePipelineCreateInfo pipelineInfo{};
6 pipelineInfo.stage = computeShaderStage;
7 pipelineInfo.layout = computePipelineLayout;
8
9 VkPipeline computePipeline;
10 vkCreateComputePipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &computePipeline);

```

37.6 Step 4: Execute the Sorting Algorithm

The sorting algorithm consists of multiple stages and passes. For each stage and pass, we dispatch the compute shader to process the data.

```

1 // Number of stages in the sort
2 uint32_t numStages = static_cast<uint32_t>(log2(elementCount));
3
4 for (uint32_t stage = 0; stage < numStages; ++stage) {
5     for (uint32_t pass = 0; pass <= stage; ++pass) {
6         // Push constants
7         PushConstants pushConstants = { stage, pass };
8         vkCmdPushConstants(commandBuffer, computePipelineLayout, VK_SHADER_STAGE_COMPUTE_BIT, 0, sizeof(
    ↳ PushConstants), &pushConstants);
9
10        // Bind pipeline and descriptor set
11        vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
12        vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
    ↳ descriptorSet, 0, nullptr);
13
14        // Dispatch compute shader
15        uint32_t workgroupCount = (elementCount + 255) / 256;
16        vkCmdDispatch(commandBuffer, workgroupCount, 1, 1);
17    }
18 }

```

37.7 Step 5: Retrieve and Verify the Sorted Data

After executing the compute pipeline, copy the sorted data back to the CPU and verify the results.

```
1 // Retrieve data from the GPU
2 std::vector<float> sortedData(elementCount);
3 downloadDataFromBuffer(dataBuffer, sortedData);
4
5 // Verify that the data is sorted
6 bool isSorted = std::is_sorted(sortedData.begin(), sortedData.end());
7 std::cout << "Data sorted successfully: " << std::boolalpha << isSorted << std::endl;
```

37.8 Optimizations and Extensions

- **Larger Datasets:** For datasets exceeding local memory limits, consider partitioning the data and merging results in multiple passes.
- **Radix Sort:** For non-floating-point data, implement a Radix Sort to handle integers efficiently.
- **Visualization:** Render a graphical representation of the sorting process to make it more interactive and intuitive.

37.9 Summary

In this chapter, we implemented a parallel Bitonic Sort using Vulkan compute shaders. This project illustrated how to use compute shaders for sorting tasks, leveraging GPU parallelism to handle large datasets efficiently.

38

Create a GPU Particle System: Simulate particles entirely on the GPU

38.1 Introduction

Particle systems are a staple of computer graphics, used to simulate effects like fire, smoke, rain, and explosions. While CPU-based particle systems are common, running particle simulations on the GPU can achieve significantly higher performance by utilizing its parallel computing capabilities. In this chapter, we will create a GPU-based particle system entirely in Vulkan, leveraging compute shaders for simulation and vertex shaders for rendering.

38.2 Overview of a GPU Particle System

A GPU particle system typically involves:

1. **Particle Data Buffers:** Buffers to store properties like position, velocity, color, and lifetime for each particle.
2. **Compute Shader:** Updates particle properties based on physics, time, and external forces.
3. **Vertex Shader:** Renders particles as points, quads, or other geometry.
4. **Descriptor Sets:** Used to pass particle data between the compute and graphics pipelines.

38.3 Step 1: Define Particle Data Structures

The particle data is stored in a structured buffer. Each particle contains attributes like position, velocity, color, and lifetime.

```

1  // Define particle structure
2  struct Particle {
3      vec4 position; // x, y, z, w (w is unused)
4      vec4 velocity; // dx, dy, dz, w (w is unused)
5      vec4 color; // r, g, b, a
6      float lifetime; // Remaining lifetime of the particle
7  };
8
9  // Create a buffer for particles
10 VkBuffer particleBuffer;
11 VkBufferCreateInfo bufferInfo{};
12 bufferInfo.size = maxParticles * sizeof(Particle);
13 bufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
14
15 vkCreateBuffer(device, &bufferInfo, nullptr, &particleBuffer);
16
17 // Allocate and bind memory
18 VkMemoryRequirements memRequirements;
19 vkGetBufferMemoryRequirements(device, particleBuffer, &memRequirements);
20 allocateAndBindBufferMemory(particleBuffer, memRequirements);

```

38.4 Step 2: Write the Compute Shader for Particle Simulation

The compute shader updates particle positions, velocities, and lifetimes each frame. Dead particles are respawned with initial properties.

```

1  // GLSL Compute Shader - Particle Simulation
2  #version 450
3
4  layout(local_size_x = 256) in;
5
6  layout(std430, binding = 0) buffer Particles {
7      struct {
8          vec4 position;
9          vec4 velocity;
10         vec4 color;
11         float lifetime;
12     } particles[];
13 };
14
15 layout(push_constant) uniform PushConstants {
16     float deltaTime; // Time since the last frame
17     vec3 gravity; // Gravity vector
18 } pc;
19
20 void main() {
21     uint idx = gl_GlobalInvocationID.x;
22     if (idx >= particles.length()) return;
23
24     // Update lifetime
25     particles[idx].lifetime -= pc.deltaTime;
26     if (particles[idx].lifetime <= 0.0) {
27         // Respawn particle
28         particles[idx].position = vec4(0.0, 0.0, 0.0, 1.0);
29         particles[idx].velocity = vec4(rand() - 0.5, rand(), rand() - 0.5, 0.0) * 5.0;

```

```

30     particles[idx].color = vec4(rand(), rand(), rand(), 1.0);
31     particles[idx].lifetime = 5.0 + rand() * 5.0;
32 }
33
34 // Update position and velocity
35 particles[idx].velocity += vec4(pc.gravity * pc.deltaTime, 0.0);
36 particles[idx].position += particles[idx].velocity * pc.deltaTime;
37 }

```

38.5 Step 3: Create and Bind Descriptor Sets

The particle buffer is bound to a descriptor set to allow the compute shader to access and modify it.

```

1  // Create descriptor set layout
2  VkDescriptorSetLayoutBinding layoutBinding0;
3  layoutBinding.binding = 0;
4  layoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
5  layoutBinding.descriptorCount = 1;
6  layoutBinding.stageFlags = VK_SHADER_STAGE_COMPUTE_BIT;
7
8  VkDescriptorSetLayout descriptorSetLayout;
9  VkDescriptorSetLayoutCreateInfo layoutInfo0;
10 layoutInfo.bindingCount = 1;
11 layoutInfo.pBindings = &layoutBinding;
12
13 vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout);
14
15 // Allocate and update descriptor set
16 VkDescriptorSet descriptorSet = allocateDescriptorSet(descriptorSetLayout);
17 VkDescriptorBufferInfo bufferInfo0;
18 bufferInfo.buffer = particleBuffer;
19 bufferInfo.offset = 0;
20 bufferInfo.range = VK_WHOLE_SIZE;
21
22 VkWriteDescriptorSet writeDescriptorSet0;
23 writeDescriptorSet.dstSet = descriptorSet;
24 writeDescriptorSet.dstBinding = 0;
25 writeDescriptorSet.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_BUFFER;
26 writeDescriptorSet.descriptorCount = 1;
27 writeDescriptorSet.pBufferInfo = &bufferInfo;
28
29 vkUpdateDescriptorSets(device, 1, &writeDescriptorSet, 0, nullptr);

```

38.6 Step 4: Render Particles Using the Graphics Pipeline

Particles are rendered as points using a simple vertex and fragment shader. The vertex shader reads particle positions from the buffer.

```

1  // GLSL Vertex Shader - Render Particles
2  #version 450
3
4  layout(location = 0) in vec4 inPosition;
5  layout(location = 1) in vec4 inColor;
6
7  layout(location = 0) out vec4 fragColor;
8
9  void main() {

```



```

10     gl_Position = inPosition;
11     gl_PointSize = 5.0; // Particle size
12     fragColor = inColor;
13 }
14
15 // GLSL Fragment Shader
16 #version 450
17
18 layout(location = 0) in vec4 fragColor;
19 layout(location = 0) out vec4 outColor;
20
21 void main() {
22     outColor = fragColor;
23 }

```

38.7 Step 5: Execute the Simulation and Rendering Loop

Combine the compute and graphics pipelines in the rendering loop.

```

1 // Dispatch compute shader to simulate particles
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout, 0, 1, &
    ↳ descriptorSet, 0, nullptr);
4 vkCmdDispatch(commandBuffer, (maxParticles + 255) / 256, 1, 1);
5
6 // Bind vertex buffer for rendering
7 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
8 VkDeviceSize offsets[] = {0};
9 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &particleBuffer, offsets);
10
11 // Draw particles
12 vkCmdDraw(commandBuffer, maxParticles, 1, 0, 0);

```

38.8 Extensions and Optimizations

- **Texture Mapping:** Use textures to render particles with varying shapes and appearances.
- **Instancing:** Combine particle systems with instanced rendering for more complex effects.
- **Collision Detection:** Add collision detection to make particles interact with the environment.

38.9 Summary

In this chapter, you implemented a GPU-based particle system using Vulkan. This project demonstrated the use of compute shaders for real-time physics simulations and their integration with graphics pipelines for rendering.

39

Simulate Fluid Dynamics: Implement 2D fluid simulation

39.1 Introduction

Fluid simulations are widely used in computer graphics to create realistic visual effects such as water, smoke, and fire. Implementing a 2D fluid simulation using Vulkan allows us to take advantage of GPU parallelism for high-performance, real-time simulations. In this chapter, we will use a compute shader to simulate 2D fluid dynamics based on the Navier-Stokes equations, visualizing the results with a texture updated in real-time.

39.2 Overview of Fluid Dynamics Simulation

Simulating fluid dynamics involves:

1. **Velocity Field:** Tracks the motion of fluid particles in 2D space.
2. **Pressure Field:** Ensures the incompressibility of the fluid.
3. **Advection, Diffusion, and Projection:** Steps to update the velocity and pressure fields over time.
4. **Visualization:** Rendering the velocity or pressure fields as textures.

We use a grid-based method where each cell in the grid represents a portion of the fluid. The compute shader updates this grid every frame.

39.3 Step 1: Set Up Grid Buffers

The fluid simulation operates on a grid. Each cell stores properties such as velocity, pressure, and density.

```

1  // Define grid resolution
2  const uint32_t gridWidth = 128;
3  const uint32_t gridHeight = 128;
4
5  // Buffer to store velocity field
6  VkBuffer velocityBuffer;
7  VkBufferCreateInfo velocityBufferInfo{};
8  velocityBufferInfo.size = gridWidth * gridHeight * sizeof(vec2); // vec2 for velocity
9  velocityBufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
10 velocityBufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
11
12 vkCreateBuffer(device, &velocityBufferInfo, nullptr, &velocityBuffer);
13
14 // Buffer to store pressure field
15 VkBuffer pressureBuffer;
16 VkBufferCreateInfo pressureBufferInfo{};
17 pressureBufferInfo.size = gridWidth * gridHeight * sizeof(float); // float for pressure
18 pressureBufferInfo.usage = VK_BUFFER_USAGE_STORAGE_BUFFER_BIT;
19
20 vkCreateBuffer(device, &pressureBufferInfo, nullptr, &pressureBuffer);
21
22 // Allocate and bind memory for both buffers
23 VkMemoryRequirements memRequirements;
24 vkGetBufferMemoryRequirements(device, velocityBuffer, &memRequirements);
25 allocateAndBindBufferMemory(velocityBuffer, memRequirements);
26
27 vkGetBufferMemoryRequirements(device, pressureBuffer, &memRequirements);
28 allocateAndBindBufferMemory(pressureBuffer, memRequirements);

```

39.4 Step 2: Write the Compute Shader for Fluid Simulation

The compute shader performs the steps of advection, diffusion, and projection to simulate fluid motion.

```

1  // GLSL Compute Shader - Fluid Simulation
2  #version 450
3
4  layout(local_size_x = 16, local_size_y = 16) in;
5
6  layout(std430, binding = 0) buffer Velocity {
7      vec2 velocities[];
8  };
9
10 layout(std430, binding = 1) buffer Pressure {
11     float pressures[];
12 };
13
14 layout(push_constant) uniform PushConstants {
15     float deltaTime; // Time step
16     float diffusion; // Diffusion coefficient
17     float viscosity; // Viscosity coefficient
18     vec2 gridSize; // Grid dimensions
19 } pc;
20
21 ivec2 wrapCoords(ivec2 coords, ivec2 gridSize) {

```

```

22     return ivec2((coords.x + gridSize.x) % gridSize.x, (coords.y + gridSize.y) % gridSize.y);
23 }
24
25 void advect(ivec2 coord, vec2 velocity, vec2 gridSize) {
26     vec2 gridPos = vec2(coord) - velocity * pc.deltaTime;
27     ivec2 sampleCoord = wrapCoords(ivec2(floor(gridPos)), ivec2(gridSize));
28     velocities[coord.y * int(gridSize.x) + coord.x] = velocities[sampleCoord.y * int(gridSize.x) + sampleCoord.
    ↪ x];
29 }
30
31 void main() {
32     ivec2 coord = ivec2(gl_GlobalInvocationID.xy);
33     if (coord.x >= pc.gridSize.x || coord.y >= pc.gridSize.y) return;
34
35     // Advection step
36     vec2 velocity = velocities[coord.y * int(pc.gridSize.x) + coord.x];
37     advect(coord, velocity, pc.gridSize);
38
39     // (Further steps like diffusion and projection can be added here)
40 }

```

39.5 Step 3: Visualize the Simulation

We visualize the velocity or pressure field by rendering it as a texture. The texture is updated with the results of the compute shader.

```

1  // Create an image for visualization
2  VkImage outputImage;
3  VkImageCreateInfo imageInfo{};
4  imageInfo.imageType = VK_IMAGE_TYPE_2D;
5  imageInfo.extent.width = gridWidth;
6  imageInfo.extent.height = gridHeight;
7  imageInfo.extent.depth = 1;
8  imageInfo.mipLevels = 1;
9  imageInfo.arrayLayers = 1;
10 imageInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
11 imageInfo.usage = VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
12
13 vkCreateImage(device, &imageInfo, nullptr, &outputImage);
14
15 // Update image with compute shader output
16 VkDescriptorImageInfo imageDescriptor{};
17 imageDescriptor.imageView = outputImageView;
18 imageDescriptor.imageLayout = VK_IMAGE_LAYOUT_GENERAL;
19
20 VkWriteDescriptorSet writeDescriptor{};
21 writeDescriptor.dstSet = descriptorSet;
22 writeDescriptor.dstBinding = 2; // Assuming binding 2 is for the output image
23 writeDescriptor.descriptorType = VK_DESCRIPTOR_TYPE_STORAGE_IMAGE;
24 writeDescriptor.pImageInfo = &imageDescriptor;
25
26 vkUpdateDescriptorSets(device, 1, &writeDescriptor, 0, nullptr);

```

39.6 Step 4: Combine Simulation and Rendering

The rendering loop combines the compute shader for simulation with a fragment shader for visualization.

```

1 // Dispatch the compute shader
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdDispatch(commandBuffer, (gridWidth + 15) / 16, (gridHeight + 15) / 16, 1);
4
5 // Render the texture
6 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
7 vkCmdDraw(commandBuffer, 6, 1, 0, 0); // Full-screen quad

```

39.7 Extensions and Optimizations

- **Interactive Forces:** Add external forces, such as user interactions, to influence the fluid.
- **Obstacles:** Integrate static or dynamic obstacles into the simulation.
- **3D Simulation:** Extend the implementation to 3D for volumetric fluid effects.

39.8 Summary

In this chapter, you implemented a basic 2D fluid simulation using Vulkan compute shaders. This project introduced grid-based fluid simulation techniques and their visualization.

40

Generate a Procedural Heightmap: Create terrains with compute shaders

40.1 Introduction

Heightmaps are a popular method for representing terrain in computer graphics. A heightmap is a grid of values, where each value represents the elevation of a point on the terrain. Procedurally generating heightmaps using compute shaders allows for the creation of detailed and dynamic terrains at runtime. In this chapter, we will use Vulkan compute shaders to generate a procedural heightmap, store it as a texture, and render it as a 3D terrain.

40.2 Overview of Heightmap Generation

To generate a procedural heightmap, we use algorithms such as:

- **Perlin Noise** or **Simplex Noise**: Generates smooth, natural-looking patterns.
- **Fractal Noise**: Combines multiple layers of noise for more detail.
- **Random Walk**: Creates rugged, uneven terrains.
- **Voronoi Patterns**: Used for stylized terrains or structures.

Once generated, the heightmap is stored in a texture and sampled in the vertex shader to deform a flat plane into a 3D terrain.

40.3 Step 1: Set Up the Heightmap Texture

The heightmap is represented as a 2D texture. We allocate an image to store the heightmap values.

```

1  // Create the heightmap image
2  VkImage heightmapImage;
3  VkImageCreateInfo imageInfo{};
4  imageInfo.imageType = VK_IMAGE_TYPE_2D;
5  imageInfo.extent.width = 512; // Heightmap resolution
6  imageInfo.extent.height = 512;
7  imageInfo.extent.depth = 1;
8  imageInfo.mipLevels = 1;
9  imageInfo.arrayLayers = 1;
10 imageInfo.format = VK_FORMAT_R32_SFLOAT; // Single float per pixel
11 imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
12 imageInfo.usage = VK_IMAGE_USAGE_STORAGE_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
13
14 vkCreateImage(device, &imageInfo, nullptr, &heightmapImage);
15
16 // Allocate and bind memory
17 VkMemoryRequirements memRequirements;
18 vkGetImageMemoryRequirements(device, heightmapImage, &memRequirements);
19 allocateAndBindImageMemory(heightmapImage, memRequirements);

```

The heightmap is initialized with an appropriate layout for compute shader operations.

```

1  // Transition image to GENERAL layout for compute access
2  transitionImageLayout(commandBuffer, heightmapImage, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_GENERAL);

```

40.4 Step 2: Write the Compute Shader for Heightmap Generation

We use a compute shader to fill the heightmap texture with procedural data.

```

1  // GLSL Compute Shader - Generate Heightmap
2  #version 450
3
4  layout(local_size_x = 16, local_size_y = 16) in;
5
6  layout(binding = 0, rgba32f) uniform image2D heightmap;
7
8  float hash(vec2 p) {
9      return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
10 }
11
12 float perlinNoise(vec2 p) {
13     vec2 i = floor(p);
14     vec2 f = fract(p);
15
16     float a = hash(i);
17     float b = hash(i + vec2(1.0, 0.0));
18     float c = hash(i + vec2(0.0, 1.0));
19     float d = hash(i + vec2(1.0, 1.0));
20
21     vec2 u = f * f * (3.0 - 2.0 * f);
22
23     return mix(a, b, u.x) +
24         (mix(c, d, u.x) - mix(a, b, u.x)) * u.y;
25 }
26

```

```

27 void main() {
28     ivec2 coord = ivec2(gl_GlobalInvocationID.xy);
29     vec2 uv = vec2(coord) / imageSize(heightmap);
30
31     // Generate fractal noise
32     float frequency = 4.0;
33     float amplitude = 1.0;
34     float height = 0.0;
35
36     for (int i = 0; i < 5; ++i) {
37         height += amplitude * perlinNoise(uv * frequency);
38         frequency *= 2.0;
39         amplitude *= 0.5;
40     }
41
42     imageStore(heightmap, coord, vec4(height, 0.0, 0.0, 0.0));
43 }

```

40.5 Step 3: Visualize the Heightmap as a 3D Terrain

The generated heightmap is sampled in a vertex shader to displace vertices of a flat plane, creating a 3D terrain.

```

1 // GLSL Vertex Shader - Terrain Rendering
2 #version 450
3
4 layout(location = 0) in vec3 inPosition;
5 layout(location = 1) in vec2 inUV;
6
7 layout(set = 0, binding = 0) uniform sampler2D heightmap;
8
9 layout(push_constant) uniform PushConstants {
10     float heightScale;
11 } pc;
12
13 out vec2 fragUV;
14
15 void main() {
16     float height = texture(heightmap, inUV).r;
17     vec3 displacedPosition = inPosition + vec3(0.0, height * pc.heightScale, 0.0);
18     gl_Position = projMatrix * viewMatrix * modelMatrix * vec4(displacedPosition, 1.0);
19     fragUV = inUV;
20 }

```

40.6 Step 4: Combine Compute and Graphics Pipelines

The rendering loop integrates the compute shader for heightmap generation and the graphics pipeline for rendering.

```

1 // Dispatch compute shader to generate the heightmap
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdDispatch(commandBuffer, (512 + 15) / 16, (512 + 15) / 16, 1);
4
5 // Render the terrain
6 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
7 vkCmdDrawIndexed(commandBuffer, indexCount, 1, 0, 0, 0);

```

40.7 Extensions and Optimizations

- **Dynamic Terrain:** Update the heightmap in real-time to simulate dynamic terrain deformation.
- **Detail Levels:** Use multiple heightmaps for different levels of detail.
- **Lighting and Shading:** Add advanced lighting techniques to enhance terrain appearance.
- **Texture Blending:** Combine the heightmap with textures for realistic terrain rendering.

40.8 Summary

This chapter demonstrated how to generate a procedural heightmap using Vulkan compute shaders and visualize it as a 3D terrain. You explored techniques for procedural noise generation and the integration of compute and graphics pipelines.

Part V

Optimization Projects

41

Use Pipeline Caching: Reduce pipeline creation overhead

41.1 Introduction

Pipeline creation in Vulkan can be an expensive operation due to the complexity of the graphics and compute pipelines. Repeatedly creating pipelines with similar configurations can lead to unnecessary overhead, especially in performance-critical applications like real-time rendering or simulations. Vulkan provides **pipeline caching**, which allows you to store and reuse pipeline states, reducing the cost of creating pipelines. In this chapter, we will explore how to implement pipeline caching and integrate it into your Vulkan application.

41.2 Overview of Pipeline Caching

Pipeline caching involves:

1. **Pipeline Cache Object**: A Vulkan object that stores compiled pipeline data.
2. **Serialization**: Saving the pipeline cache to disk for reuse in subsequent runs.
3. **Reusability**: Sharing pipeline cache data between multiple pipelines to accelerate creation.

Pipeline caching is particularly useful in applications where:

- Many pipelines with similar configurations are used.
- Pipelines are created dynamically, such as in applications with procedural graphics.

Step 1: Create a Pipeline Cache Object

The pipeline cache object is created during the initialization phase of your Vulkan application.

```

1 // Create a pipeline cache object
2 VkPipelineCache pipelineCache;
3 VkPipelineCacheCreateInfo cacheInfo{};
4 cacheInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
5 cacheInfo.initialDataSize = 0; // No initial data
6 cacheInfo.pInitialData = nullptr;
7
8 VkResult result = vkCreatePipelineCache(device, &cacheInfo, nullptr, &pipelineCache);
9 if (result != VK_SUCCESS) {
10     throw std::runtime_error("Failed to create pipeline cache");
11 }

```

The `pipelineCache` object is passed during pipeline creation to leverage cached data.

Step 2: Use the Pipeline Cache During Pipeline Creation

When creating a pipeline, you can specify the pipeline cache object in the `vkCreateGraphicsPipelines` or `vkCreateComputePipelines` function.

```

1 // Example: Creating a graphics pipeline with pipeline caching
2 VkGraphicsPipelineCreateInfo pipelineInfo{};
3 pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
4 // ... (set other pipeline configurations)
5
6 VkPipeline graphicsPipeline;
7 result = vkCreateGraphicsPipelines(device, pipelineCache, 1, &pipelineInfo, nullptr, &graphicsPipeline);
8 if (result != VK_SUCCESS) {
9     throw std::runtime_error("Failed to create graphics pipeline with cache");
10 }

```

Using the cache object ensures that any reusable pipeline data is leveraged, reducing the creation time.

Step 3: Serialize and Load Pipeline Cache Data

To maximize the benefits of pipeline caching, you can serialize the cache data to disk and reload it in subsequent runs of the application.

41.2.1 Save Pipeline Cache Data

At the end of the application's runtime, retrieve the cache data and save it to a file.

```

1 // Retrieve the size of the pipeline cache data
2 size_t cacheSize = 0;
3 vkGetPipelineCacheData(device, pipelineCache, &cacheSize, nullptr);
4
5 // Allocate a buffer to hold the cache data

```

```

6  std::vector<uint8_t> cacheData(cacheSize);
7  vkGetPipelineCacheData(device, pipelineCache, &cacheSize, cacheData.data());
8
9  // Write the cache data to a file
10 std::ofstream cacheFile("pipeline_cache.bin", std::ios::binary);
11 cacheFile.write(reinterpret_cast<const char*>(cacheData.data()), cacheSize);
12 cacheFile.close();

```

41.2.2 Load Pipeline Cache Data

At application startup, load the saved cache data from disk to initialize the pipeline cache object.

```

1  // Load cache data from file
2  std::ifstream cacheFile("pipeline_cache.bin", std::ios::binary | std::ios::ate);
3  size_t cacheSize = cacheFile.tellg();
4  cacheFile.seekg(0, std::ios::beg);
5
6  std::vector<uint8_t> cacheData(cacheSize);
7  cacheFile.read(reinterpret_cast<char*>(cacheData.data()), cacheSize);
8  cacheFile.close();
9
10 // Create pipeline cache with initial data
11 VkPipelineCacheCreateInfo cacheInfo{};
12 cacheInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO;
13 cacheInfo.initialDataSize = cacheSize;
14 cacheInfo.pInitialData = cacheData.data();
15
16 vkCreatePipelineCache(device, &cacheInfo, nullptr, &pipelineCache);

```

Step 4: Share Cache Data Between Pipelines

If your application uses multiple pipelines, they can share the same pipeline cache object. Vulkan will automatically use cached data to speed up creation.

```

1  // Create another pipeline using the same pipeline cache
2  VkComputePipelineCreateInfo computePipelineInfo{};
3  computePipelineInfo.sType = VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO;
4  // ... (set other compute pipeline configurations)
5
6  VkPipeline computePipeline;
7  result = vkCreateComputePipelines(device, pipelineCache, 1, &computePipelineInfo, nullptr, &computePipeline);
8  if (result != VK_SUCCESS) {
9      throw std::runtime_error("Failed to create compute pipeline with cache");
10 }

```

Benefits and Considerations

Using pipeline caching provides several benefits:

- **Reduced Pipeline Creation Time:** Significantly improves performance when creating pipelines with similar configurations.
- **Cross-Session Persistence:** Serialization allows reuse of pipeline states across application runs.

However, there are a few considerations:

- Pipeline cache data is not portable across different Vulkan implementations or driver versions.
- Always handle cases where the cache data might not be available or valid.

Summary

Pipeline caching is a powerful feature that helps reduce the overhead of creating Vulkan pipelines. Reusing cached data and persisting it across application runs, you can achieve significant performance improvements. In this chapter, you learned how to create, serialize, and reuse a pipeline cache object.

42

Add Command Buffer Reuse: Reuse recorded command buffers for efficiency

42.1 Introduction

In Vulkan, command buffers are used to record a sequence of GPU commands, such as rendering and compute operations. Recording commands can be a performance-intensive operation, especially when it involves repetitive sequences, such as rendering static geometry or applying fixed post-processing effects. To improve efficiency, Vulkan allows you to reuse command buffers, reducing CPU overhead by avoiding repeated recording. This chapter explores how to set up and implement command buffer reuse in Vulkan.

42.2 Overview of Command Buffer Reuse

Command buffers can be recorded once and executed multiple times, making them ideal for:

- **Static Scenes:** Rendering static objects or backgrounds that don't change across frames.
- **Precomputed Pipelines:** Fixed pipelines for operations like post-processing or off-screen rendering.
- **Compute Tasks:** Repetitive operations in compute shaders.

There are two main types of command buffers:

- **Primary Command Buffers:** Can be directly submitted to the queue.
- **Secondary Command Buffers:** Recorded to be executed from within a primary command buffer.

Reusing command buffers involves carefully managing their lifecycle to ensure validity and synchronization.

42.3 Step 1: Record the Command Buffer

Command buffers are recorded once with the desired sequence of commands.

```

1 // Allocate a command buffer
2 VkCommandBuffer commandBuffer;
3 VkCommandBufferAllocateInfo allocateInfo{};
4 allocateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
5 allocateInfo.commandPool = commandPool;
6 allocateInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
7 allocateInfo.commandBufferCount = 1;
8
9 vkAllocateCommandBuffers(device, &allocateInfo, &commandBuffer);
10
11 // Begin recording
12 VkCommandBufferBeginInfo beginInfo{};
13 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
14 beginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT; // Enable reuse
15
16 vkBeginCommandBuffer(commandBuffer, &beginInfo);
17
18 // Example: Render pass and draw commands
19 VkRenderPassBeginInfo renderPassInfo{};
20 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
21 renderPassInfo.renderPass = renderPass;
22 renderPassInfo.framebuffer = framebuffer;
23 renderPassInfo.renderArea.extent = swapchainExtent;
24
25 vkCmdBeginRenderPass(commandBuffer, &renderPassInfo, VK_SUBPASS_CONTENTS_INLINE);
26 // Record drawing commands here
27 vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);
28 vkCmdEndRenderPass(commandBuffer);
29
30 // End recording
31 vkEndCommandBuffer(commandBuffer);

```

The `VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT` flag allows the command buffer to be submitted multiple times without re-recording.

42.4 Step 2: Submit the Command Buffer for Execution

Once recorded, the command buffer can be submitted to a queue repeatedly.

```

1 // Submit the recorded command buffer
2 VkSubmitInfo submitInfo{};
3 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
4 submitInfo.commandBufferCount = 1;
5 submitInfo.pCommandBuffers = &commandBuffer;
6
7 vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
8
9 // Wait for the queue to finish execution (optional)
10 vkQueueWaitIdle(graphicsQueue);

```

This process eliminates the need to re-record commands for each frame.

42.5 Step 3: Use Secondary Command Buffers for Modularization

For complex scenes or tasks, use secondary command buffers to modularize rendering and improve performance.

```

1  // Allocate a secondary command buffer
2  VkCommandBuffer secondaryCommandBuffer;
3  VkCommandBufferAllocateInfo allocateInfo{};
4  allocateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
5  allocateInfo.commandPool = commandPool;
6  allocateInfo.level = VK_COMMAND_BUFFER_LEVEL_SECONDARY;
7  allocateInfo.commandBufferCount = 1;
8
9  vkAllocateCommandBuffers(device, &allocateInfo, &secondaryCommandBuffer);
10
11 // Begin recording secondary command buffer
12 VkCommandBufferInheritanceInfo inheritanceInfo{};
13 inheritanceInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_INHERITANCE_INFO;
14 inheritanceInfo.renderPass = renderPass;
15 inheritanceInfo.framebuffer = framebuffer;
16
17 VkCommandBufferBeginInfo beginInfo{};
18 beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
19 beginInfo.flags = VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT;
20 beginInfo.pInheritanceInfo = &inheritanceInfo;
21
22 vkBeginCommandBuffer(secondaryCommandBuffer, &beginInfo);
23
24 // Record commands for a static object
25 vkCmdDraw(secondaryCommandBuffer, staticVertexCount, 1, 0, 0);
26
27 // End recording
28 vkEndCommandBuffer(secondaryCommandBuffer);
29
30 // Use the secondary command buffer in the primary command buffer
31 vkCmdExecuteCommands(primaryCommandBuffer, 1, &secondaryCommandBuffer);

```

Secondary command buffers provide flexibility and enable parallel recording of commands.

42.6 Step 4: Manage Synchronization

When reusing command buffers, ensure proper synchronization between GPU operations to avoid data hazards.

```

1  // Synchronize GPU operations using fences or semaphores
2  VkFence fence;
3  VkFenceCreateInfo fenceInfo{};
4  fenceInfo.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
5
6  vkCreateFence(device, &fenceInfo, nullptr, &fence);
7  vkQueueSubmit(graphicsQueue, 1, &submitInfo, fence);
8
9  // Wait for the GPU to finish execution
10 vkWaitForFences(device, 1, &fence, VK_TRUE, UINT64_MAX);
11 vkDestroyFence(device, fence, nullptr);

```


Reused command buffers must always operate on valid data and resources, requiring careful synchronization and resource management.

42.7 Benefits and Considerations

Command buffer reuse provides:

- **Reduced CPU Overhead:** Minimizes the need for re-recording repetitive commands.
- **Performance Boost:** Speeds up rendering for static or modular scenes.
- **Modular Design:** Secondary command buffers improve code organization and parallelism.

However, there are some considerations:

- Reused command buffers should only contain commands that do not depend on frame-specific data.
- Any resource or state changes between frames might require re-recording.

42.8 Summary

In this chapter, we explored how to implement command buffer reuse in Vulkan, including recording reusable command buffers, using secondary command buffers for modularity, and ensuring proper synchronization. By leveraging command buffer reuse, you can significantly improve the efficiency and performance of your Vulkan application.

43

Implement Instanced Rendering: Draw multiple objects with fewer draw calls

43.1 Introduction

Instanced rendering is a powerful optimization technique in modern graphics APIs like Vulkan. Instead of issuing multiple draw calls for identical or similar objects, you can draw them in a single call by providing instance-specific data. This reduces CPU overhead and improves rendering performance, especially in scenes with many repeated objects such as trees, buildings, or particles.

This chapter demonstrates how to implement instanced rendering in Vulkan, from setting up instance data to modifying shaders for per-instance customization.

43.2 What is Instanced Rendering?

Instanced rendering allows you to:

- Render multiple copies of a geometry (e.g., a triangle or a 3D model) with variations such as position, color, or size.
- Use a single draw call with instance data passed to the GPU, avoiding the need for multiple `vkCmdDraw` calls.

In Vulkan, instanced rendering is implemented using specialized draw commands like `vkCmdDraw` or `vkCmdDrawIndexed` with instance count and instance data buffers.

43.3 Step 1: Prepare Instance Data

Instance data includes per-instance attributes such as transformation matrices, colors, or other properties. These are stored in a separate buffer alongside vertex data.

For this example, we will render multiple triangles with unique positions and colors.

```

1  // Define instance data structure
2  struct InstanceData {
3      glm::vec3 position;
4      glm::vec3 color;
5  };
6
7  // Example instance data
8  std::vector<InstanceData> instances = {
9      {{-2.0f, 0.0f, 0.0f}, {1.0f, 0.0f, 0.0f}}, // Red triangle
10     {{0.0f, 0.0f, 0.0f}, {0.0f, 1.0f, 0.0f}}, // Green triangle
11     {{2.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 1.0f}} // Blue triangle
12 };
13
14 // Create a Vulkan buffer for instance data
15 VkBuffer instanceBuffer;
16 VkDeviceMemory instanceBufferMemory;
17 // Use your existing Vulkan buffer creation functions
18 createBuffer(instanceDataSize, VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
19              VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
20              instanceBuffer, instanceBufferMemory);
21
22 // Upload instance data to the buffer
23 void* data;
24 vkMapMemory(device, instanceBufferMemory, 0, instanceDataSize, 0, &data);
25 memcpy(data, instances.data(), instanceDataSize);
26 vkUnmapMemory(device, instanceBufferMemory);

```

43.4 Step 2: Update the Vertex Shader

Modify the vertex shader to support instance-specific attributes. Instance data is accessed using the `instanceIndex` input.

Here's an example GLSL vertex shader for instanced rendering:

```

1  // Vertex shader (instanced rendering)
2  #version 450
3
4  layout(location = 0) in vec3 inPosition; // Vertex position
5  layout(location = 1) in vec3 inColor;    // Per-vertex color
6  layout(location = 2) in vec3 instancePosition; // Per-instance position
7  layout(location = 3) in vec3 instanceColor; // Per-instance color
8
9  layout(location = 0) out vec3 fragColor;
10
11 void main() {
12     // Add instance position to vertex position
13     gl_Position = vec4(inPosition + instancePosition, 1.0);
14     fragColor = instanceColor;
15 }

```

This shader combines per-vertex and per-instance data to position and color each instance

uniquely.

43.5 Step 3: Bind Multiple Buffers

Bind both the vertex and instance data buffers during the draw process.

```
1 // Bind vertex and instance buffers
2 VkBuffer buffers[] = {vertexBuffer, instanceBuffer};
3 VkDeviceSize offsets[] = {0, 0};
4 vkCmdBindVertexBuffers(commandBuffer, 0, 2, buffers, offsets);
```

Here, the first buffer (vertex data) is bound to `binding = 0`, and the second buffer (instance data) is bound to `binding = 1`. The shader will access these buffers using their corresponding bindings.

43.6 Step 4: Issue the Draw Call

Use the `vkCmdDraw` function with an instance count to render all instances in a single call.

```
1 // Draw instanced geometry
2 uint32_t vertexCount = 3; // Number of vertices per triangle
3 uint32_t instanceCount = static_cast<uint32_t>(instances.size()); // Number of instances
4 vkCmdDraw(commandBuffer, vertexCount, instanceCount, 0, 0);
```

The `instanceCount` parameter specifies the number of instances to render. The GPU iterates over all instances and uses the instance-specific data for each one.

43.7 Best Practices for Instanced Rendering

1. **Minimize Data Transfers:** Use device-local memory for instance data to reduce CPU-GPU data transfer overhead.
2. **Dynamic Updates:** For dynamic instances, map and update the instance buffer as needed.
3. **Batch Instances:** Group instances with similar rendering properties (e.g., the same shader or texture) to minimize pipeline state changes.

43.8 Summary

Instanced rendering allows you to render multiple objects with fewer draw calls, significantly improving performance in scenes with repeated geometry. By combining per-vertex and per-instance data in shaders, you can achieve dynamic variations while leveraging the efficiency of Vulkan. In this chapter, you learned how to set up instance data, modify shaders, and issue instanced draw calls.

44

Optimize Descriptor Updates: Batch descriptor set updates

44.1 Introduction

Descriptors in Vulkan are used to bind resources such as textures, buffers, and samplers to shaders. Updating descriptor sets is a critical operation that can impact performance if not managed efficiently. In this chapter, we explore techniques for batching descriptor updates to minimize overhead and improve runtime efficiency, especially in applications that require frequent descriptor changes.

44.2 Understanding Descriptor Updates

Descriptor updates involve transferring resource information (e.g., buffer bindings or texture information) to descriptor sets, which are then used by the GPU during rendering or compute operations. This process typically includes:

- Creating or updating descriptors with resource bindings.
- Writing descriptor data to descriptor sets.

Frequent descriptor updates can be expensive, as they involve CPU-GPU synchronization and command buffer re-recording. Optimizing these updates is essential for maintaining high performance in Vulkan applications.

44.3 Step 1: Batch Descriptor Writes and Copies

The Vulkan API allows multiple descriptor updates to be grouped into a single operation using the `vkUpdateDescriptorSets` function. By batching writes and copies, you can reduce the number of API calls and minimize synchronization overhead.

```

1  // Prepare multiple descriptor writes
2  std::vector<VkWriteDescriptorSet> descriptorWrites;
3
4  VkDescriptorBufferInfo bufferInfo{};
5  bufferInfo.buffer = uniformBuffer;
6  bufferInfo.offset = 0;
7  bufferInfo.range = sizeof(UniformBufferObject);
8
9  VkWriteDescriptorSet uniformWrite{};
10 uniformWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
11 uniformWrite.dstSet = descriptorSet;
12 uniformWrite.dstBinding = 0;
13 uniformWrite.dstArrayElement = 0;
14 uniformWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
15 uniformWrite.descriptorCount = 1;
16 uniformWrite.pBufferInfo = &bufferInfo;
17 descriptorWrites.push_back(uniformWrite);
18
19 VkDescriptorImageInfo imageInfo{};
20 imageInfo.imageView = textureImageView;
21 imageInfo.sampler = textureSampler;
22 imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
23
24 VkWriteDescriptorSet textureWrite{};
25 textureWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
26 textureWrite.dstSet = descriptorSet;
27 textureWrite.dstBinding = 1;
28 textureWrite.dstArrayElement = 0;
29 textureWrite.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
30 textureWrite.descriptorCount = 1;
31 textureWrite.pImageInfo = &imageInfo;
32 descriptorWrites.push_back(textureWrite);
33
34 // Update all descriptors in a single call
35 vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
36                       descriptorWrites.data(), 0, nullptr);

```

Batching updates like this reduces the overhead of issuing separate `vkUpdateDescriptorSets` calls for each descriptor.

44.4 Step 2: Use Dynamic Descriptors for Flexibility

Dynamic descriptors allow you to bind a single descriptor set with offsets to access different resources, reducing the need for frequent descriptor updates. For example, dynamic uniform buffers can be used to store per-object data, allowing the same descriptor set to be reused across multiple objects.

```

1  // Define a dynamic descriptor buffer
2  VkDescriptorBufferInfo dynamicBufferInfo{};
3  dynamicBufferInfo.buffer = dynamicUniformBuffer;
4  dynamicBufferInfo.offset = 0; // Offset will be updated dynamically
5  dynamicBufferInfo.range = sizeof(UniformBufferObject);

```

```

6
7 VkWriteDescriptorSet dynamicWrite{};
8 dynamicWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9 dynamicWrite.dstSet = descriptorSet;
10 dynamicWrite.dstBinding = 0;
11 dynamicWrite.dstArrayElement = 0;
12 dynamicWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
13 dynamicWrite.descriptorCount = 1;
14 dynamicWrite.pBufferInfo = &dynamicBufferInfo;
15
16 // Update dynamic descriptor
17 vkUpdateDescriptorSets(device, 1, &dynamicWrite, 0, nullptr);

```

When issuing a draw command, specify the dynamic offset for the descriptor:

```

1 // Bind descriptor sets with dynamic offsets
2 uint32_t dynamicOffset = currentObjectIndex * dynamicAlignment;
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
4                          pipelineLayout, 0, 1, &descriptorSet, 1, &dynamicOffset);

```

This approach minimizes the number of descriptor updates needed, as you can reuse the same descriptor set across multiple objects.

44.5 Step 3: Use Descriptor Indexing for Large Resource Sets

If your application uses a large number of resources (e.g., textures or buffers), consider using descriptor indexing. Descriptor indexing allows shaders to dynamically index into a large array of descriptors, reducing the need for frequent descriptor updates.

Enable the required extensions and features during Vulkan instance and device creation:

```

1 // Enable descriptor indexing features
2 VkPhysicalDeviceDescriptorIndexingFeatures indexingFeatures{};
3 indexingFeatures.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_DESCRIPTOR_INDEXING_FEATURES;
4 indexingFeatures.shaderSampledImageArrayNonUniformIndexing = VK_TRUE;
5
6 // Add to VkDeviceCreateInfo
7 VkDeviceCreateInfo createInfo{};
8 createInfo.pNext = &indexingFeatures;

```

In the shader, use a non-uniform index to access a descriptor:

```

1 // GLSL shader with descriptor indexing
2 layout(set = 0, binding = 0) uniform sampler2D textures[];
3
4 void main() {
5     vec4 color = texture(textures[nonUniformIndex], texCoords);
6 }

```

This approach allows you to bind a large descriptor array once and access resources dynamically in the shader.

44.6 Step 4: Optimize Descriptor Pool Allocation

Efficient descriptor pool management ensures that descriptor updates remain fast and predictable. Allocate descriptor sets in bulk and recycle unused sets when possible.

```

1 // Create a large descriptor pool
2 VkDescriptorPoolSize poolSize{};
3 poolSize.type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
4 poolSize.descriptorCount = 100;
5
6 VkDescriptorPoolCreateInfo poolInfo{};
7 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
8 poolInfo.poolSizeCount = 1;
9 poolInfo.pPoolSizes = &poolSize;
10 poolInfo.maxSets = 50;
11
12 VkDescriptorPool descriptorPool;
13 vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);

```

Recycle descriptor sets to avoid recreating them frequently.

44.7 Summary

Optimizing descriptor updates is essential for achieving high-performance rendering in Vulkan. By batching updates, leveraging dynamic descriptors, and using descriptor indexing, you can minimize CPU overhead and improve the efficiency of your rendering pipeline. This chapter covered various techniques for managing descriptors effectively, setting the stage for advanced resource management strategies in upcoming chapters.

45

Use Dynamic Buffers: Optimize buffer memory usage

45.1 Introduction

Dynamic buffers in Vulkan offer an efficient way to manage frequently changing data, such as per-frame transformations or per-object data in a large scene. By using a single buffer with dynamically adjustable offsets, you can reduce memory fragmentation, avoid frequent buffer re-creation, and streamline resource binding. In this chapter, we explore how to implement and optimize dynamic buffers for various Vulkan applications.

45.2 What are Dynamic Buffers?

Dynamic buffers use a single Vulkan buffer to store multiple data segments, with each segment accessed via an offset. This eliminates the need to create separate buffers for each resource, reducing overhead and improving resource management. Vulkan supports dynamic buffers through the `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` and `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC` descriptor types.

Dynamic buffers are especially useful for:

- Storing per-frame or per-object uniform data.
- Reducing the number of descriptor sets by reusing the same set with different offsets.
- Optimizing memory usage by consolidating multiple small allocations into a single large buffer.

45.3 Step 1: Create a Dynamic Buffer

A dynamic buffer is a standard Vulkan buffer, but its data layout and usage are carefully structured to support dynamic offsets.

```

1 // Determine the alignment requirement for dynamic uniform buffers
2 VkPhysicalDeviceProperties deviceProperties;
3 vkGetPhysicalDeviceProperties(physicalDevice, &deviceProperties);
4 size_t minAlignment = deviceProperties.limits.minUniformBufferOffsetAlignment;
5
6 // Align data size to the required alignment
7 size_t dataSize = sizeof(UniformBufferObject);
8 size_t alignedDataSize = (dataSize + minAlignment - 1) & ~(minAlignment - 1);
9
10 // Create a buffer large enough to hold multiple objects
11 VkDeviceSize bufferSize = alignedDataSize * MAX_OBJECTS;
12
13 createBuffer(bufferSize, VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
14              VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
15              dynamicBuffer, dynamicBufferMemory);

```

The `alignedDataSize` ensures that each object's data is correctly aligned according to the GPU's requirements. The buffer size is a multiple of the aligned size to accommodate multiple objects.

45.4 Step 2: Update the Buffer with Per-Object Data

Map the buffer memory and write per-object data at the appropriate offsets.

```

1 void* data;
2 vkMapMemory(device, dynamicBufferMemory, 0, bufferSize, 0, &data);
3
4 for (size_t i = 0; i < objectCount; ++i) {
5     UniformBufferObject ubo = computeUboForObject(i);
6     memcpy(static_cast<char*>(data) + i * alignedDataSize, &ubo, sizeof(ubo));
7 }
8
9 vkUnmapMemory(device, dynamicBufferMemory);

```

Each object's data is placed at an offset calculated as `i * alignedDataSize`, ensuring proper alignment and separation of data within the buffer.

45.5 Step 3: Create and Bind a Dynamic Descriptor

Create a descriptor set that uses the `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC` descriptor type.

```

1 // Define a dynamic descriptor buffer info
2 VkDescriptorBufferInfo bufferInfo{};
3 bufferInfo.buffer = dynamicBuffer;
4 bufferInfo.offset = 0; // Offset will be set dynamically
5 bufferInfo.range = alignedDataSize;
6
7 VkWriteDescriptorSet descriptorWrite{};

```

```
8 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9 descriptorWrite.dstSet = descriptorSet;
10 descriptorWrite.dstBinding = 0;
11 descriptorWrite.dstArrayElement = 0;
12 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC;
13 descriptorWrite.descriptorCount = 1;
14 descriptorWrite.pBufferInfo = &bufferInfo;
15
16 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);
```

The descriptor is configured with a single buffer but does not specify a fixed offset. Instead, the offset is passed dynamically during binding.

45.6 Step 4: Issue Draw Calls with Dynamic Offsets

When issuing draw commands, specify the dynamic offset for the current object. This offset determines which segment of the buffer the GPU will use.

```
1 for (size_t i = 0; i < objectCount; ++i) {
2     uint32_t dynamicOffset = static_cast<uint32_t>(i * alignedDataSize);
3     vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
4                             pipelineLayout, 0, 1, &descriptorSet, 1, &dynamicOffset);
5
6     vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);
7 }
```

The `dynamicOffset` parameter shifts the buffer binding to the appropriate segment for each object, enabling efficient reuse of the same descriptor set.

45.7 Advantages of Dynamic Buffers

1. **Reduced Descriptor Count:** Reuse a single descriptor set across multiple objects or frames, reducing the need to manage large descriptor pools.
2. **Efficient Memory Management:** Consolidate multiple resources into a single buffer, minimizing fragmentation and allocation overhead.
3. **Improved Performance:** Reduce CPU overhead by eliminating frequent descriptor set updates or rebindings.

45.8 Limitations and Considerations

- **Alignment Requirements:** Ensure proper alignment for dynamic buffer offsets as specified by the device properties.
- **Maximum Descriptor Sets:** Be aware of the limits on the number of dynamic descriptor sets and offsets supported by the hardware.
- **Buffer Size:** Allocate a buffer large enough to hold all necessary segments while considering alignment padding.

45.9 Summary

Dynamic buffers are an essential tool for optimizing memory usage and resource management in Vulkan. By structuring data with proper alignment, reusing descriptor sets with dynamic offsets, and batching updates, you can significantly reduce overhead in applications with a high number of objects or frequent updates. This chapter provided a step-by-step guide to implementing dynamic buffers, preparing you for more advanced memory management techniques in Vulkan.

DRAFT
version
0.681

46

Asynchronous Compute and Graphics: Use separate queues for compute and graphics

46.1 Introduction

Vulkan's explicit control over the GPU allows applications to leverage multiple queues for parallel execution of graphics and compute workloads. This capability is especially useful for tasks like post-processing, physics simulation, or AI computations, which can run asynchronously alongside traditional rendering tasks. In this chapter, we will explore how to use separate compute and graphics queues to maximize GPU utilization and improve performance.

46.2 Understanding Queue Families and Parallel Execution

In Vulkan, queue families represent groups of GPU command queues, each capable of specific types of operations, such as graphics, compute, or transfer. Using separate queues for compute and graphics allows the GPU to execute commands from both queues in parallel, provided the workloads are independent.

To take advantage of asynchronous compute and graphics:

1. Identify the available queue families on the physical device.
2. Allocate separate queues for compute and graphics tasks.
3. Synchronize the workloads where dependencies exist using semaphores or fences.

46.3 Step 1: Query and Select Queue Families

Start by querying the physical device for supported queue families and identifying those that support compute and graphics operations.

```

1  // Query the physical device for queue families
2  uint32_t queueFamilyCount = 0;
3  vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, nullptr);
4
5  std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
6  vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueFamilyCount, queueFamilies.data());
7
8  // Find separate compute and graphics queue families
9  int graphicsFamily = -1;
10 int computeFamily = -1;
11
12 for (uint32_t i = 0; i < queueFamilyCount; ++i) {
13     if (queueFamilies[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) {
14         graphicsFamily = i;
15     }
16     if ((queueFamilies[i].queueFlags & VK_QUEUE_COMPUTE_BIT) &&
17         !(queueFamilies[i].queueFlags & VK_QUEUE_GRAPHICS_BIT)) {
18         computeFamily = i;
19     }
20     if (graphicsFamily != -1 && computeFamily != -1) {
21         break;
22     }
23 }
24
25 if (graphicsFamily == -1 || computeFamily == -1) {
26     throw std::runtime_error("Failed to find required queue families!");
27 }

```

Separate queue families for graphics and compute workloads ensure that these tasks can run in parallel.

46.4 Step 2: Create Logical Device with Multiple Queues

When creating the logical device, request both a graphics queue and a compute queue.

```

1  // Specify queue priorities
2  float queuePriority = 1.0f;
3
4  std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
5  VkDeviceQueueCreateInfo graphicsQueueInfo{};
6  graphicsQueueInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
7  graphicsQueueInfo.queueFamilyIndex = graphicsFamily;
8  graphicsQueueInfo.queueCount = 1;
9  graphicsQueueInfo.pQueuePriorities = &queuePriority;
10 queueCreateInfos.push_back(graphicsQueueInfo);
11
12 VkDeviceQueueCreateInfo computeQueueInfo{};
13 computeQueueInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
14 computeQueueInfo.queueFamilyIndex = computeFamily;
15 computeQueueInfo.queueCount = 1;
16 computeQueueInfo.pQueuePriorities = &queuePriority;
17 queueCreateInfos.push_back(computeQueueInfo);
18
19 VkDeviceCreateInfo deviceCreateInfo{};

```

```

20 deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
21 deviceCreateInfo.queueCreateInfoCount = static_cast<uint32_t>(queueCreateInfos.size());
22 deviceCreateInfo.pQueueCreateInfos = queueCreateInfos.data();
23
24 // Create the logical device
25 VkDevice device;
26 if (vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device) != VK_SUCCESS) {
27     throw std::runtime_error("Failed to create logical device!");
28 }
29
30 // Retrieve the queues
31 VkQueue graphicsQueue;
32 VkQueue computeQueue;
33 vkGetDeviceQueue(device, graphicsFamily, 0, &graphicsQueue);
34 vkGetDeviceQueue(device, computeFamily, 0, &computeQueue);

```

Separate queues allow you to submit commands for compute and graphics tasks independently.

46.5 Step 3: Record Command Buffers for Compute and Graphics

Prepare separate command buffers for the compute and graphics queues. For example, a compute task might process data for a particle system, while a graphics task renders the scene.

```

1 // Record compute command buffer
2 VkCommandBuffer computeCmdBuffer;
3 // Begin and record compute commands
4 beginCommandBuffer(computeCmdBuffer);
5 // Record compute operations
6 vkCmdDispatch(computeCmdBuffer, workGroupX, workGroupY, workGroupZ);
7 endCommandBuffer(computeCmdBuffer);
8
9 // Record graphics command buffer
10 VkCommandBuffer graphicsCmdBuffer;
11 // Begin and record graphics commands
12 beginCommandBuffer(graphicsCmdBuffer);
13 // Record rendering operations
14 vkCmdDraw(graphicsCmdBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
15 endCommandBuffer(graphicsCmdBuffer);

```

46.6 Step 4: Synchronize Queues with Semaphores

Use semaphores to synchronize operations between the compute and graphics queues. For example, a compute shader might update a buffer that the graphics pipeline uses for rendering.

```

1 // Create semaphores for synchronization
2 VkSemaphore computeSemaphore;
3 VkSemaphoreCreateInfo semaphoreInfo{};
4 semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
5
6 if (vkCreateSemaphore(device, &semaphoreInfo, nullptr, &computeSemaphore) != VK_SUCCESS) {
7     throw std::runtime_error("Failed to create semaphore!");
8 }
9
10 // Submit compute commands
11 VkSubmitInfo computeSubmitInfo{};
12 computeSubmitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;

```

```

13 computeSubmitInfo.commandBufferCount = 1;
14 computeSubmitInfo.pCommandBuffers = &computeCmdBuffer;
15 computeSubmitInfo.signalSemaphoreCount = 1;
16 computeSubmitInfo.pSignalSemaphores = &computeSemaphore;
17
18 vkQueueSubmit(computeQueue, 1, &computeSubmitInfo, VK_NULL_HANDLE);
19
20 // Submit graphics commands, waiting on the compute semaphore
21 VkPipelineStageFlags waitStages[] = {VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT};
22 VkSubmitInfo graphicsSubmitInfo{};
23 graphicsSubmitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
24 graphicsSubmitInfo.commandBufferCount = 1;
25 graphicsSubmitInfo.pCommandBuffers = &graphicsCmdBuffer;
26 graphicsSubmitInfo.waitSemaphoreCount = 1;
27 graphicsSubmitInfo.pWaitSemaphores = &computeSemaphore;
28 graphicsSubmitInfo.pWaitDstStageMask = waitStages;
29
30 vkQueueSubmit(graphicsQueue, 1, &graphicsSubmitInfo, VK_NULL_HANDLE);

```

46.7 Advantages of Asynchronous Compute and Graphics

1. **Improved GPU Utilization:** By running compute and graphics tasks in parallel, you can maximize GPU efficiency and reduce idle time.
2. **Pipeline Independence:** Offload heavy compute workloads to a separate queue, preventing them from blocking the graphics pipeline.
3. **Scalability:** Suitable for applications with complex processing needs, such as simulations, AI, or real-time physics.

46.8 Limitations and Considerations

- **Queue Family Support:** Not all devices support separate queue families for compute and graphics. Ensure your device supports this feature before implementation.
- **Synchronization Complexity:** Managing dependencies between compute and graphics tasks requires careful synchronization to avoid race conditions or stalling.
- **Memory Barriers:** Use appropriate memory barriers to ensure data consistency when sharing resources between queues.

46.9 Summary

Asynchronous compute and graphics capabilities in Vulkan open up new possibilities for optimizing GPU workloads. By using separate queues for compute and graphics, you can achieve parallelism and improve the performance of resource-intensive applications. This chapter provided a foundation for leveraging multiple queues, setting the stage for advanced workload management in Vulkan.

47

Measure Performance with Queries: Add timestamp queries to measure GPU time

47.1 Introduction

Performance measurement is critical for optimizing Vulkan applications. Vulkan provides precise tools for performance monitoring through queries, including timestamp queries, which allow developers to measure the execution time of GPU commands. This chapter explores how to implement timestamp queries to measure GPU execution time, providing valuable insights into application performance.

47.2 What are Timestamp Queries?

Timestamp queries capture the current GPU time at specific points in the command buffer execution. By recording timestamps before and after a set of commands, you can calculate the elapsed GPU time for that workload. Timestamp queries are managed through query pools, which serve as containers for query results.

47.3 Step 1: Create a Query Pool

The first step in using timestamp queries is creating a query pool. A query pool is configured to track a specified number of queries and supports different query types, including timestamps.

```
1 // Create a query pool for timestamps
2 VkQueryPoolCreateInfo queryPoolInfo{};
3 queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
```



```

4 queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
5 queryPoolInfo.queryCount = 2; // We'll measure start and end timestamps
6
7 VkQueryPool queryPool;
8 if (vkCreateQueryPool(device, &queryPoolInfo, nullptr, &queryPool) != VK_SUCCESS) {
9     throw std::runtime_error("Failed to create query pool!");
10 }

```

In this example, the query pool is configured to hold two timestamps, one for the start and another for the end of a workload.

47.4 Step 2: Record Timestamps in the Command Buffer

Once the query pool is created, timestamps can be recorded within a command buffer using `vkCmdWriteTimestamp`. This function inserts a timestamp query at a specified pipeline stage.

```

1 // Begin recording the command buffer
2 VkCommandBuffer commandBuffer = beginCommandBuffer();
3
4 // Record the first timestamp at the start of the workload
5 vkCmdWriteTimestamp(commandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, queryPool, 0);
6
7 // Record some workload (e.g., rendering commands)
8 vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
9
10 // Record the second timestamp at the end of the workload
11 vkCmdWriteTimestamp(commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, queryPool, 1);
12
13 // End recording the command buffer
14 endCommandBuffer(commandBuffer);

```

In this example, the first timestamp is recorded at the `TOP_OF_PIPE` stage, and the second timestamp is recorded at the `BOTTOM_OF_PIPE` stage, bracketing the workload.

47.5 Step 3: Retrieve Query Results

After submitting the command buffer, query results can be retrieved to calculate the elapsed GPU time. Vulkan provides two options for retrieving results: synchronously or asynchronously.

```

1 // Submit the command buffer
2 VkSubmitInfo submitInfo{};
3 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
4 submitInfo.commandBufferCount = 1;
5 submitInfo.pCommandBuffers = &commandBuffer;
6
7 vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);
8 vkQueueWaitIdle(queue); // Ensure the GPU has finished executing
9
10 // Retrieve the timestamp results
11 uint64_t timestamps[2];
12 vkGetQueryPoolResults(device, queryPool, 0, 2, sizeof(timestamps), timestamps, sizeof(uint64_t),
13                       VK_QUERY_RESULT_64_BIT | VK_QUERY_RESULT_WAIT_BIT);
14
15 // Calculate the elapsed time

```

```
16 uint64_t timeDifference = timestamps[1] - timestamps[0];
17 double timestampPeriod = physicalDeviceProperties.limits.timestampPeriod; // In nanoseconds
18 double elapsedTimeMs = timeDifference * timestampPeriod * 1e-6;
19
20 std::cout << "Elapsed GPU time: " << elapsedTimeMs << " ms" << std::endl;
```

The `timestampPeriod` property, retrieved from the physical device properties, indicates the duration of each timestamp tick in nanoseconds. Multiplying the difference in timestamps by this period provides the elapsed time.

47.6 Best Practices for Using Timestamp Queries

1. **Minimize Query Pool Size:** Allocate query pools with the exact number of queries needed to reduce overhead.
2. **Pipeline Stage Selection:** Carefully choose pipeline stages for timestamp recording to ensure meaningful measurements.
3. **Avoid Frequent Queries:** Excessive use of timestamp queries can degrade performance by adding synchronization overhead.
4. **Batch Queries:** Record multiple queries in a single pool and retrieve them in batches to reduce API call overhead.

47.7 Limitations of Timestamp Queries

- **Device-Specific Accuracy:** The accuracy of timestamp queries depends on the GPU's timestamp resolution and clock synchronization.
- **Synchronization Overhead:** Retrieving query results requires ensuring GPU execution is complete, which may introduce stalls if not managed carefully.
- **Pipeline Behavior:** Measuring times for complex pipelines may require multiple timestamps and additional analysis.

47.8 Summary

Timestamp queries provide a powerful mechanism for measuring GPU performance in Vulkan applications. By accurately timing workloads, you can identify bottlenecks, evaluate optimization efforts, and better understand how your application utilizes the GPU. This chapter demonstrated the complete process of implementing timestamp queries, preparing you to integrate performance measurements into your own Vulkan projects.

48

Use Vulkan Subpasses: Optimize render passes with subpasses

48.1 Introduction

Vulkan introduces a unique rendering optimization mechanism called **subpasses**, which allows multiple operations to share the same render pass. Subpasses improve performance by reducing memory bandwidth usage when rendering dependent operations, such as performing post-processing directly after rendering a scene. In this chapter, you will learn how to use subpasses in Vulkan to create more efficient rendering workflows.

48.2 What Are Subpasses?

A subpass is a logical subdivision of a render pass, allowing different stages of rendering to occur sequentially while reusing attachments. Instead of writing intermediate results to memory and re-reading them, subpasses allow Vulkan to manage data within the GPU, minimizing memory traffic.

For example, you can use subpasses to:

1. Render a scene into a color attachment.
2. Perform post-processing effects, like bloom, in a subsequent subpass.

48.3 Step 1: Define Subpasses in a Render Pass

When creating a render pass, subpasses are defined as part of the `VkRenderPassCreateInfo` structure. Each subpass specifies the attachments it uses and how they interact with other subpasses.

```

1  // Create attachment descriptions
2  VkAttachmentDescription colorAttachment0;
3  colorAttachment0.format = swapchainImageFormat;
4  colorAttachment0.samples = VK_SAMPLE_COUNT_1_BIT;
5  colorAttachment0.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6  colorAttachment0.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
7  colorAttachment0.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
8  colorAttachment0.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
9
10 // Create attachment references
11 VkAttachmentReference colorAttachmentRef0;
12 colorAttachmentRef0.attachment = 0; // Index of color attachment in the render pass
13 colorAttachmentRef0.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
14
15 // Define the first subpass
16 VkSubpassDescription subpass0;
17 subpass0.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
18 subpass0.colorAttachmentCount = 1;
19 subpass0.pColorAttachments = &colorAttachmentRef0;
20
21 // Create the render pass
22 VkRenderPassCreateInfo renderPassInfo0;
23 renderPassInfo0.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
24 renderPassInfo0.attachmentCount = 1;
25 renderPassInfo0.pAttachments = &colorAttachment;
26 renderPassInfo0.subpassCount = 1;
27 renderPassInfo0.pSubpasses = &subpass;
28
29 VkRenderPass renderPass;
30 if (vkCreateRenderPass(device, &renderPassInfo0, nullptr, &renderPass) != VK_SUCCESS) {
31     throw std::runtime_error("Failed to create render pass!");
32 }

```

Here, we define a single subpass that renders to a color attachment. You can extend this approach to include multiple subpasses for advanced workflows.

48.4 Step 2: Add Dependencies Between Subpasses

When multiple subpasses are used, dependencies must be specified to ensure proper synchronization between them. Dependencies control the order of operations and prevent memory hazards.

```

1  // Define subpass dependencies
2  VkSubpassDependency dependency0;
3  dependency0.srcSubpass = VK_SUBPASS_EXTERNAL; // Source outside the render pass
4  dependency0.dstSubpass = 0; // Destination is the first subpass
5  dependency0.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
6  dependency0.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
7  dependency0.srcAccessMask = 0;
8  dependency0.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
9
10 renderPassInfo0.dependencyCount = 1;
11 renderPassInfo0.pDependencies = &dependency;

```

```

12
13 // Update the render pass creation
14 if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass) != VK_SUCCESS) {
15     throw std::runtime_error("Failed to create render pass with dependencies!");
16 }

```

This dependency ensures that the color attachment is ready before being written to in the first subpass.

48.5 Step 3: Record Commands for Subpasses

When recording command buffers, subpass transitions are managed by `vkCmdNextSubpass`, allowing you to switch between subpasses.

```

1 // Begin the render pass
2 VkRenderPassBeginInfo renderPassBeginInfo{};
3 renderPassBeginInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
4 renderPassBeginInfo.renderPass = renderPass;
5 renderPassBeginInfo.framebuffer = framebuffer;
6 renderPassBeginInfo.renderArea.offset = {0, 0};
7 renderPassBeginInfo.renderArea.extent = swapchainExtent;
8
9 VkClearValue clearValue{};
10 clearValue.color = {{0.0f, 0.0f, 0.0f, 1.0f}};
11 renderPassBeginInfo.clearValueCount = 1;
12 renderPassBeginInfo.pClearValues = &clearValue;
13
14 vkCmdBeginRenderPass(commandBuffer, &renderPassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
15
16 // Record commands for the first subpass
17 vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
18
19 // Transition to the next subpass
20 vkCmdNextSubpass(commandBuffer, VK_SUBPASS_CONTENTS_INLINE);
21
22 // Record commands for the next subpass (if applicable)
23
24 // End the render pass
25 vkCmdEndRenderPass(commandBuffer);

```

Each call to `vkCmdNextSubpass` transitions to the subsequent subpass, preserving the attachments between stages.

48.6 Advantages of Using Subpasses

1. **Reduced Memory Traffic:** Subpasses avoid writing intermediate results to memory, improving performance by keeping data within the GPU.
2. **Simplified Synchronization:** Vulkan handles synchronization between subpasses, reducing the need for explicit barriers.
3. **Increased Efficiency:** Optimized for mobile and embedded platforms where bandwidth is a limiting factor.

48.7 Limitations and Considerations

- **Complexity:** Subpass configuration can become complex for multi-pass rendering pipelines.
- **Hardware Support:** While widely supported, the efficiency of subpasses depends on the GPU architecture.
- **Attachment Layouts:** Ensure proper attachment layouts are used for compatibility between subpasses.

48.8 Summary

Subpasses are a powerful feature of Vulkan that enable optimized rendering by leveraging intra-pass operations. By reducing memory traffic and automating synchronization, subpasses make rendering pipelines more efficient. This chapter covered the foundational steps for implementing subpasses, preparing you to integrate them into your Vulkan applications for improved performance and flexibility.

49

Implement a Frame Limiter: Limit the FPS for consistency

49.1 Introduction

In real-time applications such as games and simulations, controlling the frame rate is essential for maintaining consistency and optimizing resource usage. A **frame limiter** ensures that the application does not exceed a target frame rate, preventing unnecessary GPU load and improving stability. In this chapter, you will learn how to implement a simple frame limiter in Vulkan to achieve consistent rendering.

49.2 Why Use a Frame Limiter?

Frame limiters serve several purposes:

1. **Prevent Excessive GPU Usage:** Rendering as fast as possible can cause overheating and excessive power consumption.
2. **Improve Predictability:** A fixed frame rate ensures smooth and predictable gameplay or animations.
3. **Sync with Display Refresh Rate:** Limiting FPS to the display's refresh rate can prevent tearing and reduce power usage, especially on mobile devices.

49.3 Step 1: Understand Timing in Vulkan

Vulkan does not provide a direct mechanism for frame timing, so the frame limiter relies on external time measurements. This can be achieved using high-resolution timers, such as `std::chrono` in C++.

49.4 Step 2: Calculate the Target Frame Time

To limit the frame rate, calculate the target frame duration based on the desired frames per second (FPS). For example, a target FPS of 60 corresponds to a frame duration of approximately 16.67 milliseconds.

```
1 // Set target FPS
2 const double targetFPS = 60.0;
3 const double targetFrameTime = 1.0 / targetFPS; // In seconds
```

49.5 Step 3: Measure Frame Execution Time

Using a high-resolution timer, measure the duration of each frame. Compare this duration with the target frame time to determine whether the application should wait before proceeding to the next frame.

```
1 #include <chrono>
2
3 // Record the start time of the frame
4 auto frameStartTime = std::chrono::high_resolution_clock::now();
5
6 // Perform rendering and presentation
7 renderFrame(); // Function that encapsulates rendering logic
8
9 // Record the end time of the frame
10 auto frameEndTime = std::chrono::high_resolution_clock::now();
11
12 // Calculate frame duration
13 std::chrono::duration<double> frameDuration = frameEndTime - frameStartTime;
```

49.6 Step 4: Implement Frame Delay

If the frame duration is shorter than the target frame time, introduce a delay to meet the desired frame rate. Use a precise sleep function, such as `std::this_thread::sleep_for`, to pause execution.

```
1 #include <thread>
2
3 // Calculate the remaining time to meet the target frame time
4 double frameTime = frameDuration.count(); // Convert duration to seconds
5 if (frameTime < targetFrameTime) {
6     double sleepTime = targetFrameTime - frameTime;
7     std::this_thread::sleep_for(std::chrono::duration<double>(sleepTime));
8 }
```

This ensures that the total frame time matches the desired interval for the target FPS.

49.7 Step 5: Integrate into the Main Loop

Combine the frame timing logic with Vulkan's rendering loop to maintain a consistent frame rate.

```

1 while (!shouldCloseWindow()) {
2     // Record the start time of the frame
3     auto frameStartTime = std::chrono::high_resolution_clock::now();
4
5     // Perform rendering
6     renderFrame();
7
8     // Record the end time of the frame
9     auto frameEndTime = std::chrono::high_resolution_clock::now();
10
11    // Calculate frame duration
12    std::chrono::duration<double> frameDuration = frameEndTime - frameStartTime;
13
14    // Sleep if necessary
15    double frameTime = frameDuration.count();
16    if (frameTime < targetFrameTime) {
17        double sleepTime = targetFrameTime - frameTime;
18        std::this_thread::sleep_for(std::chrono::duration<double>(sleepTime));
19    }
20 }
```

This loop continuously measures frame time, performs rendering, and waits if needed to limit FPS.

49.8 Considerations and Best Practices

1. **Precision:** Ensure that your timer and sleep mechanism provide sufficient precision for smooth frame limiting.
2. **Dynamic Frame Rates:** Allow for dynamic adjustment of the target FPS to accommodate different hardware or power-saving modes.
3. **V-Sync:** Combine frame limiting with Vulkan's presentation modes (e.g., `VK_PRESENT_MODE_FIFO_KHR`) to further sync with the display refresh rate.
4. **Monitor Frame Timing:** Continuously monitor actual frame timing to ensure the limiter functions correctly.

49.9 Summary

Implementing a frame limiter in Vulkan improves application consistency and optimizes resource usage. By using high-resolution timers and introducing delays when necessary, you can maintain a predictable frame rate and reduce unnecessary GPU strain. This chapter provided a step-by-step guide to adding a frame limiter to your Vulkan application, setting the stage for more efficient and stable rendering pipelines.

50

Use Device Extensions: Enable advanced GPU features via extensions

50.1 Introduction

Vulkan provides a modular design that allows you to enable advanced GPU features through **extensions**. Device extensions extend the core Vulkan API, providing access to additional functionality such as ray tracing, enhanced memory management, and multi-GPU support. In this chapter, you will learn how to query and enable device extensions to unlock the full potential of modern GPUs.

50.2 What Are Device Extensions?

Device extensions are optional features supported by specific GPUs. These features are not always part of the Vulkan core API but can provide significant performance or functionality enhancements when available. Examples of popular extensions include:

- `VK_KHR_swapchain` : Required for presenting rendered images to the screen.
- `VK_EXT_descriptor_indexing` : Provides advanced descriptor management.
- `VK_NV_ray_tracing` : Adds support for ray tracing on NVIDIA GPUs.

To use an extension, you must query its availability on the target GPU and explicitly enable it during device creation.

50.3 Step 1: Query Supported Extensions

Before enabling an extension, you need to check if it is supported by the physical device. Vulkan provides the `vkEnumerateDeviceExtensionProperties` function to retrieve the list of supported extensions.

```

1 // Query the number of extensions
2 uint32_t extensionCount = 0;
3 vkEnumerateDeviceExtensionProperties(physicalDevice, nullptr, &extensionCount, nullptr);
4
5 // Allocate space for extension properties
6 std::vector<VkExtensionProperties> availableExtensions(extensionCount);
7 vkEnumerateDeviceExtensionProperties(physicalDevice, nullptr, &extensionCount, availableExtensions.data());
8
9 // Print supported extensions
10 std::cout << "Available extensions:" << std::endl;
11 for (const auto& extension : availableExtensions) {
12     std::cout << "\t" << extension.extensionName << std::endl;
13 }
```

This code retrieves the list of available extensions for a given physical device and prints their names. You can compare this list against the extensions your application requires.

50.4 Step 2: Specify Required Extensions

Once you know which extensions are available, create a list of the extensions you want to enable. At a minimum, most Vulkan applications require the `VK_KHR_swapchain` extension for presenting rendered images.

```

1 // Define required extensions
2 const std::vector<const char*> requiredExtensions = {
3     VK_KHR_SWAPCHAIN_EXTENSION_NAME // Swapchain extension
4 };
```

50.5 Step 3: Check for Extension Support

To ensure that all required extensions are supported, compare your list of required extensions against the list of available extensions.

```

1 // Check for required extensions
2 bool checkDeviceExtensionSupport(VkPhysicalDevice device, const std::vector<const char*>& requiredExtensions) {
3     // Query available extensions
4     uint32_t extensionCount;
5     vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, nullptr);
6     std::vector<VkExtensionProperties> availableExtensions(extensionCount);
7     vkEnumerateDeviceExtensionProperties(device, nullptr, &extensionCount, availableExtensions.data());
8
9     // Verify each required extension is supported
10    for (const char* required : requiredExtensions) {
11        bool found = false;
12        for (const auto& extension : availableExtensions) {
13            if (strcmp(required, extension.extensionName) == 0) {
14                found = true;
15                break;
16            }
17        }
18    }
19 }
```

```

16     }
17 }
18 if (!found) return false;
19 }
20 return true;
21 }

```

Call this function during physical device selection to filter out devices that do not support your required extensions.

50.6 Step 4: Enable Extensions During Logical Device Creation

To use extensions, pass their names to the `VkDeviceCreateInfo` structure when creating the logical device.

```

1 // Create device creation info
2 VkDeviceCreateInfo createInfo{};
3 createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
4 createInfo.enabledExtensionCount = static_cast<uint32_t>(requiredExtensions.size());
5 createInfo.ppEnabledExtensionNames = requiredExtensions.data();
6
7 // Create the logical device
8 VkDevice device;
9 if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS) {
10     throw std::runtime_error("Failed to create logical device with extensions!");
11 }

```

By enabling the desired extensions during device creation, you can use the corresponding Vulkan features.

50.7 Common Device Extensions

Here are a few commonly used extensions and their purposes:

- `VK_KHR_swapchain`: Essential for presenting images to the screen.
- `VK_EXT_debug_marker`: Useful for debugging Vulkan applications.
- `VK_KHR_acceleration_structure`: Required for ray tracing pipelines.
- `VK_EXT_memory_budget`: Provides information about memory usage on the GPU.

50.8 Considerations When Using Extensions

1. **Compatibility**: Not all extensions are supported on all GPUs. Always verify support during physical device selection.
2. **Portability**: Excessive use of proprietary extensions can reduce the portability of your application.
3. **Documentation**: Refer to the Vulkan Specification for details about each extension.

50.9 Conclusion

Device extensions provide a powerful way to access advanced GPU features in Vulkan. By querying supported extensions, checking compatibility, and enabling them during device creation, you can unlock the full capabilities of your hardware. This chapter demonstrated the steps to integrate device extensions into your Vulkan application, preparing you to leverage features such as swapchains, ray tracing, and enhanced descriptor management.

DRAFT
version
0.681

Part VI

Advanced Vulkan Features

51

Ray Tracing Basics: Render a simple scene with ray tracing

51.1 Introduction

Ray tracing is a powerful rendering technique that simulates the way light interacts with surfaces to produce realistic visuals. With Vulkan's support for ray tracing through extensions such as `VK_KHR_ray_tracing_pipeline`, developers can now implement advanced rendering features directly on modern GPUs. This chapter introduces the basics of setting up and using ray tracing in Vulkan to render a simple scene.

51.2 Overview of Vulkan Ray Tracing

Vulkan's ray tracing functionality is built on several key components:

- **Acceleration Structures:** Data structures that store information about the geometry in the scene for efficient ray traversal.
- **Ray Tracing Pipelines:** Specialized pipelines for handling ray generation, intersection, and shading.
- **Shader Stages:** Unique shader stages like ray generation, closest-hit, and miss shaders are used to process rays.

Before starting, ensure that your GPU supports the required Vulkan ray tracing extensions:

- `VK_KHR_acceleration_structure`
- `VK_KHR_ray_tracing_pipeline`
- `VK_KHR_deferred_host_operations`

51.3 Step 1: Enable Ray Tracing Extensions

Enable the necessary extensions during logical device creation, as explained in the previous chapter.

```

1 // Required extensions for ray tracing
2 const std::vector<const char*> requiredExtensions = {
3     VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
4     VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME,
5     VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME
6 };
7
8 // Pass these extensions during logical device creation
9 VkDeviceCreateInfo createInfo{};
10 createInfo.enabledExtensionCount = static_cast<uint32_t>(requiredExtensions.size());
11 createInfo.ppEnabledExtensionNames = requiredExtensions.data();

```

Additionally, ensure your Vulkan instance has validation layers and debugging enabled to facilitate development.

51.4 Step 2: Create Acceleration Structures

Acceleration structures (AS) are the foundation of ray tracing in Vulkan. They are divided into two types:

1. **Bottom-Level Acceleration Structure (BLAS)**: Represents the geometry of individual objects.
2. **Top-Level Acceleration Structure (TLAS)**: Represents the scene hierarchy and references multiple BLAS instances.

Bottom-Level Acceleration Structure (BLAS)

The BLAS stores the geometry of a single object, such as a triangle or a cube. Begin by defining the geometry and building the BLAS.

```

1 // Define triangle geometry
2 VkGeometryTrianglesKHR triangles{};
3 triangles.vertexData = vertexBufferAddress;
4 triangles.vertexStride = sizeof(Vertex);
5 triangles.maxVertex = vertexCount;
6 triangles.indexData = indexBufferAddress;
7 triangles.indexType = VK_INDEX_TYPE_UINT32;
8
9 // Geometry instance for BLAS
10 VkAccelerationStructureGeometryKHR geometry{};
11 geometry.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
12 geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
13 geometry.geometry.triangles = triangles;
14 geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;
15
16 // Build the BLAS
17 // (The process involves creating buffers, allocating memory, and using ray tracing commands)

```

Top-Level Acceleration Structure (TLAS)

The TLAS references multiple BLAS instances and provides the ray tracer with a scene-wide hierarchy.

```

1 // Instance data for TLAS
2 VkAccelerationStructureInstanceKHR instance{};
3 instance.transform = { ... }; // Transformation matrix
4 instance.instanceCustomIndex = 0;
5 instance.mask = 0xFF;
6 instance.instanceShaderBindingTableRecordOffset = 0;
7 instance.flags = VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
8
9 // Build the TLAS
10 // (Similar to BLAS, but involves instances instead of geometry)

```

51.5 Step 3: Create the Ray Tracing Pipeline

The ray tracing pipeline is specialized for handling ray tracing-specific shaders. It includes:

- A **ray generation shader** that emits rays.
- A **closest-hit shader** that processes intersections with geometry.
- A **miss shader** that handles rays that do not hit any geometry.

Ray Generation Shader

The ray generation shader is responsible for initiating rays and passing them into the ray tracing pipeline.

```

1 // Example GLSL for ray generation
2 #version 460
3 #extension GL_EXT_ray_tracing : require
4
5 layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
6 layout(location = 0) rayPayloadEXT vec3 payload;
7
8 void main() {
9     vec3 rayOrigin = vec3(0.0, 0.0, 0.0);
10    vec3 rayDirection = vec3(0.0, 0.0, -1.0);
11    traceRayEXT(topLevelAS, 0, 0xFF, 0, 1, 0, rayOrigin, 0.0, rayDirection, 1000.0, payload);
12 }

```

Closest-Hit Shader

This shader calculates the color and material properties of a ray when it intersects with geometry.

```

1 // Example GLSL for closest-hit shader
2 #version 460
3 #extension GL_EXT_ray_tracing : require
4
5 layout(location = 0) rayPayloadInEXT vec3 payload;
6
7 void main() {

```

```

8     payload = vec3(1.0, 0.0, 0.0); // Red color for hit objects
9 }

```

Miss Shader

This shader handles rays that do not intersect with any geometry.

```

1 // Example GLSL for miss shader
2 #version 460
3 #extension GL_EXT_ray_tracing : require
4
5 layout(location = 0) rayPayloadInEXT vec3 payload;
6
7 void main() {
8     payload = vec3(0.0, 0.0, 1.0); // Blue color for background
9 }

```

51.6 Step 4: Create and Use Shader Binding Tables (SBTs)

The Shader Binding Table (SBT) maps ray tracing shaders to their execution contexts. Vulkan requires that you create SBTs for your pipeline.

```

1 // Create shader binding table (SBT)
2 // The SBT contains ray generation, miss, and hit groups
3 VkStridedDeviceAddressRegionKHR raygenRegion = { ... };
4 VkStridedDeviceAddressRegionKHR missRegion = { ... };
5 VkStridedDeviceAddressRegionKHR hitRegion = { ... };
6
7 // Bind the SBT during dispatch
8 vkCmdTraceRaysKHR(commandBuffer, &raygenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

51.7 Step 5: Dispatch Rays

Use the `vkCmdTraceRaysKHR` function to dispatch rays in the ray tracing pipeline. The ray generation shader will handle the rays and delegate their processing to the appropriate shaders.

```

1 // Dispatch rays
2 vkCmdTraceRaysKHR(commandBuffer, &raygenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

51.8 Summary

This chapter introduced the basics of Vulkan ray tracing, guiding you through enabling ray tracing extensions, creating acceleration structures, and setting up the ray tracing pipeline.

52

Add BVH Acceleration Structures: Create bounding volume hierarchies

52.1 Introduction

Bounding Volume Hierarchies (BVH) are essential data structures used to accelerate ray tracing. By organizing geometric data into a hierarchical tree of bounding volumes, BVH allows the ray tracing pipeline to efficiently cull geometry and reduce the number of intersection tests. This chapter guides you through the process of creating and managing BVH acceleration structures in Vulkan.

52.2 Overview of Acceleration Structures in Vulkan

Vulkan uses two types of acceleration structures:

- **Bottom-Level Acceleration Structure (BLAS):** Represents individual objects or geometric primitives (e.g., triangles or meshes).
- **Top-Level Acceleration Structure (TLAS):** Represents the scene hierarchy, referencing multiple BLAS instances and their transformations.

The BVH is implemented through these acceleration structures, enabling efficient traversal and intersection testing in the ray tracing pipeline.

52.3 Step 1: Define the Geometry for BLAS

The first step in creating a BVH is defining the geometry for the bottom-level acceleration structure. Vulkan supports triangles and axis-aligned bounding boxes (AABBs) as primitive types.

```

1 // Define triangle geometry for the BLAS
2 VkGeometryTrianglesKHR triangles{};
3 triangles.sType = VK_STRUCTURE_TYPE_GEOMETRY_TRIANGLES_KHR;
4 triangles.vertexData.deviceAddress = vertexBufferAddress; // GPU address of vertex buffer
5 triangles.vertexStride = sizeof(Vertex);
6 triangles.maxVertex = vertexCount;
7 triangles.indexData.deviceAddress = indexBufferAddress; // GPU address of index buffer
8 triangles.indexType = VK_INDEX_TYPE_UINT32;
9
10 // Configure BLAS geometry
11 VkAccelerationStructureGeometryKHR geometry{};
12 geometry.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
13 geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
14 geometry.geometry.triangles = triangles;
15 geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR; // No alpha testing

```

52.4 Step 2: Build the BLAS

After defining the geometry, the next step is to build the BLAS. This involves calculating the size of required buffers, allocating memory, and issuing build commands.

Query Build Sizes

Use `vkGetAccelerationStructureBuildSizesKHR` to determine the size of the buffers needed for the BLAS.

```

1 // Query size for BLAS
2 VkAccelerationStructureBuildSizesInfoKHR buildSizes{};
3 buildSizes.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_SIZES_INFO_KHR;
4
5 VkAccelerationStructureBuildGeometryInfoKHR buildInfo{};
6 buildInfo.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
7 buildInfo.type = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
8 buildInfo.geometryCount = 1;
9 buildInfo.pGeometries = &geometry;
10
11 vkGetAccelerationStructureBuildSizesKHR(device,
12     VK_ACCELERATION_STRUCTURE_BUILD_TYPE_DEVICE_KHR,
13     &buildInfo,
14     &primitiveCount,
15     &buildSizes);

```

Allocate Buffers

Allocate buffers for the BLAS, including the acceleration structure itself and temporary scratch buffers required during construction.

```

1 // Create the acceleration structure buffer

```

```

2  VkBufferCreateInfo bufferCreateInfo = { ... }; // Define buffer properties
3  VkBuffer blasBuffer;
4  vkCreateBuffer(device, &bufferCreateInfo, nullptr, &blasBuffer);
5
6  // Allocate memory for the buffer
7  VkMemoryRequirements memoryRequirements;
8  vkGetBufferMemoryRequirements(device, blasBuffer, &memoryRequirements);
9  VkDeviceMemory blasMemory;
10 vkAllocateMemory(device, &memoryAllocateInfo, nullptr, &blasMemory);
11 vkBindBufferMemory(device, blasBuffer, blasMemory, 0);

```

Issue Build Commands

Build the BLAS using the Vulkan command buffer.

```

1  // Record commands for BLAS construction
2  VkCommandBuffer commandBuffer = beginCommandBuffer();
3
4  VkAccelerationStructureBuildRangeInfoKHR buildRange{};
5  buildRange.primitiveCount = primitiveCount;
6
7  const VkAccelerationStructureBuildRangeInfoKHR* pBuildRange = &buildRange;
8
9  vkCmdBuildAccelerationStructuresKHR(commandBuffer, 1, &buildInfo, &pBuildRange);
10
11 endCommandBuffer(commandBuffer);

```

52.5 Step 3: Build the TLAS

The top-level acceleration structure (TLAS) references instances of BLAS. Each instance includes a transformation matrix and an instance ID.

Define TLAS Instances

Create instance data for the TLAS, which includes transformations and references to BLAS handles.

```

1  // Define TLAS instance
2  VkAccelerationStructureInstanceKHR instance{};
3  instance.transform = { ... }; // Transformation matrix (row-major)
4  instance.instanceCustomIndex = 0; // Custom ID for the instance
5  instance.mask = 0xFF; // Visibility mask
6  instance.instanceShaderBindingTableRecordOffset = 0;
7  instance.flags = VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
8  instance.accelerationStructureReference = blasDeviceAddress; // GPU address of BLAS

```

Build the TLAS

Follow a process similar to BLAS construction, but instead of geometry, use instance data.

```

1  // Configure TLAS geometry
2  VkAccelerationStructureGeometryKHR tlasGeometry{};
3  tlasGeometry.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
4  tlasGeometry.geometryType = VK_GEOMETRY_TYPE_INSTANCES_KHR;

```



```

5  tlasGeometry.geometry.instances.data.deviceAddress = instanceBufferAddress;
6  tlasGeometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;
7
8  // Build TLAS using command buffer
9  vkCmdBuildAccelerationStructuresKHR(commandBuffer, 1, &tlasBuildInfo, &pBuildRange);

```

52.6 Step 4: Integrate BVH with Ray Tracing Pipeline

Once the BLAS and TLAS are built, bind the TLAS to your ray tracing pipeline via a descriptor set. The TLAS serves as the top-level acceleration structure for ray traversal.

```

1  // Bind TLAS to ray tracing pipeline
2  VkWriteDescriptorSetAccelerationStructureKHR descriptorAS{};
3  descriptorAS.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
4  descriptorAS.accelerationStructureCount = 1;
5  descriptorAS.pAccelerationStructures = &tlasHandle;
6
7  VkWriteDescriptorSet writeDescriptorSet{};
8  writeDescriptorSet.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9  writeDescriptorSet.dstSet = descriptorSet;
10 writeDescriptorSet.dstBinding = 0;
11 writeDescriptorSet.descriptorType = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
12 writeDescriptorSet.pNext = &descriptorAS;
13
14 vkUpdateDescriptorSets(device, 1, &writeDescriptorSet, 0, nullptr);

```

52.7 Summary

In this chapter, you learned how to create Bounding Volume Hierarchies (BVH) using Vulkan's acceleration structure APIs. By constructing BLAS for individual objects and TLAS for the scene, you enable efficient ray traversal for ray tracing. BVH construction is a critical optimization step, and mastering it prepares you for implementing more complex ray tracing effects such as reflections, shadows, and global illumination.

53

Render Reflections: Add ray-traced reflections

53.1 Introduction

Reflections are a hallmark of realism in computer graphics, allowing objects to mirror their surroundings accurately. Using Vulkan's ray tracing capabilities, you can calculate reflections by tracing rays from the camera to reflective surfaces and beyond. This chapter guides you through implementing ray-traced reflections, leveraging acceleration structures and the ray tracing pipeline.

53.2 Reflection Principles

Reflections in ray tracing are calculated by casting rays from the surface of a reflective object. These rays are determined by the surface normal and the direction of the incoming ray (view ray). The intersection of the reflected ray with other scene objects determines the color to be reflected.

In Vulkan, this process involves:

- Defining and tracing rays within a ray generation shader.
- Calculating reflection directions using surface normals.
- Fetching reflection data by intersecting the reflected rays with the scene geometry.

53.3 Step 1: Extend the Ray Tracing Pipeline

To implement reflections, extend the ray tracing pipeline to include reflection handling. Update your shaders and descriptor sets to manage additional resources.

Update the Ray Generation Shader

In the ray generation shader, calculate the view direction and trace reflected rays.

```

1 // Ray generation shader
2
3 #version 460
4 #extension GL_EXT_ray_tracing : require
5
6 layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
7 layout(location = 0) rayPayloadEXT vec3 reflectionColor;
8
9 void main() {
10     vec3 rayOrigin = ...; // Camera origin
11     vec3 rayDir = ...;    // Primary ray direction
12
13     // Trace the primary ray
14     traceRayEXT(topLevelAS, RAY_FLAGS_OPAQUE_EXT, 0xFF, 0, 1, 0, rayOrigin, 0.0, rayDir, 1e32, 0);
15
16     // Output the result
17     vec3 finalColor = reflectionColor;
18     imageStore(outputImage, ivec2(gl_LaunchIDEXT.xy), vec4(finalColor, 1.0));
19 }

```

Add Reflection Calculations in the Closest Hit Shader

Modify the closest hit shader to calculate reflected rays.

```

1 // Closest hit shader
2
3 #version 460
4 #extension GL_EXT_ray_tracing : require
5
6 layout(location = 0) rayPayloadInEXT vec3 reflectionColor;
7
8 layout(binding = 0) uniform sampler2D albedoTexture;
9
10 void main() {
11     vec3 normal = normalize(hitNormal); // Normal at the intersection point
12     vec3 viewDir = normalize(rayOrigin - hitPosition);
13     vec3 reflectedDir = reflect(viewDir, normal);
14
15     // Trace the reflected ray
16     vec3 secondaryColor;
17     traceRayEXT(topLevelAS, RAY_FLAGS_OPAQUE_EXT, 0xFF, 0, 1, 0, hitPosition, 0.001, reflectedDir, 1e32, 0);
18
19     // Combine reflection with albedo
20     vec3 baseColor = texture(albedoTexture, texCoords).rgb;
21     reflectionColor = mix(baseColor, secondaryColor, reflectionFactor);
22 }

```

53.4 Step 2: Update Descriptor Sets

Add necessary resources for textures, acceleration structures, and other reflection-related data.

```

1 // Write descriptor for top-level acceleration structure
2 VkWriteDescriptorSetAccelerationStructureKHR asDescriptor{0};
3 asDescriptor.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET_ACCELERATION_STRUCTURE_KHR;
4 asDescriptor.accelerationStructureCount = 1;
5 asDescriptor.pAccelerationStructures = &tlas;
6
7 VkWriteDescriptorSet writeAS{0};
8 writeAS.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9 writeAS.dstSet = descriptorSet;
10 writeAS.dstBinding = 0;
11 writeAS.descriptorType = VK_DESCRIPTOR_TYPE_ACCELERATION_STRUCTURE_KHR;
12 writeAS.pNext = &asDescriptor;
13
14 // Update the descriptor set
15 vkUpdateDescriptorSets(device, 1, &writeAS, 0, nullptr);

```

53.5 Step 3: Integrate Reflections into the Render Pass

With the shaders updated and the resources configured, integrate reflections into your Vulkan render pass. Ensure that your swapchain image or off-screen framebuffer captures the output from the ray tracing pipeline.

```

1 // Record commands for ray tracing and image presentation
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rayTracingPipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, pipelineLayout, 0, 1, &
  ↳ descriptorSet, 0, nullptr);
4
5 // Dispatch ray tracing commands
6 vkCmdTraceRaysKHR(commandBuffer, &rayGenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

53.6 Step 4: Optimize Reflection Performance

Ray-traced reflections can be computationally expensive. To optimize:

- Limit the maximum reflection depth (number of bounces).
- Use lower-resolution images or blur post-processing for reflections.
- Implement hybrid approaches, combining ray tracing with rasterized techniques like screen space reflections (SSR).

53.7 Summary

In this chapter, you learned how to add ray-traced reflections to your Vulkan application. By leveraging the ray tracing pipeline and BVH acceleration structures, you can achieve highly realistic reflective surfaces. As reflections are computationally intensive, optimization techniques are crucial to maintaining real-time performance.

54

Render Refractions: Implement glass-like materials

54.1 Introduction

Refraction is a phenomenon where light bends as it passes through a transparent material, such as glass or water. Rendering refractions realistically enhances the visual quality of scenes involving transparent objects. In this chapter, you will learn how to use Vulkan's ray tracing capabilities to simulate refractions by calculating the bending of light rays and rendering objects as seen through refractive surfaces.

54.2 Understanding Refraction

Refraction is described by Snell's Law:

$$n_1 \sin \theta_1 = n_2 \sin \theta_2$$

Here:

- n_1 and n_2 are the refractive indices of the two media (e.g., air and glass).
- θ_1 is the angle of incidence.
- θ_2 is the angle of refraction.

To simulate refraction:

1. Compute the refracted ray direction using the surface normal and Snell's Law.

2. Trace the refracted ray through the scene.
3. Use the resulting intersection to determine the color or texture to display.

54.3 Step 1: Extend the Ray Tracing Shaders

Modify the Closest Hit Shader

The closest hit shader computes the refracted ray direction based on the material's refractive index. It then traces the refracted ray through the scene.

```

1  // Closest hit shader for refraction
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(location = 0) rayPayloadInEXT vec3 refractedColor;
7
8  layout(binding = 0) uniform sampler2D albedoTexture;
9  layout(binding = 1) uniform float refractiveIndex; // e.g., 1.5 for glass
10
11 void main() {
12     vec3 normal = normalize(hitNormal); // Surface normal at the intersection point
13     vec3 viewDir = normalize(rayOrigin - hitPosition); // View ray direction
14
15     // Compute the refracted ray direction
16     float eta = 1.0 / refractiveIndex; // Ratio of refractive indices (air to material)
17     float cosI = dot(-viewDir, normal);
18     float k = 1.0 - eta * eta * (1.0 - cosI * cosI);
19
20     vec3 refractedDir = (k > 0.0)
21         ? eta * viewDir + (eta * cosI - sqrt(k)) * normal
22         : vec3(0.0); // Total internal reflection if k < 0
23
24     // Trace the refracted ray
25     traceRayEXT(topLevelAS, RAY_FLAGS_OPAQUE_EXT, 0xFF, 0, 1, 0,
26         hitPosition, 0.001, refractedDir, 1e32, 0);
27
28     // Mix the refraction with the surface texture
29     vec3 baseColor = texture(albedoTexture, texCoords).rgb;
30     refractedColor = mix(baseColor, refractedColor, 0.7);
31 }

```

Update the Ray Generation Shader

The ray generation shader remains largely unchanged but ensures that refracted rays are traced and their results are stored.

```

1  // Ray generation shader
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
7  layout(location = 0) rayPayloadEXT vec3 refractedColor;
8
9  void main() {
10     vec3 rayOrigin = ...; // Camera origin
11     vec3 rayDir = ...;    // Primary ray direction
12

```

```

13 // Trace the primary ray
14 traceRayEXT(topLevelAS, RAY_FLAGS_OPAQUE_EXT, 0xFF, 0, 1, 0, rayOrigin, 0.0, rayDir, 1e32, 0);
15
16 // Output the result
17 imageStore(outputImage, ivec2(gl_LaunchIDEXT.xy), vec4(refractedColor, 1.0));
18 }

```

54.4 Step 2: Update Vulkan Resources

Add resources to your Vulkan application for storing material properties such as the refractive index. Update descriptor sets to bind this data to your shaders.

```

1 // Define a uniform buffer for material properties
2 struct MaterialProperties {
3     float refractiveIndex;
4 };
5
6 // Create and upload buffer
7 VkBuffer materialBuffer;
8 createBuffer(device, sizeof(MaterialProperties), VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, &materialBuffer);
9 uploadBufferData(device, materialBuffer, &materialProperties);
10
11 // Update descriptor sets
12 VkDescriptorBufferInfo materialBufferInfo{0};
13 materialBufferInfo.buffer = materialBuffer;
14 materialBufferInfo.offset = 0;
15 materialBufferInfo.range = sizeof(MaterialProperties);
16
17 VkWriteDescriptorSet writeMaterial{0};
18 writeMaterial.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
19 writeMaterial.dstSet = descriptorSet;
20 writeMaterial.dstBinding = 1;
21 writeMaterial.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
22 writeMaterial.pBufferInfo = &materialBufferInfo;
23
24 vkUpdateDescriptorSets(device, 1, &writeMaterial, 0, nullptr);

```

54.5 Step 3: Integrate Refractions into the Render Pass

Integrate the refraction computations into your ray tracing pipeline and render pass. This involves recording commands to dispatch ray tracing and presenting the result.

```

1 // Bind pipeline and descriptor sets
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rayTracingPipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, pipelineLayout, 0, 1, &
  ↳ descriptorSet, 0, nullptr);
4
5 // Dispatch ray tracing
6 vkCmdTraceRaysKHR(commandBuffer, &rayGenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

54.6 Step 4: Optimize Refractions

Refraction calculations can be computationally intensive. Consider the following optimizations:

- **Limit ray recursion depth:** Prevent excessive bounces by setting a maximum recursion depth.
- **Use lower-resolution refraction textures:** Reduce the computational cost of secondary rays.
- **Combine with screen-space refraction:** Use hybrid approaches for better performance in scenes where accuracy is less critical.

54.7 Summary

This chapter introduced the implementation of glass-like refractions using Vulkan's ray tracing capabilities. By simulating light bending and tracing refracted rays through the scene, you can create realistic transparent materials. With this foundation, you can extend the technique to implement other effects, such as water surfaces or advanced light scattering in volumetric objects.

55

Path Tracing Basics: Render a simple path-traced scene

55.1 Introduction

Path tracing is a rendering technique that simulates the global illumination of a scene by tracing rays and their interactions with surfaces. Unlike traditional rasterization, path tracing handles effects like reflections, refractions, shadows, and indirect lighting in a unified manner. This chapter introduces the fundamentals of path tracing in Vulkan, guiding you through the creation of a simple path-traced scene.

55.2 What is Path Tracing?

Path tracing extends the concept of ray tracing by simulating the journey of light rays as they bounce across surfaces. Each ray contributes to the final pixel color by sampling light contributions from direct and indirect illumination. This iterative process, combined with Monte Carlo integration, produces realistic lighting effects.

Key features of path tracing:

- **Global Illumination:** Includes both direct and indirect lighting.
- **Monte Carlo Sampling:** Randomly samples rays to approximate the lighting equation.
- **Physically Based Rendering (PBR):** Uses realistic material models and light interactions.

55.3 Step 1: Extend the Ray Tracing Pipeline for Path Tracing

Path tracing builds on the Vulkan ray tracing pipeline. You'll use the ray generation, closest hit, and miss shaders to implement light sampling and multiple ray bounces.

Ray Generation Shader

The ray generation shader initializes primary rays and handles iterative sampling.

```

1  // Ray generation shader
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
7  layout(location = 0) rayPayloadEXT vec3 accumulatedColor;
8
9  layout(location = 1) uniform vec3 cameraOrigin;
10 layout(location = 2) uniform vec3 cameraDirection;
11 layout(location = 3) uniform float lensRadius;
12 layout(location = 4) uniform float focusDistance;
13 layout(location = 5) uniform int maxBounces;
14
15 void main() {
16     vec3 rayOrigin = cameraOrigin;
17     vec3 rayDir = calculatePrimaryRayDirection(gl_LaunchIDEXT.xy, cameraDirection, lensRadius, focusDistance);
18
19     accumulatedColor = vec3(0.0);
20     vec3 throughput = vec3(1.0);
21
22     for (int bounce = 0; bounce < maxBounces; ++bounce) {
23         traceRayEXT(topLevelAS, RAY_FLAGS_NONE_EXT, 0xFF, 0, 1, 0, rayOrigin, 0.001, rayDir, 1e32, 0);
24
25         if (terminateRay()) {
26             break;
27         }
28
29         accumulatedColor += throughput * computeLighting();
30         throughput *= computeMaterialThroughput();
31         rayOrigin = nextRayOrigin();
32         rayDir = nextRayDirection();
33     }
34
35     imageStore(outputImage, ivec2(gl_LaunchIDEXT.xy), vec4(accumulatedColor, 1.0));
36 }

```

Closest Hit Shader

The closest hit shader computes the interaction of the ray with the surface, including material properties and light sampling.

```

1  // Closest hit shader
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(location = 0) rayPayloadInEXT vec3 accumulatedColor;
7
8  void main() {
9     vec3 normal = normalize(hitNormal);

```

```

10   vec3 viewDir = normalize(rayOrigin - hitPosition);
11
12   // Direct lighting
13   vec3 directLighting = sampleLightSources(hitPosition, normal);
14
15   // Update ray data for indirect lighting
16   updateRayData(viewDir, normal, hitPosition);
17
18   accumulatedColor = directLighting;
19 }

```

Miss Shader

The miss shader handles rays that escape the scene, typically contributing background color or environment maps.

```

1  // Miss shader
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(location = 0) rayPayloadInEXT vec3 accumulatedColor;
7
8  void main() {
9      accumulatedColor = vec3(0.1, 0.1, 0.2); // Background color
10 }

```

55.4 Step 2: Configure Vulkan Resources

Update your Vulkan application to manage buffers for random number generation, Monte Carlo sampling, and per-pixel accumulations.

```

1  // Create uniform buffer for camera and rendering settings
2  struct PathTracingSettings {
3      glm::vec3 cameraOrigin;
4      glm::vec3 cameraDirection;
5      float lensRadius;
6      float focusDistance;
7      int maxBounces;
8  };
9
10 VkBuffer settingsBuffer;
11 createBuffer(device, sizeof(PathTracingSettings), VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, &settingsBuffer);
12
13 // Update descriptor sets
14 VkWriteDescriptorSet writeSettings{};
15 writeSettings.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
16 writeSettings.dstSet = descriptorSet;
17 writeSettings.dstBinding = 0;
18 writeSettings.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
19 writeSettings.pBufferInfo = &settingsBufferInfo;
20
21 vkUpdateDescriptorSets(device, 1, &writeSettings, 0, nullptr);

```


55.5 Step 3: Integrate Path Tracing into the Render Loop

Record commands to dispatch path tracing rays and accumulate results across frames.

```

1 // Bind pipeline and descriptor sets
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rayTracingPipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, pipelineLayout, 0, 1, &
  ↳ descriptorSet, 0, nullptr);
4
5 // Dispatch ray tracing for each frame
6 vkCmdTraceRaysKHR(commandBuffer, &rayGenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

55.6 Optimizations and Challenges

Path tracing is computationally expensive due to the high number of rays and bounces. Techniques to improve performance include:

- **Denoising:** Use post-processing filters to reduce noise from undersampling.
- **Importance Sampling:** Focus rays on significant light sources or materials.
- **Adaptive Sampling:** Stop tracing rays that contribute negligibly to the final result.

55.7 Summary

This chapter introduced the fundamentals of path tracing with Vulkan, enabling realistic global illumination. While simple path tracing can produce stunning visuals, advanced techniques such as denoising and optimization are necessary for real-time performance. With this foundation, you can build more complex lighting models and materials in your Vulkan applications.

56

Simulate Global Illumination: Add realistic lighting effects

56.1 Introduction

Global illumination (GI) is a rendering technique that calculates the way light bounces and scatters across surfaces in a scene, creating realistic lighting effects. By considering both direct and indirect lighting, GI enhances the visual fidelity of rendered scenes, providing soft shadows, color bleeding, and natural light diffusion. This chapter demonstrates how to simulate global illumination in Vulkan by extending the concepts of ray tracing and path tracing.

56.2 What is Global Illumination?

Global illumination accounts for:

- **Direct Lighting:** Light directly reaching a surface from a source.
- **Indirect Lighting:** Light reflected or refracted from other surfaces.

Unlike basic ray tracing, which focuses on primary rays and direct lighting, GI requires simulating multiple light bounces and combining their contributions.

Key Effects Achieved by GI

- **Soft Shadows:** Shadows that fade at the edges, resulting from scattered light.
- **Color Bleeding:** Surfaces reflecting their color onto nearby objects.
- **Ambient Occlusion:** Darkening in crevices due to limited light accessibility.

56.3 Step 1: Extend Path Tracing for GI

Global illumination builds upon path tracing by emphasizing the contribution of indirect lighting. Each ray bounce samples light from the scene, accumulating the effects of multiple reflections.

Ray Generation Shader with GI

The ray generation shader manages multiple bounces and gathers both direct and indirect light contributions.

```

1  // Ray generation shader for global illumination
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(set = 0, binding = 0) uniform accelerationStructureEXT topLevelAS;
7  layout(location = 0) rayPayloadEXT vec3 accumulatedColor;
8
9  layout(location = 1) uniform vec3 cameraOrigin;
10 layout(location = 2) uniform vec3 cameraDirection;
11 layout(location = 3) uniform int maxBounces;
12
13 void main() {
14     vec3 rayOrigin = cameraOrigin;
15     vec3 rayDir = calculatePrimaryRayDirection(gl_LaunchIDEXT.xy, cameraDirection);
16
17     accumulatedColor = vec3(0.0);
18     vec3 throughput = vec3(1.0);
19
20     for (int bounce = 0; bounce < maxBounces; ++bounce) {
21         traceRayEXT(topLevelAS, RAY_FLAGS_NONE_EXT, 0xFF, 0, 1, 0, rayOrigin, 0.001, rayDir, 1e32, 0);
22
23         if (terminateRay()) {
24             break;
25         }
26
27         vec3 directLighting = sampleDirectLight(rayOrigin, rayDir);
28         accumulatedColor += throughput * directLighting;
29
30         throughput *= computeMaterialThroughput();
31         rayOrigin = nextRayOrigin();
32         rayDir = nextRayDirection();
33     }
34
35     imageStore(outputImage, ivec2(gl_LaunchIDEXT.xy), vec4(accumulatedColor, 1.0));
36 }

```

Closest Hit Shader with Indirect Light Sampling

The closest hit shader calculates indirect lighting contributions by sampling reflected light from nearby surfaces.

```

1  // Closest hit shader for global illumination
2
3  #version 460
4  #extension GL_EXT_ray_tracing : require
5
6  layout(location = 0) rayPayloadInEXT vec3 accumulatedColor;

```

```

7
8 void main() {
9     vec3 normal = normalize(hitNormal);
10    vec3 viewDir = normalize(rayOrigin - hitPosition);
11
12    // Sample direct light
13    vec3 directLight = sampleLightSources(hitPosition, normal);
14
15    // Sample indirect light using a reflected ray
16    vec3 indirectLight = sampleIndirectLight(hitPosition, normal, viewDir);
17
18    accumulatedColor = directLight + indirectLight;
19 }

```

56.4 Step 2: Implement Light Sampling Techniques

Global illumination requires efficient light sampling methods to reduce noise and enhance performance. Two common techniques are:

- **Importance Sampling:** Focus ray directions on significant light contributions (e.g., towards light sources).
- **Hemisphere Sampling:** Randomly sample directions in a hemisphere oriented around the surface normal.

```

1 // Importance sampling for light sources
2 vec3 sampleLightSources(vec3 position, vec3 normal) {
3     vec3 lightDir = normalize(lightPosition - position);
4     float NdotL = max(dot(normal, lightDir), 0.0);
5     return lightColor * NdotL / length(lightPosition - position);
6 }
7
8 // Hemisphere sampling for indirect light
9 vec3 sampleIndirectLight(vec3 position, vec3 normal, vec3 viewDir) {
10    vec3 randomDir = sampleHemisphere(normal);
11    vec3 rayOrigin = position + randomDir * 0.001;
12    vec3 rayDir = randomDir;
13
14    vec3 indirectColor;
15    traceRayEXT(topLevelAS, RAY_FLAGS_NONE_EXT, 0xFF, 0, 1, 0, rayOrigin, 0.001, rayDir, 1e32, 0);
16    return indirectColor;
17 }

```

56.5 Step 3: Optimize for Performance

Simulating global illumination can be computationally expensive. Consider the following optimizations:

1. **Denoising:** Apply spatial and temporal denoising filters to reduce noise caused by low sampling rates.
2. **Adaptive Sampling:** Stop tracing rays that contribute little to the final image.
3. **Importance Sampling:** Direct more rays towards significant light sources or reflective areas.
4. **Lower Bounce Count:** Limit the maximum number of ray bounces to reduce computational cost.

56.6 Step 4: Integrate into Vulkan's Render Loop

Modify your Vulkan application to handle the extended ray tracing pipeline and additional shader resources.

```

1  // Update descriptor sets to include light properties
2  VkDescriptorBufferInfo lightBufferInfo{0};
3  lightBufferInfo.buffer = lightBuffer;
4  lightBufferInfo.offset = 0;
5  lightBufferInfo.range = sizeof(LightProperties);
6
7  VkWriteDescriptorSet writeLight{0};
8  writeLight.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
9  writeLight.dstSet = descriptorSet;
10 writeLight.dstBinding = 1;
11 writeLight.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
12 writeLight.pBufferInfo = &lightBufferInfo;
13
14 vkUpdateDescriptorSets(device, 1, &writeLight, 0, nullptr);
15
16 // Record ray tracing commands
17 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rayTracingPipeline);
18 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, pipelineLayout, 0, 1, &
19     ↳ descriptorSet, 0, nullptr);
19 vkCmdTraceRaysKHR(commandBuffer, &rayGenRegion, &missRegion, &hitRegion, &callableRegion, width, height, 1);

```

56.7 Summary

This chapter introduced the implementation of global illumination using Vulkan, extending the ray tracing pipeline to handle indirect lighting and realistic light bounces. While computationally demanding, GI significantly improves scene realism by simulating natural light behavior. By employing optimizations such as importance sampling and denoising, you can achieve visually appealing results with acceptable performance.

57

Implement Vulkan Video Decode: Use Vulkan for video decoding

57.1 Introduction

Vulkan's capabilities extend beyond graphics rendering and compute workloads to include video decoding, a feature that enables efficient hardware-accelerated video playback. This chapter introduces the Vulkan Video extension, showing you how to decode video streams using Vulkan. By leveraging GPU hardware for decoding, Vulkan offers high performance and low latency for video playback in applications such as media players, game engines, and real-time streaming.

57.2 What is Vulkan Video?

Vulkan Video is an extension to the Vulkan API that supports video decoding and encoding. It provides access to GPU video hardware to offload intensive operations from the CPU, resulting in improved performance and reduced power consumption.

Key Features of Vulkan Video

- **Hardware Acceleration:** Decoding is performed directly on the GPU.
- **Pipeline Integration:** Decoded video frames can be seamlessly used in rendering pipelines.
- **Extensibility:** Support for various video codecs, such as H.264 and H.265 (HEVC).

57.3 Prerequisites

Before starting, ensure the following:

- A Vulkan implementation with support for the Vulkan Video extension.
- Necessary Vulkan Video headers and libraries.
- A basic understanding of Vulkan queues, command buffers, and memory management.

57.4 Step 1: Enable Vulkan Video Extensions

Start by enabling the Vulkan Video extensions when creating the Vulkan instance and device. The required extensions include:

- `VK_KHR_video_queue`
- `VK_KHR_video_decode_queue`
- Codec-specific extensions like `VK_KHR_video_decode_h264` or `VK_KHR_video_decode_h265`.

```

1 // Enable Vulkan Video extensions
2 const char* videoExtensions[] = {
3     "VK_KHR_video_queue",
4     "VK_KHR_video_decode_queue",
5     "VK_KHR_video_decode_h264" // For H.264 codec support
6 };
7
8 VkInstanceCreateInfo instanceCreateInfo = {};
9 instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
10 instanceCreateInfo.enabledExtensionCount = sizeof(videoExtensions) / sizeof(videoExtensions[0]);
11 instanceCreateInfo.ppEnabledExtensionNames = videoExtensions;
12
13 vkCreateInstance(&instanceCreateInfo, nullptr, &instance);

```

57.5 Step 2: Set Up Video Decode Resources

Create a Video Session

The video session represents the decoding context and is initialized with codec-specific parameters.

```

1 // Video session creation for H.264 decoding
2 VkVideoSessionCreateInfoKHR videoSessionInfo = {};
3 videoSessionInfo.sType = VK_STRUCTURE_TYPE_VIDEO_SESSION_CREATE_INFO_KHR;
4 videoSessionInfo.videoCodecOperation = VK_VIDEO_CODEC_OPERATION_DECODE_H264_BIT_KHR;
5 videoSessionInfo.queueFamilyIndex = videoQueueFamilyIndex;
6
7 VkVideoSessionKHR videoSession;
8 vkCreateVideoSessionKHR(device, &videoSessionInfo, nullptr, &videoSession);

```

Allocate Video Session Memory

Allocate memory to store intermediate decoding data, such as bitstream buffers and reference frames.

```
1 // Allocate memory for video session
2 VkVideoSessionMemoryRequirementsKHR memoryRequirements = {};
3 vkGetVideoSessionMemoryRequirementsKHR(device, videoSession, &memoryRequirements);
4
5 VkDeviceMemory videoMemory;
6 VkMemoryAllocateInfo memoryAllocateInfo = {};
7 memoryAllocateInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
8 memoryAllocateInfo.allocationSize = memoryRequirements.size;
9 vkAllocateMemory(device, &memoryAllocateInfo, nullptr, &videoMemory);
10 vkBindVideoSessionMemoryKHR(device, videoSession, 1, &videoMemory);
```

57.6 Step 3: Prepare the Video Decode Pipeline

The decode pipeline processes the input bitstream and outputs decoded frames.

```
1 // Create video decode pipeline
2 VkVideoDecodeInfoKHR decodeInfo = {};
3 decodeInfo.sType = VK_STRUCTURE_TYPE_VIDEO_DECODE_INFO_KHR;
4 decodeInfo.srcBuffer = inputBitstreamBuffer;
5 decodeInfo.dstPictureResource = outputImage;
6
7 vkCmdDecodeVideoKHR(commandBuffer, &decodeInfo);
```

57.7 Step 4: Submit Video Decode Commands

Record and submit commands for decoding video frames.

```
1 // Record decode commands
2 vkCmdBeginVideoCodingKHR(commandBuffer, &beginCodingInfo);
3 vkCmdDecodeVideoKHR(commandBuffer, &decodeInfo);
4 vkCmdEndVideoCodingKHR(commandBuffer, &endCodingInfo);
5
6 // Submit the command buffer
7 VkSubmitInfo submitInfo = {};
8 submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
9 submitInfo.commandBufferCount = 1;
10 submitInfo.pCommandBuffers = &commandBuffer;
11
12 vkQueueSubmit(videoQueue, 1, &submitInfo, VK_NULL_HANDLE);
13 vkQueueWaitIdle(videoQueue);
```

57.8 Step 5: Use Decoded Frames in Rendering

Once decoded, the video frames are stored in GPU memory as textures. These textures can be directly sampled in shaders or composited in rendering pipelines.

```
1 // Use the decoded frame as a texture
```

```
2 VkDescriptorImageInfo imageInfo = {};  
3 imageInfo.imageView = decodedFrameImageView;  
4 imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;  
5  
6 vkUpdateDescriptorSets(device, 1, &writeDescriptor, 0, nullptr);
```

57.9 Optimizations

To improve performance and reduce latency:

1. **Batch Decode Operations:** Decode multiple frames in a single batch.
2. **Use Asynchronous Queues:** Utilize separate queues for video decode and graphics tasks.
3. **Pipeline Integration:** Avoid unnecessary data transfers by directly using decoded textures in Vulkan pipelines.

57.10 Summary

This chapter introduced the Vulkan Video extension for hardware-accelerated video decoding. By leveraging Vulkan's low-level API, you can achieve efficient video playback while integrating decoded frames into rendering pipelines. With this knowledge, you can build applications that handle complex video tasks, such as media players and real-time streaming solutions.

58

Render HDR Content: Render high-dynamic-range content

58.1 Introduction

High-Dynamic-Range (HDR) rendering significantly enhances visual quality by representing a broader range of colors and luminosity than standard-dynamic-range (SDR) rendering. HDR content is especially impactful in applications such as gaming, cinematic rendering, and virtual reality. This chapter demonstrates how to enable and render HDR content using Vulkan, covering the necessary adjustments to your swapchain, framebuffers, and shaders to take full advantage of HDR displays.

58.2 What is HDR Rendering?

HDR rendering allows you to:

- **Display Bright Highlights and Deep Shadows:** Represent a wide range of brightness values in a scene.
- **Use Extended Color Gamuts:** Support richer and more vibrant colors with formats like Rec. 2020.
- **Provide Realism:** Mimic how light behaves in the real world, leading to more realistic scenes.

58.3 Step 1: Configure the Swapchain for HDR

To enable HDR, ensure that your Vulkan application outputs to a swapchain that supports HDR-capable formats, such as `VK_FORMAT_R16G16B16A16_SFLOAT`. This format provides a higher precision for color values, enabling better representation of HDR content.

```

1 // Choose an HDR-compatible swapchain format
2 VkSurfaceFormatKHR desiredFormat = {};
3 desiredFormat.format = VK_FORMAT_R16G16B16A16_SFLOAT;
4 desiredFormat.colorSpace = VK_COLOR_SPACE_HDR10_ST2084_EXT;
5
6 VkSwapchainCreateInfoKHR swapchainCreateInfo = {};
7 swapchainCreateInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
8 swapchainCreateInfo.surface = surface;
9 swapchainCreateInfo.imageFormat = desiredFormat.format;
10 swapchainCreateInfo.imageColorSpace = desiredFormat.colorSpace;
11 // Additional swapchain settings...
12 vkCreateSwapchainKHR(device, &swapchainCreateInfo, nullptr, &swapchain);

```

58.4 Step 2: Use HDR-Compatible Framebuffers

HDR rendering requires framebuffers that can store high-precision color data. Allocate image views with the same HDR-compatible format as the swapchain.

```

1 // Create framebuffer with HDR format
2 VkImageCreateInfo imageCreateInfo = {};
3 imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
4 imageCreateInfo.format = VK_FORMAT_R16G16B16A16_SFLOAT;
5 imageCreateInfo.usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_TRANSFER_SRC_BIT;
6 imageCreateInfo.extent = { width, height, 1 };
7
8 VkImage hdrImage;
9 vkCreateImage(device, &imageCreateInfo, nullptr, &hdrImage);
10
11 // Create an image view for the framebuffer
12 VkImageViewCreateInfo viewCreateInfo = {};
13 viewCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
14 viewCreateInfo.image = hdrImage;
15 viewCreateInfo.format = VK_FORMAT_R16G16B16A16_SFLOAT;
16
17 VkImageView hdrImageView;
18 vkCreateImageView(device, &viewCreateInfo, nullptr, &hdrImageView);

```

58.5 Step 3: Implement Tone Mapping in Shaders

HDR content needs to be tone-mapped to fit within the display's luminance range. A tone mapping algorithm, such as Reinhard or ACES, compresses HDR color values to a displayable range while preserving visual details.

```

1 // Vertex Shader
2 #version 450
3 layout(location = 0) in vec3 inPosition;
4 layout(location = 1) in vec2 inTexCoord;
5
6 layout(location = 0) out vec2 fragTexCoord;

```

```

7
8 void main() {
9     gl_Position = vec4(inPosition, 1.0);
10    fragTexCoord = inTexCoord;
11 }
12
13 // Fragment Shader
14 #version 450
15 layout(location = 0) in vec2 fragTexCoord;
16 layout(location = 0) out vec4 outColor;
17
18 layout(binding = 0) uniform sampler2D hdrTexture;
19
20 vec3 ReinhardToneMapping(vec3 color) {
21     return color / (1.0 + color);
22 }
23
24 void main() {
25     vec3 hdrColor = texture(hdrTexture, fragTexCoord).rgb;
26     vec3 toneMappedColor = ReinhardToneMapping(hdrColor);
27     outColor = vec4(toneMappedColor, 1.0);
28 }

```

58.6 Step 4: Set Proper Color Spaces and Metadata

HDR rendering requires careful management of color spaces. Use Vulkan extensions to set color space metadata, ensuring the display interprets your HDR content correctly.

```

1 // Set HDR metadata for the swapchain
2 VkHdrMetadataEXT hdrMetadata = {0};
3 hdrMetadata.sType = VK_STRUCTURE_TYPE_HDR_METADATA_EXT;
4 hdrMetadata.displayPrimaryRed = { 0.680, 0.320 };
5 hdrMetadata.displayPrimaryGreen = { 0.265, 0.690 };
6 hdrMetadata.displayPrimaryBlue = { 0.150, 0.060 };
7 hdrMetadata.maxLuminance = 1000.0;
8 hdrMetadata.minLuminance = 0.01;
9 hdrMetadata.maxContentLightLevel = 1000.0;
10 hdrMetadata.maxFrameAverageLightLevel = 200.0;
11
12 vkSetHdrMetadataEXT(device, swapchainImageCount, swapchainImages, &hdrMetadata);

```

58.7 Step 5: Debug and Test HDR Output

Testing HDR output requires an HDR-compatible display and proper drivers. Tools such as debug overlays and color analysis utilities can help verify luminance and color output.

Debugging Tips

- Ensure the swapchain format and color space match the display's capabilities.
- Verify tone mapping output by visualizing luminance histograms.
- Use tools like RenderDoc to inspect HDR textures.

58.8 Summary

This chapter covered the steps required to render high-dynamic-range content with Vulkan, from configuring the swapchain and framebuffers to implementing tone mapping and setting metadata. By incorporating HDR, you can dramatically enhance the visual quality of your Vulkan applications, making them more immersive and realistic.

DRAFT
version
0.681

59

Add Vulkan Mesh Shaders: Experiment with next-gen mesh shaders

59.1 Introduction

Mesh shaders are a modern replacement for the traditional vertex and geometry shaders, offering a more flexible and efficient pipeline for processing complex geometry. They allow developers to manage geometry in a programmable way, enabling dynamic culling, level-of-detail adjustments, and efficient data handling. This chapter introduces Vulkan mesh shaders, explaining how to enable and implement them in your application.

59.2 What Are Mesh Shaders?

Mesh shaders are part of the **task-mesh pipeline**, replacing the traditional vertex processing stages. Instead of handling geometry sequentially, mesh shaders allow developers to process geometry in batches, directly generating primitives for rasterization. This approach provides several benefits:

- **Dynamic Geometry Processing:** Perform frustum culling, level-of-detail (LOD) selection, and instancing entirely on the GPU.
- **Improved Efficiency:** Reduce CPU-to-GPU communication by processing geometry data on the GPU.
- **Parallel Processing:** Leverage the GPU's compute capabilities for highly parallel geometry processing.

59.3 Step 1: Enable Mesh Shader Extensions

To use mesh shaders, enable the required Vulkan extensions during instance and device creation. The key extensions include:

- `VK_EXT_mesh_shader`

```

1 // Enable Vulkan Mesh Shader extension
2 const char* meshShaderExtensions[] = {
3     "VK_EXT_mesh_shader"
4 };
5
6 VkDeviceCreateInfo deviceCreateInfo = {};
7 deviceCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
8 deviceCreateInfo.enabledExtensionCount = sizeof(meshShaderExtensions) / sizeof(meshShaderExtensions[0]);
9 deviceCreateInfo.ppEnabledExtensionNames = meshShaderExtensions;
10
11 vkCreateDevice(physicalDevice, &deviceCreateInfo, nullptr, &device);

```

59.4 Step 2: Create a Pipeline with Mesh Shaders

Mesh shaders require a new type of pipeline that includes task and mesh shader stages.

```

1 // Specify task and mesh shader stages
2 VkPipelineShaderStageCreateInfo taskShaderStage = {};
3 taskShaderStage.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
4 taskShaderStage.stage = VK_SHADER_STAGE_TASK_BIT_EXT;
5 taskShaderStage.module = taskShaderModule;
6 taskShaderStage.pName = "main";
7
8 VkPipelineShaderStageCreateInfo meshShaderStage = {};
9 meshShaderStage.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
10 meshShaderStage.stage = VK_SHADER_STAGE_MESH_BIT_EXT;
11 meshShaderStage.module = meshShaderModule;
12 meshShaderStage.pName = "main";
13
14 // Configure pipeline stages
15 VkPipelineShaderStageCreateInfo shaderStages[] = { taskShaderStage, meshShaderStage };
16
17 // Create pipeline
18 VkGraphicsPipelineCreateInfo pipelineInfo = {};
19 pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
20 pipelineInfo.stageCount = sizeof(shaderStages) / sizeof(shaderStages[0]);
21 pipelineInfo.pStages = shaderStages;
22 // Additional pipeline configurations...
23
24 vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &pipelineInfo, nullptr, &pipeline);

```

59.5 Step 3: Write Mesh and Task Shaders

The task shader distributes work to mesh shaders, while the mesh shader generates geometry and outputs primitives.

Task Shader (GLSL)

```

1 // Task Shader
2 #version 450
3 #extension GL_EXT_mesh_shader : enable
4
5 layout(local_size_x=32) in;
6
7 void main() {
8     // Dispatch work to mesh shaders
9     if (gl_LocalInvocationID.x == 0) {
10         gl_TaskCount = 1; // One mesh shader group
11     }
12 }

```

Mesh Shader (GLSL)

```

1 // Mesh Shader
2 #version 450
3 #extension GL_EXT_mesh_shader : enable
4
5 layout(local_size_x=32) in;
6 layout(triangles) out;
7 layout(max_vertices=3, max_primitives=1) out;
8
9 out gl_MeshPerVertexEXT {
10     vec4 gl_Position;
11 } vertices[];
12
13 void main() {
14     // Define a triangle
15     vertices[0].gl_Position = vec4(-0.5, -0.5, 0.0, 1.0);
16     vertices[1].gl_Position = vec4(0.5, -0.5, 0.0, 1.0);
17     vertices[2].gl_Position = vec4(0.0, 0.5, 0.0, 1.0);
18
19     // Emit primitives
20     gl_PrimitiveIndicesEXT[0] = 0;
21     gl_PrimitiveIndicesEXT[1] = 1;
22     gl_PrimitiveIndicesEXT[2] = 2;
23
24     gl_PrimitiveCountEXT = 1; // Emit one triangle
25 }

```

59.6 Step 4: Dispatch Mesh Shaders

Use the `vkCmdDrawMeshTasksEXT` command to dispatch task and mesh shaders.

```

1 // Dispatch task and mesh shaders
2 vkCmdDrawMeshTasksEXT(commandBuffer, 1, 0);

```

59.7 Step 5: Debug and Optimize

Mesh shaders provide powerful tools, but their efficiency depends on proper usage:

- **Cull Early:** Use the task shader for frustum and back-face culling.
- **Batch Geometry:** Group geometry processing to maximize parallelism.

- **Test Performance:** Use GPU performance counters to measure throughput and bottlenecks.

59.8 Summary

This chapter introduced Vulkan mesh shaders and their integration into the graphics pipeline. By enabling flexible geometry processing and reducing CPU-GPU communication, mesh shaders are ideal for rendering complex scenes efficiently. Experiment with task and mesh shaders to explore their potential for your applications.

DRAFT
version
0.681

60

Add Tiled Rendering: Use tile-based techniques for mobile GPUs

60.1 Introduction

Tiled rendering is a technique used by many mobile GPUs to optimize rendering by dividing the framebuffer into smaller tiles and processing each tile independently. This approach minimizes memory bandwidth usage and improves performance on mobile devices. Vulkan provides mechanisms to explicitly control rendering within tiles, allowing you to take full advantage of tile-based architectures.

In this chapter, you will learn how to implement tiled rendering using Vulkan's render passes and subpasses, while understanding its benefits and trade-offs.

60.2 What is Tiled Rendering?

Tiled rendering splits the framebuffer into smaller, manageable regions called tiles. Rendering operations for each tile are performed in a high-speed on-chip memory, reducing costly reads and writes to system memory. This technique offers:

- **Reduced Memory Bandwidth:** Minimized interaction with external memory.
- **Improved Cache Coherency:** Each tile fits into local memory, avoiding frequent cache invalidations.
- **Energy Efficiency:** Particularly beneficial for battery-powered devices.

60.3 Step 1: Design Render Passes for Tiled Rendering

Vulkan render passes and subpasses naturally align with tile-based rendering. Subpasses allow for multiple rendering steps within a single tile, reducing memory usage between passes.

```

1 // Configure a render pass with subpasses
2 VkAttachmentDescription colorAttachment = {};
3 colorAttachment.format = swapchainImageFormat;
4 colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
5 colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
6 colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
7 colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
8 colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
9
10 VkAttachmentReference colorAttachmentRef = {};
11 colorAttachmentRef.attachment = 0;
12 colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
13
14 VkSubpassDescription subpass = {};
15 subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
16 subpass.colorAttachmentCount = 1;
17 subpass.pColorAttachments = &colorAttachmentRef;
18
19 VkRenderPassCreateInfo renderPassInfo = {};
20 renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
21 renderPassInfo.attachmentCount = 1;
22 renderPassInfo.pAttachments = &colorAttachment;
23 renderPassInfo.subpassCount = 1;
24 renderPassInfo.pSubpasses = &subpass;
25
26 VkRenderPass renderPass;
27 vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass);

```

60.4 Step 2: Utilize Load and Store Operations

Tiled rendering benefits from careful use of attachment load and store operations. Use `VK_ATTACHMENT_LOAD_OP_CLEAR` to clear tiles efficiently at the start of a render pass and `VK_ATTACHMENT_STORE_OP_DONT_CARE` for intermediate passes to avoid unnecessary memory writes.

```

1 // Use clear and discard operations for tiled rendering
2 colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
3 colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;

```

60.5 Step 3: Optimize Shaders for Tiles

When working with tiles, ensure your shaders are efficient and avoid unnecessary memory accesses. Use Vulkan's input attachments to pass data between subpasses, reducing dependency on global memory.

```

1 // Fragment Shader (Subpass Input Example)
2 #version 450
3 layout(set = 0, binding = 0, input_attachment_index = 0) uniform subpassInput inputColor;
4
5 layout(location = 0) out vec4 outColor;

```

```

6
7 void main() {
8     vec4 color = subpassLoad(inputColor);
9     outColor = color * vec4(0.8, 0.8, 1.0, 1.0); // Apply a blue tint
10 }

```

60.6 Step 4: Manage Tile-Specific Resources

Tile-based GPUs benefit from minimal resource overhead per tile. Avoid excessive resource bindings and prefer smaller, localized datasets. Use Vulkan descriptor sets efficiently to group resources used per tile.

```

1 // Bind tile-specific resources
2 VkDescriptorSetAllocateInfo allocInfo = {};
3 allocInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
4 allocInfo.descriptorPool = descriptorPool;
5 allocInfo.descriptorSetCount = 1;
6 allocInfo.pSetLayouts = &descriptorSetLayout;
7
8 VkDescriptorSet descriptorSet;
9 vkAllocateDescriptorSets(device, &allocInfo, &descriptorSet);
10
11 // Update descriptor sets with tile-specific data
12 VkWriteDescriptorSet descriptorWrite = {};
13 descriptorWrite.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
14 descriptorWrite.dstSet = descriptorSet;
15 descriptorWrite.dstBinding = 0;
16 descriptorWrite.dstArrayElement = 0;
17 descriptorWrite.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
18 descriptorWrite.descriptorCount = 1;
19 descriptorWrite.pBufferInfo = &bufferInfo;
20
21 vkUpdateDescriptorSets(device, 1, &descriptorWrite, 0, nullptr);

```

60.7 Step 5: Test and Debug Tiled Rendering

Testing tiled rendering on mobile devices is crucial to understanding its impact. Use GPU profiling tools, such as ARM's Mali Graphics Debugger or Qualcomm's Snapdragon Profiler, to analyze tile memory usage and rendering efficiency.

Debugging Tips

- **Overdraw:** Minimize overdraw by sorting geometry front-to-back.
- **Load Balancing:** Ensure tiles process an even workload to avoid bottlenecks.
- **Render Pass Configuration:** Validate your subpass configurations for tile-based efficiency.

60.8 Summary

Tiled rendering is a powerful optimization technique for mobile GPUs, and Vulkan's explicit control over render passes and subpasses allows you to harness its benefits. By structuring

your rendering pipeline to minimize memory usage and leverage on-chip tile processing, you can achieve significant performance improvements, especially on mobile devices. Experiment with these techniques to better understand their impact on your Vulkan applications.

DRAFT
version
0.681

Part VII

Cross-Platform Development

61

Add GLFW for Windowing: Use GLFW for cross-platform Vulkan windows

61.1 Introduction

Creating a Vulkan application often requires setting up a window to display rendered content. GLFW is a popular library that simplifies the creation of windows and handles input across multiple platforms. It provides a convenient way to integrate Vulkan with window management, making it a go-to choice for developers. In this chapter, you will learn how to set up GLFW for Vulkan-based windowing.

61.2 What is GLFW?

GLFW is a lightweight, open-source library designed for creating windows with OpenGL, Vulkan, and other rendering backends. It provides:

- **Cross-Platform Support:** Works seamlessly on Windows, macOS, and Linux.
- **Vulkan Integration:** Includes functions to interface with Vulkan directly.
- **Input Handling:** Offers an easy-to-use API for keyboard, mouse, and joystick inputs.

61.3 Step 1: Install GLFW

Before using GLFW, ensure it is installed on your development system. You can either compile it from source or install it via a package manager.

Linux

```
1 sudo apt-get install libglfw3 libglfw3-dev
```

macOS

```
1 brew install glfw
```

Windows

Download the latest precompiled binaries or build GLFW from source. Link the GLFW library to your project in your build system.

61.4 Step 2: Initialize GLFW

Start by initializing GLFW and ensuring it supports Vulkan.

```
1 // Initialize GLFW
2 if (!glfwInit()) {
3     throw std::runtime_error("Failed to initialize GLFW");
4 }
5
6 // Specify that we are using Vulkan
7 glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
8
9 // Optional: Disable window resizing
10 glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

61.5 Step 3: Create a GLFW Window

Use GLFW to create a Vulkan-compatible window.

```
1 // Create a window
2 GLFWwindow* window = glfwCreateWindow(800, 600, "Vulkan Window", nullptr, nullptr);
3 if (!window) {
4     glfwTerminate();
5     throw std::runtime_error("Failed to create GLFW window");
6 }
```

GLFW provides a pointer to the window object, which you can use to manage the window lifecycle and handle user inputs.

61.6 Step 4: Retrieve Vulkan Extensions from GLFW

GLFW simplifies the process of querying Vulkan extensions required for window surface creation.

```

1 // Query Vulkan extensions required by GLFW
2 uint32_t glfwExtensionCount = 0;
3 const char** glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);
4
5 // Pass these extensions when creating the Vulkan instance
6 VkInstanceCreateInfo instanceCreateInfo = {};
7 instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
8 instanceCreateInfo.enabledExtensionCount = glfwExtensionCount;
9 instanceCreateInfo.ppEnabledExtensionNames = glfwExtensions;
10
11 VkInstance instance;
12 if (vkCreateInstance(&instanceCreateInfo, nullptr, &instance) != VK_SUCCESS) {
13     throw std::runtime_error("Failed to create Vulkan instance");
14 }

```

61.7 Step 5: Create a Vulkan Surface with GLFW

GLFW offers a helper function to create a Vulkan surface associated with its window.

```

1 // Create a Vulkan surface for the window
2 VkSurfaceKHR surface;
3 if (glfwCreateWindowSurface(instance, window, nullptr, &surface) != VK_SUCCESS) {
4     throw std::runtime_error("Failed to create Vulkan surface");
5 }

```

This surface will be used during swapchain creation, allowing Vulkan to render content directly to the window.

61.8 Step 6: Handle Window Events

GLFW provides an event polling mechanism to process user inputs and window events.

```

1 // Main loop
2 while (!glfwWindowShouldClose(window)) {
3     glfwPollEvents();
4     // Render your Vulkan content here
5 }

```

Handling events like window resizing, closing, or input can be easily achieved using GLFW's callback functions.

61.9 Step 7: Clean Up Resources

When the application ends, clean up GLFW resources to avoid memory leaks.

```

1 // Clean up
2 vkDestroySurfaceKHR(instance, surface, nullptr);
3 glfwDestroyWindow(window);
4 glfwTerminate();

```

61.10 Summary

GLFW is a powerful library that simplifies Vulkan window management and input handling across platforms. By using GLFW, you can focus on Vulkan development without worrying about platform-specific windowing code. This chapter demonstrated how to set up GLFW for Vulkan, create a window, and integrate it with a Vulkan surface. In the following chapters, we will use this setup as the foundation for rendering Vulkan content.

DRAFT
version
0.681

62

Add SDL2 for Windowing: Integrate SDL2 with Vulkan

62.1 Introduction

SDL2 (Simple DirectMedia Layer) is a versatile library widely used for game development and multimedia applications. It supports Vulkan, providing cross-platform functionality for window creation, input handling, and other essential features. In this chapter, we will explore how to integrate SDL2 with Vulkan for window management, laying a robust foundation for cross-platform Vulkan applications.

62.2 Why SDL2?

SDL2 offers several advantages that make it a popular choice for Vulkan applications:

- **Cross-Platform:** SDL2 works seamlessly on Windows, macOS, Linux, and other platforms.
- **Vulkan Support:** It provides specific APIs for Vulkan instance extensions and surface creation.
- **Input Handling:** Supports keyboard, mouse, and joystick input.
- **Event System:** Efficiently handles window and input events.

62.3 Step 1: Install SDL2

First, install SDL2 on your development machine.

Linux

```
1 sudo apt-get install libsdl2-dev
```

macOS

```
1 brew install sdl2
```

Windows

Download the precompiled binaries from the SDL2 website or install via package manager if available.

62.4 Step 2: Initialize SDL2

Initialize SDL2 and configure it for Vulkan.

```
1 // Initialize SDL2
2 if (SDL_Init(SDL_INIT_VIDEO) != 0) {
3     throw std::runtime_error("Failed to initialize SDL2");
4 }
5
6 // Configure SDL2 to use Vulkan
7 SDL_Window* window = SDL_CreateWindow(
8     "Vulkan Window",
9     SDL_WINDOWPOS_CENTERED,
10    SDL_WINDOWPOS_CENTERED,
11    800, 600,
12    SDL_WINDOW_VULKAN
13 );
14
15 if (!window) {
16     throw std::runtime_error("Failed to create SDL2 window");
17 }
```

62.5 Step 3: Retrieve Vulkan Extensions from SDL2

SDL2 provides a function to query the Vulkan extensions required for window surface creation. Pass these extensions when creating your Vulkan instance.

```
1 // Get Vulkan extensions required by SDL2
2 unsigned int sdlExtensionCount = 0;
3 if (SDL_Vulkan_GetInstanceExtensions(window, &sdlExtensionCount, nullptr)) {
4     throw std::runtime_error("Failed to get Vulkan extensions from SDL2");
5 }
6
7 std::vector<const char*> sdlExtensions(sdlExtensionCount);
8 if (SDL_Vulkan_GetInstanceExtensions(window, &sdlExtensionCount, sdlExtensions.data())) {
9     throw std::runtime_error("Failed to get Vulkan extensions from SDL2");
10 }
11
12 // Pass extensions to Vulkan instance creation
```

```

13 VkInstanceCreateInfo instanceCreateInfo = {};
14 instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
15 instanceCreateInfo.enabledExtensionCount = sdlExtensionCount;
16 instanceCreateInfo.ppEnabledExtensionNames = sdlExtensions.data();
17
18 VkInstance instance;
19 if (vkCreateInstance(&instanceCreateInfo, nullptr, &instance) != VK_SUCCESS) {
20     throw std::runtime_error("Failed to create Vulkan instance");
21 }

```

62.6 Step 4: Create a Vulkan Surface with SDL2

SDL2 provides a convenient function to create a Vulkan-compatible surface for its window.

```

1 // Create a Vulkan surface for the SDL2 window
2 VkSurfaceKHR surface;
3 if (!SDL_Vulkan_CreateSurface(window, instance, &surface)) {
4     throw std::runtime_error("Failed to create Vulkan surface with SDL2");
5 }

```

This surface will be used during swapchain creation to render Vulkan content to the SDL2 window.

62.7 Step 5: Handle SDL2 Events

SDL2's event system allows you to handle window events, such as resizing, and input events like key presses and mouse clicks.

```

1 // Main event loop
2 bool running = true;
3 SDL_Event event;
4
5 while (running) {
6     while (SDL_PollEvent(&event)) {
7         if (event.type == SDL_QUIT) {
8             running = false;
9         } else if (event.type == SDL_WINDOWEVENT) {
10             if (event.window.event == SDL_WINDOWEVENT_RESIZED) {
11                 // Handle window resizing
12                 int width = event.window.data1;
13                 int height = event.window.data2;
14                 // Recreate Vulkan swapchain here
15             }
16         }
17     }
18
19     // Render your Vulkan content here
20 }

```

62.8 Step 6: Clean Up Resources

When your application is ready to exit, clean up Vulkan and SDL2 resources.


```
1 // Clean up
2 vkDestroySurfaceKHR(instance, surface, nullptr);
3 vkDestroyInstance(instance, nullptr);
4 SDL_DestroyWindow(window);
5 SDL_Quit();
```

62.9 Summary

SDL2 is a powerful library that simplifies Vulkan windowing and input handling across multiple platforms. In this chapter, you learned how to use SDL2 to create a Vulkan-compatible window, retrieve the necessary Vulkan extensions, and create a surface for rendering. With SDL2's comprehensive feature set, you can efficiently manage windowing and user input in your Vulkan applications.

63

Cross-Compile for Android: Build a Vulkan app for Android

63.1 Introduction

Building Vulkan applications for Android involves setting up a cross-compilation environment and adapting your application for mobile GPUs and the Android platform. This chapter guides you through the process of setting up your development environment, configuring the Android Native Development Kit (NDK), and running a Vulkan-based app on an Android device.

63.2 Overview of Vulkan on Android

Android devices with Vulkan support allow developers to access advanced GPU features for high-performance graphics and compute tasks. To run a Vulkan app on Android, you'll need to:

- Set up the Android development environment.
- Write an Android-specific Vulkan initialization code.
- Cross-compile your application using the Android NDK.

63.3 Step 1: Set Up the Android Development Environment

Install the required tools for Android development:

1. **Android Studio:** The primary IDE for Android development. Download and install it from <https://developer.android.com/studio>.
2. **Android NDK:** A set of tools for building native code for Android. Install it via the SDK Manager in Android Studio or download it separately.
3. **Vulkan SDK:** Install the Vulkan SDK for Android from <https://vulkan.lunarg.com>.

Ensure you have the necessary drivers and runtime libraries for Vulkan on your target Android device.

63.4 Step 2: Create an Android Project

Create a new Android project using Android Studio:

1. Open Android Studio and select **New Project**.
2. Choose an **Empty Activity**.
3. Configure the project with a unique package name and select **C++ Support**.
4. Choose a minimum API level that supports Vulkan (e.g., API 24 or higher).

Android Studio generates a basic project structure, including C++ support with an `Android.mk` or `CMakeLists.txt` file.

63.5 Step 3: Write Vulkan Initialization Code for Android

Android uses the `ANativeWindow` system for Vulkan surfaces. Update your app's native code to initialize Vulkan on Android:

```

1 // Obtain the native window
2 ANativeWindow* nativeWindow = app->window;
3
4 // Create a Vulkan surface for Android
5 VkAndroidSurfaceCreateInfoKHR surfaceCreateInfo = {};
6 surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_ANDROID_SURFACE_CREATE_INFO_KHR;
7 surfaceCreateInfo.window = nativeWindow;
8
9 VkSurfaceKHR surface;
10 if (vkCreateAndroidSurfaceKHR(instance, &surfaceCreateInfo, nullptr, &surface) != VK_SUCCESS) {
11     throw std::runtime_error("Failed to create Vulkan surface on Android");
12 }
```

You will also need to ensure the Vulkan instance is created with the required Android-specific extensions:

```

1 const std::vector<const char*> instanceExtensions = {
2     VK_KHR_SURFACE_EXTENSION_NAME,
3     VK_KHR_ANDROID_SURFACE_EXTENSION_NAME
4 };
```

63.6 Step 4: Configure Build Files

Edit the `CMakeLists.txt` or `Android.mk` file to include Vulkan headers and link against the Vulkan loader:

For `CMakeLists.txt` :

```
1 find_package(Vulkan REQUIRED)
2
3 add_library(native-lib SHARED
4     src/main/cpp/native-lib.cpp)
5
6 target_include_directories(native-lib PRIVATE
7     ${Vulkan_INCLUDE_DIRS})
8
9 target_link_libraries(native-lib Vulkan::Vulkan android log)
```

For `Android.mk` :

```
1 LOCAL_PATH := $(call my-dir)
2
3 include $(CLEAR_VARS)
4
5 LOCAL_MODULE := native-lib
6 LOCAL_SRC_FILES := native-lib.cpp
7
8 LOCAL_LDLIBS := -lvulkan -landroid -llog
9
10 include $(BUILD_SHARED_LIBRARY)
```

63.7 Step 5: Cross-Compile the Application

Build the project using Android Studio or the command line:

```
1 // Build the project using Gradle
2 ./gradlew assembleDebug
```

Ensure the generated APK includes your native Vulkan libraries. Check the `.so` files in the `lib/arm64-v8a/` or other architecture-specific directories.

63.8 Step 6: Deploy and Test on an Android Device

Connect your Android device to your computer and enable USB debugging. Deploy the app from Android Studio:

1. Select the connected device in the [Run/Debug Configuration](#).
2. Click [Run](#) to install and launch the app on the device.

Verify that Vulkan is running correctly by inspecting the device logs using `adb logcat` :

```
1 adb logcat | grep Vulkan
```

63.9 Step 7: Debug and Optimize

For debugging Vulkan on Android:

- Use tools like [RenderDoc](#) or [Android GPU Inspector](#).
- Check Vulkan validation layers to catch API errors.

Optimize your Vulkan app for mobile GPUs by minimizing memory usage and leveraging tile-based rendering if supported by the hardware.

63.10 Summary

In this chapter, we covered the essentials of setting up Vulkan for Android, including environment configuration, cross-compilation, and deployment. With Vulkan, you can unlock the full potential of modern Android GPUs, enabling high-performance graphics and compute operations. This setup will serve as the foundation for creating advanced Vulkan applications tailored to the Android ecosystem.

64

Test on macOS (MoltenVK): Run Vulkan apps on macOS using MoltenVK

64.1 Introduction

Apple platforms like macOS, iOS, and iPadOS do not natively support Vulkan. Instead, developers can use MoltenVK, an open-source Vulkan implementation that translates Vulkan calls into Metal, Apple's graphics API. This chapter explains how to set up your Vulkan application to run on macOS using MoltenVK, enabling cross-platform Vulkan development and testing.

64.2 Why Use MoltenVK?

MoltenVK serves as a compatibility layer, allowing Vulkan applications to run on Apple devices by mapping Vulkan calls to Metal. Benefits of using MoltenVK include:

- Cross-platform development without rewriting code for Metal.
- Full Vulkan 1.1 feature set support (and partial Vulkan 1.2 support).
- Seamless integration with the Vulkan SDK.
- Active support and regular updates.

64.3 Step 1: Install the Vulkan SDK with MoltenVK

To use MoltenVK, you need the Vulkan SDK installed on macOS. Follow these steps:

1. Visit <https://vulkan.lunarg.com> and download the Vulkan SDK for macOS.
2. Install the Vulkan SDK by running the downloaded package.
3. Set up environment variables in your shell configuration file (`~/zshrc` or `~/bash_profile` →):

```

1 export VULKAN_SDK=/path/to/vulkan-sdk-version/macOS
2 export PATH=$VULKAN_SDK/bin:$PATH
3 export DYLD_LIBRARY_PATH=$VULKAN_SDK/lib:$DYLD_LIBRARY_PATH
4 export VK_ICD_FILENAMES=$VULKAN_SDK/etc/vulkan/icd.d/MoltenVK_icd.json
5 export VK_LAYER_PATH=$VULKAN_SDK/etc/vulkan/explicit_layer.d

```

After editing, reload your shell configuration:

```
1 source ~/.zshrc
```

Verify the installation by running:

```
1 vulkaninfo
```

This command displays Vulkan capabilities supported via MoltenVK.

64.4 Step 2: Modify Your Vulkan Application for macOS

To ensure your Vulkan application works with MoltenVK:

- Use the `VK_KHR_surface` and `VK_MVK_macos_surface` extensions for creating macOS-specific surfaces.
- Include the Metal Surface extension in your Vulkan instance creation.

Add the macOS surface extension during Vulkan instance creation:

```

1 const std::vector<const char*> instanceExtensions = {
2     VK_KHR_SURFACE_EXTENSION_NAME,
3     VK_MVK_MACOS_SURFACE_EXTENSION_NAME
4 };
5
6 VkInstanceCreateInfo createInfo = {};
7 createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
8 createInfo.enabledExtensionCount = static_cast<uint32_t>(instanceExtensions.size());
9 createInfo.ppEnabledExtensionNames = instanceExtensions.data();

```

Create a Vulkan surface using MoltenVK's macOS surface extension:

```

1 VkMacOSSurfaceCreateInfoMVK surfaceCreateInfo = {};
2 surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_MACOS_SURFACE_CREATE_INFO_MVK;
3 surfaceCreateInfo.pView = static_cast<void*>(yourNSViewPointer);
4
5 VkSurfaceKHR surface;
6 if (vkCreateMacOSSurfaceMVK(instance, &surfaceCreateInfo, nullptr, &surface) != VK_SUCCESS) {
7     throw std::runtime_error("Failed to create Vulkan surface on macOS");
8 }

```

64.5 Step 3: Configure the Build System

Update your build system to link against MoltenVK. If you are using `CMake`, update your `CMakeLists.txt`:

```
1 find_package(Vulkan REQUIRED)
2
3 add_executable(MyVulkanApp main.cpp)
4
5 target_include_directories(MyVulkanApp PRIVATE ${Vulkan_INCLUDE_DIRS})
6 target_link_libraries(MyVulkanApp Vulkan::Vulkan)
```

Ensure that you specify the correct paths to the Vulkan SDK when running `cmake`:

```
1 cmake -DQMAKE_PREFIX_PATH=$VULKAN_SDK ..
```

64.6 Step 4: Build and Run on macOS

Compile your Vulkan application as you would on other platforms. The Vulkan SDK on macOS includes all necessary libraries for MoltenVK. Once built, run your application:

```
1 ./MyVulkanApp
```

Your Vulkan app should open a window and render content using MoltenVK.

64.7 Step 5: Debugging and Validation Layers

MoltenVK supports Vulkan's validation layers, allowing you to debug your Vulkan app on macOS. Enable validation layers during Vulkan instance creation:

```
1 const std::vector<const char*> validationLayers = {
2     "VK_LAYER_KHRONOS_validation"
3 };
4
5 VkInstanceCreateInfo createInfo = {};
6 createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
7 createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.size());
8 createInfo.ppEnabledLayerNames = validationLayers.data();
```

Run your app with debugging tools like `vulkaninfo` or [RenderDoc](#) to analyze performance and debug rendering issues.

64.8 Step 6: Optimize for Metal via MoltenVK

MoltenVK translates Vulkan calls to Metal. Optimize your application with the following considerations:

- Minimize Vulkan API calls to reduce translation overhead.

- Use pipeline caching to speed up Metal shader compilation.
- Avoid features not natively supported by Metal (e.g., some Vulkan extensions).

64.9 Summary

With MoltenVK, Vulkan developers can target macOS and expand their application's reach to Apple's ecosystem. This chapter covered the setup process, code modifications, and debugging tools necessary to run Vulkan apps on macOS. By leveraging MoltenVK, you can maintain a single Vulkan codebase while supporting cross-platform development, including Apple's platforms.

DRAFT
version
0.681

65

Cross-Compile for Raspberry Pi: Use Vulkan on a Raspberry Pi

65.1 Introduction

The Raspberry Pi, a popular single-board computer, has gained Vulkan support starting with the Raspberry Pi 4. Leveraging Vulkan on this platform enables efficient graphics and compute operations, making it a great choice for lightweight gaming, simulations, or educational projects. This chapter guides you through setting up a cross-compilation environment to build and deploy Vulkan applications on a Raspberry Pi.

65.2 Why Use Vulkan on Raspberry Pi?

Vulkan offers significant performance advantages on hardware-constrained devices like the Raspberry Pi by:

- Providing fine-grained control over the GPU.
- Allowing efficient multi-threading.
- Reducing CPU overhead compared to OpenGL.

The Vulkan driver for Raspberry Pi 4, developed by the Mesa project, supports essential Vulkan features for modern graphics applications.

65.3 Step 1: Set Up the Raspberry Pi

Begin by preparing your Raspberry Pi with the required software:

1. **Install Raspberry Pi OS:** - Download Raspberry Pi OS from <https://www.raspberrypi.org/software/operating-systems/>. - Flash it onto an SD card using a tool like [Raspberry Pi Imager](#) or [balenaEtcher](#).
2. **Update the System:** Boot into Raspberry Pi OS and update the system packages:

```
1 sudo apt update && sudo apt upgrade
2
```

3. **Install Vulkan Drivers:** Ensure the Mesa Vulkan driver (**v3dv**) is installed:

```
1 sudo apt install mesa-vulkan-drivers
2
```

Confirm Vulkan support by running:

```
1 vulkaninfo | grep v3dv
2
```

65.4 Step 2: Set Up a Cross-Compilation Environment

Cross-compilation is more efficient than building directly on the Raspberry Pi. Follow these steps:

1. **Install the Vulkan SDK on Your Host Machine:** Download and install the Vulkan SDK for your host OS from <https://vulkan.lunarg.com>.
2. **Install Cross-Compilation Tools:** Install tools for cross-compiling to ARM (e.g., GCC or Clang for ARM). On a Linux host, use:

```
1 sudo apt install gcc-arm-linux-gnueabi g++-arm-linux-gnueabi
2
```

3. **Set Up CMake Toolchain File:** Create a CMake toolchain file to configure cross-compilation:

```
1 set(CMAKE_SYSTEM_NAME Linux)
2 set(CMAKE_SYSTEM_PROCESSOR arm)
3
4 set(CMAKE_C_COMPILER arm-linux-gnueabi-gcc)
5 set(CMAKE_CXX_COMPILER arm-linux-gnueabi-g++)
6 set(CMAKE_FIND_ROOT_PATH /usr/arm-linux-gnueabi)
7
8 set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
9 set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
10 set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
11
```

Save this file as **rpi-toolchain.cmake**.

65.5 Step 3: Configure Vulkan Application for Raspberry Pi

When setting up your Vulkan application, ensure it uses the Vulkan extensions supported by the Raspberry Pi. For instance, the Raspberry Pi supports `VK_KHR_surface` and `VK_KHR_display`, which are useful for rendering without a windowing system.

Initialize a Vulkan surface using `VK_KHR_display` for direct-to-display rendering:

```

1  VkDisplayPropertiesKHR displayProperties;
2  uint32_t displayCount = 0;
3
4  // Enumerate displays
5  vkGetPhysicalDeviceDisplayPropertiesKHR(physicalDevice, &displayCount, nullptr);
6  std::vector<VkDisplayPropertiesKHR> displays(displayCount);
7  vkGetPhysicalDeviceDisplayPropertiesKHR(physicalDevice, &displayCount, displays.data());
8
9  // Choose a display (e.g., the first available)
10 VkDisplayKHR display = displays[0].display;
11
12 VkDisplaySurfaceCreateInfoKHR surfaceCreateInfo = {};
13 surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_DISPLAY_SURFACE_CREATE_INFO_KHR;
14 surfaceCreateInfo.displayMode = ...; // Obtain a compatible display mode
15 surfaceCreateInfo.imageExtent.width = 1920;
16 surfaceCreateInfo.imageExtent.height = 1080;
17
18 // Create the surface
19 VkSurfaceKHR surface;
20 vkCreateDisplayPlaneSurfaceKHR(instance, &surfaceCreateInfo, nullptr, &surface);

```

If you're using a desktop environment (like LXDE), use `VK_KHR_xcb_surface` to create a windowed surface.

65.6 Step 4: Cross-Compile the Application

Configure and build your application using CMake with the toolchain file:

```

1  cmake -DCMAKE_TOOLCHAIN_FILE=path/to/rpi-toolchain.cmake \
2  -DVulkan_INCLUDE_DIR=$VULKAN_SDK/include \
3  -DVulkan_LIBRARY=$VULKAN_SDK/lib ..
4  make

```

The resulting binaries will be built for ARM and ready to run on the Raspberry Pi.

65.7 Step 5: Deploy and Run on the Raspberry Pi

Copy the compiled binary and any necessary resources to your Raspberry Pi:

```
1  scp MyVulkanApp pi@raspberrypi.local:/home/pi/
```

Run the application on the Raspberry Pi:

```
1  ./MyVulkanApp
```


65.8 Step 6: Debugging and Performance Optimization

For debugging:

- Use **RenderDoc**, which supports Vulkan applications on Raspberry Pi.
- Enable validation layers to catch API errors.

For performance optimization:

- Reduce memory usage by optimizing buffer allocations.
- Minimize pipeline state changes and use pipeline caching.
- Leverage tile-based rendering optimizations provided by the Raspberry Pi GPU.

65.9 Summary

This chapter covered the process of cross-compiling Vulkan applications for the Raspberry Pi, from setting up a development environment to deploying and running your application. The Raspberry Pi's Vulkan support opens up new opportunities for efficient graphics and compute programming on a compact, affordable platform. With the steps outlined here, you are ready to explore Vulkan's capabilities on the Raspberry Pi.

66

Test on Linux: Set up a Vulkan development environment for Linux

66.1 Introduction

Linux is a robust and flexible platform for Vulkan development, offering comprehensive driver support, development tools, and debugging utilities. This chapter guides you through setting up a Vulkan development environment on Linux, testing your applications, and troubleshooting common issues.

66.2 Why Develop Vulkan Applications on Linux?

Linux provides several benefits for Vulkan development:

- **Open-source graphics drivers:** Modern distributions include Vulkan-capable drivers for popular GPUs, such as AMD's RADV, Intel's ANV, and NVIDIA's proprietary Vulkan drivers.
- **Development tools:** Linux offers powerful tools like [RenderDoc](#), [Vulkan Validation Layers](#), and [vulkaninfo](#).
- **Cross-platform compatibility:** Vulkan's portability makes it easy to develop on Linux and deploy to other platforms.

66.3 Step 1: Install Vulkan Drivers

Before starting development, ensure that your GPU supports Vulkan and the appropriate drivers are installed.

1. **Check GPU Compatibility:** Verify your GPU's Vulkan support on the manufacturer's website or using `vulkaninfo` (installed with the Vulkan SDK).
2. **Install Vulkan Drivers:** - For AMD GPUs, install the Mesa drivers, which include the RADV Vulkan driver:

```
1 sudo apt install mesa-vulkan-drivers
2
```

- For Intel GPUs, install the Mesa drivers as well:

```
1 sudo apt install mesa-vulkan-drivers
2
```

- For NVIDIA GPUs, download and install the proprietary Vulkan-capable driver from <https://www.nvidia.com> or use your distribution's package manager:

```
1 sudo apt install nvidia-driver vulkan-utils
2
```

3. **Verify Installation:** Run `vulkaninfo` to confirm that Vulkan is supported:

```
1 vulkaninfo | less
2
```

This command lists Vulkan devices and extensions available on your system.

66.4 Step 2: Install the Vulkan SDK

The Vulkan SDK includes essential tools, libraries, and headers for Vulkan development. Follow these steps to install it:

1. Download the Vulkan SDK from <https://vulkan.lunarg.com/sdk/home>.
2. Extract and install the SDK package. For example:

```
1 tar -xvf vulkansdk-linux-x.x.x.x.tar.gz
2
```

3. Add the Vulkan SDK to your environment variables by editing `~/.bashrc` or `~/.zshrc`:

```
1 export VULKAN_SDK=/path/to/vulkan-sdk-version/x.x.x.x/x86_64
2 export PATH=$VULKAN_SDK/bin:$PATH
3 export LD_LIBRARY_PATH=$VULKAN_SDK/lib:$LD_LIBRARY_PATH
4 export VK_ICD_FILENAMES=$VULKAN_SDK/etc/vulkan/icd.d
5 export VK_LAYER_PATH=$VULKAN_SDK/etc/vulkan/explicit_layer.d
6
```

4. Reload your shell configuration:

```

1 source ~/.bashrc
2

```

66.5 Step 3: Configure Your Build Environment

Most Linux Vulkan applications are built using **CMake**. Set up a simple `CMakeLists.txt` file for your Vulkan project:

```

1 cmake_minimum_required(VERSION 3.10)
2 project(VulkanApp)
3
4 find_package(Vulkan REQUIRED)
5
6 add_executable(VulkanApp main.cpp)
7
8 target_include_directories(VulkanApp PRIVATE ${Vulkan_INCLUDE_DIRS})
9 target_link_libraries(VulkanApp Vulkan::Vulkan)

```

Generate the build files and compile your application:

```

1 cmake -B build -S .
2 cmake --build build

```

66.6 Step 4: Create a Vulkan Surface on Linux

To create a Vulkan surface on Linux, use extensions such as `VK_KHR_xcb_surface` or `VK_KHR_xlib_surface`. For an XCB-based surface:

1. Include the Vulkan XCB header:

```

1 #include <vulkan/vulkan_xcb.h>
2

```

2. Create the surface:

```

1 VkXcbSurfaceCreateInfoKHR surfaceCreateInfo = {0};
2 surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR;
3 surfaceCreateInfo.connection = xcb_connection;
4 surfaceCreateInfo.window = xcb_window;
5
6 VkSurfaceKHR surface;
7 if (vkCreateXcbSurfaceKHR(instance, &surfaceCreateInfo, nullptr, &surface) != VK_SUCCESS) {
8     throw std::runtime_error("Failed to create Vulkan surface");
9 }
10

```

For Wayland-based systems, use `VK_KHR_wayland_surface` instead.

66.7 Step 5: Test and Debug on Linux

1. **Run Your Application:** Execute your Vulkan application on Linux. Ensure that you have appropriate permissions and dependencies:

```
1 ./build/VulkanApp
2
```

2. **Use Validation Layers:** Enable validation layers to catch Vulkan API misuse:

```
1 const std::vector<const char*> validationLayers = {
2     "VK_LAYER_KHRONOS_validation"
3 };
4
```

3. **Analyze Performance with RenderDoc:** Install and run **RenderDoc** to capture frames and analyze GPU performance:

```
1 sudo apt install renderdoc
2 renderdoccmd capture ./build/VulkanApp
3
```

4. **Check GPU Usage:** Use tools like **vulkaninfo** or **nvidia-smi** (for NVIDIA GPUs) to monitor Vulkan activity.

66.8 Summary

Linux is an excellent platform for Vulkan development, offering robust driver support and a powerful set of development tools. This chapter provided a complete walkthrough for setting up a Vulkan development environment, configuring your application, and testing on Linux. By following these steps, you can effectively develop, test, and debug Vulkan applications on this open-source operating system.

67

Use Vulkan in Unity: Integrate Vulkan into Unity projects

67.1 Introduction

Unity is a powerful and widely used game engine that supports Vulkan for high-performance rendering. By enabling Vulkan in Unity, you can take advantage of improved rendering performance, enhanced graphical fidelity, and better control over low-level GPU operations. This chapter provides a step-by-step guide to integrating Vulkan into Unity projects.

67.2 Why Use Vulkan in Unity?

Unity's default rendering backends, such as DirectX or OpenGL, are sufficient for many projects, but Vulkan offers several key advantages:

- **Improved Performance:** Vulkan's lower CPU overhead and multithreading capabilities allow for more efficient rendering.
- **Advanced Features:** Vulkan exposes modern GPU features that enable cutting-edge visual effects.
- **Cross-Platform Support:** Vulkan provides consistent performance across Windows, Linux, and Android.

67.3 Step 1: Check Unity Version and GPU Support

Before enabling Vulkan in Unity, ensure the following:

1. **Unity Version:** Use Unity 2018.1 or later, as Vulkan support was introduced in this version.
2. **GPU Compatibility:** Verify that your target platform and GPUs support Vulkan. Unity provides Vulkan support for Windows, Linux, and Android.

67.4 Step 2: Enable Vulkan in Unity

To use Vulkan in your Unity project:

1. Open your Unity project.
2. Go to **Edit > Project Settings > Player**.
3. In the **Other Settings** section, locate the **Graphics APIs** list.
4. Remove the default API (e.g., Direct3D or OpenGL) and add **Vulkan** from the drop-down menu.
5. Save the settings.

Unity will now use Vulkan for rendering in the editor and when building the project for supported platforms.

67.5 Step 3: Test Vulkan Rendering in the Unity Editor

After enabling Vulkan, test your project in the Unity Editor:

1. Run the project in **Play Mode**.
2. Observe the rendering quality and performance metrics. Unity provides a **Stats** window (accessible via the Game View) to monitor FPS and resource usage.
3. Look for graphical artifacts or performance issues. If problems arise, check the Unity console for Vulkan-specific warnings or errors.

67.6 Step 4: Build and Test on Target Platforms

Building your Vulkan-enabled project for deployment involves the same steps as any Unity build, but with Vulkan-specific considerations.

1. For Windows/Linux:

- Go to **File > Build Settings**.
- Select your target platform (**Windows** or **Linux**).
- Ensure the **Vulkan** Graphics API is enabled.
- Build and run the application on a Vulkan-compatible system.

2. For Android:

- In the Player Settings, ensure Vulkan is listed under Graphics APIs.
- Build the APK file and deploy it to a Vulkan-supported Android device.
- Use tools like **adb logcat** to monitor Vulkan-specific debug information.

67.7 Step 5: Debug and Optimize Vulkan in Unity

Unity includes several tools and settings for debugging and optimizing Vulkan applications:

1. **Use Vulkan Validation Layers:** Enable validation layers in the **Project Settings** to catch Vulkan API errors and inefficiencies.
2. **RenderDoc Integration:** Capture frames using **RenderDoc**, a popular tool for Vulkan debugging and performance analysis. Download RenderDoc and integrate it with Unity via the command line or plugin support.
3. **Optimize Shaders and Assets:** Vulkan excels with optimized shaders. Use Unity's Shader Graph or write custom HLSL shaders tailored for Vulkan.

67.8 Step 6: Leverage Vulkan-Specific Features

With Vulkan enabled, you can experiment with advanced rendering techniques in Unity:

- **Compute Shaders:** Vulkan's compute capabilities are ideal for particle systems, physics simulations, and image processing.
- **Tiled Rendering:** Optimize rendering on mobile devices by leveraging Vulkan's tile-based rendering.
- **Multi-GPU Support:** Vulkan enables explicit multi-GPU programming for high-end performance on compatible hardware.

67.9 Common Issues and Troubleshooting

1. **Editor Crashes or Freezes:** Ensure your GPU drivers are up-to-date. Vulkan requires the latest drivers for stable operation.
2. **Rendering Artifacts:** Check for unsupported shader features or incorrect use of Vulkan extensions.
3. **Performance Issues:** Use Unity's Profiler to identify bottlenecks and optimize rendering pipelines.

67.10 Summary

Integrating Vulkan into Unity projects unlocks powerful rendering capabilities and performance improvements. By following this chapter, you can enable Vulkan, test your projects on various platforms, and begin leveraging Vulkan-specific features for advanced graphics. With Vulkan in Unity, you can push the boundaries of what's possible in game and application development.

68

Integrate Vulkan with Unreal Engine: Use Vulkan in Unreal Engine

68.1 Introduction

Unreal Engine (UE) is a powerful game engine known for its cutting-edge graphics and robust tools. By integrating Vulkan, you can achieve higher rendering performance, access advanced GPU features, and improve cross-platform compatibility, especially for mobile and Linux development. This chapter explains how to enable and use Vulkan in Unreal Engine, ensuring optimal performance and visual fidelity.

68.2 Why Use Vulkan with Unreal Engine?

Using Vulkan in Unreal Engine offers several advantages:

- **Enhanced Performance:** Vulkan's low-overhead API and multithreading capabilities improve CPU-GPU communication.
- **Modern GPU Features:** Vulkan provides access to advanced rendering techniques like compute shaders and ray tracing.
- **Cross-Platform Flexibility:** Vulkan is ideal for targeting platforms like Android, Linux, and cloud-based solutions.

68.3 Step 1: Verify System Requirements

Before enabling Vulkan in Unreal Engine, ensure the following:

1. **Unreal Engine Version:** Vulkan support is available starting with Unreal Engine 4.16. Ensure you are using UE 4.16 or later.
2. **GPU and Driver Compatibility:** Confirm that your GPU supports Vulkan and install the latest drivers from the manufacturer's website.

68.4 Step 2: Enable Vulkan in Unreal Engine

To enable Vulkan rendering in your Unreal Engine project, follow these steps:

1. **Open Project Settings:**

- Navigate to **Edit > Project Settings**.

2. **Enable Vulkan:**

- Under the **Platforms** section, select the target platform (e.g., **Windows**, **Linux**, or **Android**).
- Locate the **Graphics API** settings and check **Vulkan**.
- For Android, disable **OpenGL ES** if Vulkan is the primary rendering API.

3. **Adjust Rendering Settings:**

- Navigate to **Engine > Rendering**.
- Ensure features such as **Ray Tracing**, **Compute Shaders**, or **MSAA** are configured appropriately for Vulkan.

4. **Save and Restart:**

- Unreal Engine may prompt you to restart the editor to apply Vulkan settings.

68.5 Step 3: Test Vulkan Rendering in the Editor

Once Vulkan is enabled:

1. Launch your project in the Unreal Engine editor.
2. Use the **Play** mode to observe Vulkan rendering performance and check for any graphical issues.
3. Open the **Output Log** (accessible via **Window > Developer Tools**) to view Vulkan-related debug information.

68.6 Step 4: Build and Test on Target Platforms

Unreal Engine simplifies building and deploying Vulkan applications. Follow these steps for common platforms:

1. **For Windows/Linux:**

- Go to **File > Package Project** and select the target platform.
- Ensure Vulkan is enabled in the packaging settings.
- Build the project and run it on a Vulkan-compatible system.

2. **For Android:**

- Install Android Studio and configure the Android SDK and NDK for Vulkan development.
- In **Project Settings > Platforms > Android**, enable Vulkan under **Graphics API**.
- Package the APK and deploy it to a Vulkan-supported Android device.

68.7 Step 5: Debug and Optimize Vulkan in Unreal Engine

1. **Use the Unreal Profiler:**

- Launch the **Session Frontend** tool (**Window > Developer Tools > Session Frontend**).
- Monitor GPU and CPU usage to identify Vulkan-related bottlenecks.

2. **Enable Validation Layers:**

- Add Vulkan validation layers to detect API misuse. Modify your **DefaultEngine.ini** file:

```
1  [/Script/VulkanRHI.VulkanSettings]
2  bUseValidationLayers=True
3
```

3. **Capture Frames with RenderDoc:**

- Unreal Engine supports integration with **RenderDoc**, a popular GPU debugging tool. Use it to analyze Vulkan frames and optimize shaders or pipelines.

4. **Optimize Material Complexity:**

- Open the **Shader Complexity View** in Unreal Engine to visualize and reduce expensive material operations.

68.8 Step 6: Experiment with Vulkan-Specific Features

Leverage Vulkan's advanced capabilities within Unreal Engine:

- **Compute Shaders:** Unreal Engine supports custom compute shaders for tasks like particle systems and procedural generation.
- **Ray Tracing:** Enable ray tracing in **Project Settings** for high-fidelity reflections, shadows, and global illumination.
- **Multiview Rendering:** Optimize rendering for VR and AR applications using Vulkan's multiview extensions.

68.9 Common Issues and Troubleshooting

1. **Editor Crashes:** Update your GPU drivers and ensure your system meets Vulkan's hardware requirements.
2. **Rendering Artifacts:** Check for unsupported materials or post-processing effects.
3. **Performance Drops:** Use Unreal Engine's profiling tools to identify and resolve Vulkan-related bottlenecks.

68.10 Summary

Integrating Vulkan with Unreal Engine allows developers to harness the power of this modern API while benefiting from Unreal's robust features and workflows. This chapter provided a step-by-step guide to enabling Vulkan, testing your projects, and optimizing performance. By using Vulkan in Unreal Engine, you can achieve better rendering efficiency and create visually stunning applications for various platforms.

Part VIII

Fun and Interactive Projects

69

Create a Vulkan Game Engine: Start with a simple engine core

69.1 Introduction

Creating your own game engine is a rewarding challenge that allows for complete control over rendering, asset management, and gameplay systems. Using Vulkan as the core graphics API provides access to high-performance rendering, efficient GPU usage, and modern graphics features. This chapter guides you through the initial steps of building a simple Vulkan-powered game engine core, focusing on modularity, scalability, and future extensibility.

69.2 Why Build a Vulkan Game Engine?

While existing game engines like Unity and Unreal Engine are powerful, building your own engine offers several unique benefits:

- **Complete Control:** Tailor the engine to meet your specific project needs, free from pre-built constraints.
- **Performance Optimization:** Utilize Vulkan's low-level access to the GPU for fine-grained optimization.
- **Learning Opportunity:** Gain a deep understanding of rendering pipelines, resource management, and game engine architecture.

69.3 Step 1: Design the Engine Structure

A well-designed game engine is modular and scalable. Start with a simple structure that can evolve as features are added. Divide your engine into the following core modules:

1. **Core**: Manages the engine lifecycle and system initialization.
2. **Graphics**: Handles Vulkan setup, rendering, and pipeline management.
3. **Input**: Processes user input from devices like keyboards, mice, or controllers.
4. **Assets**: Manages textures, shaders, and other resources.
5. **Game Logic**: Manages game-specific behavior and logic.

69.4 Step 2: Initialize the Engine Core

The engine core serves as the foundation of your Vulkan game engine. Create an initialization routine that sets up the necessary systems and provides a main loop for the engine.

69.4.1 Engine Class Structure

Define a base class for your engine:

```
1 class VulkanEngine {
2 public:
3     void Init();           // Initialize the engine
4     void MainLoop();       // Run the main loop
5     void Cleanup();        // Clean up resources
6
7 private:
8     bool isRunning;        // Flag to control the main loop
9 };
```

69.4.2 Initialization Routine

In the `Init()` method, set up Vulkan, load assets, and prepare the engine systems:

```
1 void VulkanEngine::Init() {
2     isRunning = true;
3
4     // Initialize Vulkan instance and device
5     VulkanRenderer.Init();
6
7     // Load assets
8     AssetManager.LoadAssets();
9
10    // Set up game logic
11    GameManager.Init();
12 }
```

69.4.3 Main Loop and Cleanup

The main loop drives the engine, updating game logic and rendering frames:

```

1 void VulkanEngine::MainLoop() {
2     while (isRunning) {
3         InputManager.ProcessInput();
4         GameManager.Update();
5         VulkanRenderer.RenderFrame();
6     }
7 }
8 void VulkanEngine::Cleanup() {
9     VulkanRenderer.Cleanup();
10    AssetManager.Cleanup();
11 }

```

69.5 Step 3: Set Up Vulkan for Rendering

The graphics module is responsible for Vulkan initialization and rendering. Start by encapsulating Vulkan-specific functionality in a separate class.

69.5.1 Graphics Module Class Structure

Define a class to manage Vulkan objects:

```

1 class VulkanRenderer {
2 public:
3     void Init();
4     void RenderFrame();
5     void Cleanup();
6
7 private:
8     VkInstance instance;
9     VkDevice device;
10    VkSwapchainKHR swapchain;
11    // Add other Vulkan objects as needed
12 };

```

69.5.2 Vulkan Initialization

In the `Init()` method, perform Vulkan setup:

1. Create a Vulkan instance.
2. Select a physical device.
3. Create a logical device and queues.
4. Set up a swapchain for rendering.

```

1 void VulkanRenderer::Init() {
2     CreateInstance();
3     SelectPhysicalDevice();
4     CreateLogicalDevice();
5     SetupSwapchain();
6 }

```


69.5.3 Rendering Frames

In the `RenderFrame()` method, record Vulkan commands to draw a frame:

```
1 void VulkanRenderer::RenderFrame() {
2     BeginCommandBuffer();
3     RenderObjects();
4     EndCommandBuffer();
5     SubmitToQueue();
6     PresentFrame();
7 }
```

69.6 Step 4: Modular Input Handling

Implement an input system to process events like key presses and mouse movement. Use a platform-specific library like GLFW or SDL2 for cross-platform input support.

```
1 class InputManager {
2 public:
3     static void ProcessInput();
4 };
5 void InputManager::ProcessInput() {
6     // Example: Poll events with GLFW
7     glfwPollEvents();
8 }
```

69.7 Step 5: Build a Simple Demo Scene

Create a basic scene to test the engine. Render a triangle or quad using Vulkan:

1. Load shaders for the pipeline.
2. Use vertex and index buffers to define the geometry.
3. Render the object in the `RenderFrame()` method.

69.8 Step 6: Plan for Future Features

Your engine should be designed with extensibility in mind. Future additions might include:

- Advanced rendering features like PBR, shadow mapping, or ray tracing.
- A physics engine for collisions and dynamics.
- Scripting support for game logic.
- Networking for multiplayer functionality.

69.9 Summary

Building a Vulkan-powered game engine is a complex but highly rewarding process. By starting with a simple, modular structure, you can lay the foundation for a scalable and feature-rich engine. This chapter introduced the basic steps for creating an engine core, setting up Vulkan rendering, and preparing for future growth. As you progress, you can expand this core to include advanced features and create your own complete game engine.

DRAFT
version
0.681

70

Build a Vulkan Game (e.g., Pong): Create a basic 2D game

70.1 Introduction

Creating a simple game like Pong is an excellent way to solidify your understanding of Vulkan and game development concepts. This project combines Vulkan's rendering capabilities with basic game logic to create a playable 2D game. This chapter walks you through building a minimalist Pong game with Vulkan, covering rendering, input handling, and game logic.

70.2 Game Design Overview

Pong is a straightforward 2D game with two paddles, a bouncing ball, and a scoring system. The game features:

- **Two paddles:** Controlled by the player(s).
- **A ball:** Moves across the screen and bounces off walls and paddles.
- **Scoring system:** Tracks points when the ball passes a paddle.

The core components of the game include:

1. **Graphics:** Rendering the paddles, ball, and background using Vulkan.
2. **Input Handling:** Capturing player controls for paddle movement.
3. **Game Logic:** Managing ball physics, collisions, and scoring.

70.3 Step 1: Initialize Vulkan and Set Up the Game Loop

Start by creating a basic Vulkan setup and game loop, as outlined in the previous chapter on creating a Vulkan game engine.

70.3.1 Engine Initialization

Initialize Vulkan and create a simple swapchain to render frames. Use a 2D orthographic projection for rendering since Pong is a 2D game.

In the `VulkanRenderer` class:

```
1 void VulkanRenderer::Init() {
2     CreateInstance();
3     SelectPhysicalDevice();
4     CreateLogicalDevice();
5     SetupSwapchain();
6     SetupRenderPass();
7     SetupFramebuffers();
8     CreatePipeline(); // Use a basic pipeline for 2D rendering
9 }
```

70.3.2 Game Loop Structure

The main game loop should handle input, update game logic, and render frames:

```
1 while (isRunning) {
2     InputManager.ProcessInput();
3     GameManager.Update();
4     VulkanRenderer.RenderFrame();
5 }
```

70.4 Step 2: Render Game Elements

Create simple graphics for the paddles, ball, and background.

70.4.1 Vertex Data for 2D Shapes

Define vertices for the paddles and ball as rectangles:

```
1 struct Vertex {
2     glm::vec2 position; // 2D position
3     glm::vec3 color;    // RGB color
4 };
5
6 std::vector<Vertex> paddleVertices = {
7     {{-0.05f, -0.3f}, {1.0f, 1.0f, 1.0f}},
8     {{ 0.05f, -0.3f}, {1.0f, 1.0f, 1.0f}},
9     {{ 0.05f, 0.3f}, {1.0f, 1.0f, 1.0f}},
10    {{-0.05f, 0.3f}, {1.0f, 1.0f, 1.0f}}
```

```
11 };
```

Use Vulkan buffers to store vertex data and bind them during rendering.

70.4.2 Rendering the Scene

In the `RenderFrame()` function, draw the paddles and ball:

```
1 void VulkanRenderer::RenderFrame() {
2     BeginCommandBuffer();
3     BindPipeline();
4     DrawObject(paddleVertices, paddleTransform);
5     DrawObject(ballVertices, ballTransform);
6     EndCommandBuffer();
7     SubmitToQueue();
8 }
```

70.5 Step 3: Implement Input Handling

Use a library like GLFW or SDL2 to capture player input. Map keys to paddle movement:

```
1 void InputManager::ProcessInput() {
2     if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
3         leftPaddlePosition.y += paddleSpeed * deltaTime;
4     }
5     if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
6         leftPaddlePosition.y -= paddleSpeed * deltaTime;
7     }
8     if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
9         rightPaddlePosition.y += paddleSpeed * deltaTime;
10    }
11    if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS) {
12        rightPaddlePosition.y -= paddleSpeed * deltaTime;
13    }
14 }
```

70.6 Step 4: Add Game Logic

Implement the ball's movement, collisions, and scoring in the `GameManager` class.

70.6.1 Ball Movement and Collision

Update the ball's position each frame and check for collisions:

```
1 void GameManager::Update() {
2     // Move the ball
3     ballPosition += ballVelocity * deltaTime;
4
5     // Bounce off walls
6     if (ballPosition.y > 1.0f || ballPosition.y < -1.0f) {
```



```

7     ballVelocity.y *= -1;
8 }
9
10 // Bounce off paddles
11 if (CheckCollision(ball, leftPaddle) || CheckCollision(ball, rightPaddle)) {
12     ballVelocity.x *= -1;
13 }
14
15 // Scoring
16 if (ballPosition.x > 1.0f) {
17     leftPlayerScore++;
18     ResetBall();
19 } else if (ballPosition.x < -1.0f) {
20     rightPlayerScore++;
21     ResetBall();
22 }
23 }

```

70.6.2 Collision Detection

Implement a simple axis-aligned bounding box (AABB) collision check:

```

1 bool CheckCollision(const Object& a, const Object& b) {
2     return (a.position.x < b.position.x + b.size.x &&
3             a.position.x + a.size.x > b.position.x &&
4             a.position.y < b.position.y + b.size.y &&
5             a.position.y + a.size.y > b.position.y);
6 }

```

70.7 Step 5: Display the Score

Render the score using a simple text-rendering library or custom shaders for bitmap fonts. Position the score at the top of the screen.

70.8 Step 6: Test and Optimize

Run the game and fine-tune:

1. Adjust paddle speed and ball velocity for enjoyable gameplay.
2. Test collision accuracy and make necessary adjustments.
3. Profile and optimize rendering to ensure smooth performance.

70.9 Summary

This chapter walked you through building a simple 2D game, Pong, using Vulkan. By combining Vulkan rendering with input handling and game logic, you've created a functional game that demonstrates key concepts in both Vulkan and game development. This foundational project can be extended with additional features like AI opponents, power-ups, or networking for multiplayer gameplay.

71

Render a 3D Model: Load and render 3D models (e.g., OBJ or GLTF)

71.1 Introduction

Rendering 3D models is a critical capability in modern graphics programming, and Vulkan provides the flexibility and power to load and render complex 3D assets. In this chapter, we will focus on loading a 3D model file (such as OBJ or GLTF), processing its data, and rendering it in a Vulkan-based application.

By the end of this chapter, you will understand how to:

1. Parse 3D model data from files.
2. Create Vulkan buffers for vertex and index data.
3. Render the model using Vulkan's graphics pipeline.

71.2 Step 1: Understand the 3D Model Format

To render a 3D model, you need to read its data from a file. Common formats include:

- **OBJ:** A simple text-based format for vertices, texture coordinates, and faces.
- **GLTF:** A modern format for 3D models, supporting binary data and rich features like animations.

For simplicity, this chapter will focus on rendering a static OBJ file. Libraries like `tinyobjloader` or `assimp` can simplify the parsing process.

71.3 Step 2: Load the Model Data

Use a library such as `tinyobjloader` to load the model's vertices and indices. Here is an example of loading a model using `tinyobjloader`:

```

1  // Include tinyobjloader
2  #include "tiny_obj_loader.h"
3
4  std::vector<Vertex> vertices;
5  std::vector<uint32_t> indices;
6
7  void LoadModel(const std::string& filePath) {
8      tinyobj::attrib_t attrib;
9      std::vector<tinyobj::shape_t> shapes;
10     std::vector<tinyobj::material_t> materials;
11     std::string warn, err;
12
13     // Load OBJ file
14     if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err, filePath.c_str())) {
15         throw std::runtime_error(warn + err);
16     }
17
18     // Parse vertices and indices
19     for (const auto& shape : shapes) {
20         for (const auto& index : shape.mesh.indices) {
21             Vertex vertex = {};
22             vertex.position = {
23                 attrib.vertices[3 * index.vertex_index + 0],
24                 attrib.vertices[3 * index.vertex_index + 1],
25                 attrib.vertices[3 * index.vertex_index + 2]
26             };
27             vertex.normal = {
28                 attrib.normals[3 * index.normal_index + 0],
29                 attrib.normals[3 * index.normal_index + 1],
30                 attrib.normals[3 * index.normal_index + 2]
31             };
32             vertices.push_back(vertex);
33             indices.push_back(index.vertex_index);
34         }
35     }
36 }

```

71.4 Step 3: Create Vulkan Buffers

Once the model data is loaded, create Vulkan buffers for the vertex and index data.

71.4.1 Vertex Buffer

Create a Vulkan buffer to hold the vertex data:

```

1  VkBuffer vertexBuffer;
2  VkDeviceMemory vertexBufferMemory;
3
4  CreateBuffer(vertices.size() * sizeof(Vertex),
5              VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
6              VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,

```

```

7     vertexBuffer, vertexBufferMemory);
8 CopyDataToBuffer(vertices.data(), vertexBufferMemory, vertices.size() * sizeof(Vertex));

```

71.4.2 Index Buffer

Similarly, create a buffer for the index data:

```

1 VkBuffer indexBuffer;
2 VkDeviceMemory indexBufferMemory;
3
4 CreateBuffer(indices.size() * sizeof(uint32_t),
5             VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
6             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
7             indexBuffer, indexBufferMemory);
8 CopyDataToBuffer(indices.data(), indexBufferMemory, indices.size() * sizeof(uint32_t));

```

71.5 Step 4: Update the Pipeline for 3D Rendering

Modify the graphics pipeline to support 3D rendering. Use a vertex shader to transform the model's vertices into world space.

71.5.1 Vertex Shader (GLSL)

Write a vertex shader that applies a model-view-projection (MVP) matrix:

```

1 #version 450
2
3 layout(location = 0) in vec3 inPosition;
4 layout(location = 1) in vec3 inNormal;
5
6 layout(set = 0, binding = 0) uniform UniformBufferObject {
7     mat4 model;
8     mat4 view;
9     mat4 proj;
10 } ubo;
11
12 layout(location = 0) out vec3 fragNormal;
13
14 void main() {
15     gl_Position = ubo.proj * ubo.view * ubo.model * vec4(inPosition, 1.0);
16     fragNormal = inNormal;
17 }

```

71.5.2 Fragment Shader (GLSL)

Write a fragment shader for basic lighting:

```

1 #version 450
2
3 layout(location = 0) in vec3 fragNormal;
4 layout(location = 0) out vec4 outColor;
5
6 void main() {

```

```

7   vec3 normal = normalize(fragNormal);
8   float light = max(dot(normal, vec3(0.0, 0.0, 1.0)), 0.0);
9   outColor = vec4(vec3(light), 1.0);
10 }

```

71.6 Step 5: Render the Model

Bind the vertex and index buffers, and draw the model:

```

1  vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
2  vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT32);
3  vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);

```

71.7 Step 6: Add Transformations and Camera Controls

Use uniform buffers to pass transformation matrices (model, view, and projection) to the vertex shader. Update the matrices each frame to allow camera movement and object transformations.

71.8 Step 7: Test and Extend

Load a simple OBJ model, such as a cube or sphere, to test your implementation. Once it works:

- Add support for materials and textures.
- Use the GLTF format for richer features like animations.

71.9 Summary

In this chapter, you learned how to load and render a 3D model using Vulkan. This process involves parsing model data, creating Vulkan buffers, and updating the graphics pipeline. Rendering 3D models is a foundational skill for developing 3D applications, and this chapter provides the groundwork for further exploration into advanced rendering techniques.

72

Add Skeletal Animation: Render animated 3D characters

72.1 Introduction

Skeletal animation is a technique used to animate 3D characters by manipulating an underlying skeleton or rig. Each vertex of the 3D model is influenced by one or more bones, and transformations applied to the skeleton dynamically deform the model. Vulkan provides the tools necessary to implement skeletal animation efficiently on the GPU.

In this chapter, you will:

- Understand the basics of skeletal animation and bone hierarchies.
- Use uniform buffers or storage buffers to pass bone transformation data.
- Implement skinning in the vertex shader.
- Render an animated 3D character in Vulkan.

72.2 Step 1: Understand the Skeleton and Skinning Process

Skeletal animation consists of:

1. **Bones and Hierarchies:** A skeleton is a hierarchy of bones, each with a parent-child relationship. Transformations applied to a parent bone affect all its child bones.
2. **Skinning:** Each vertex of the 3D model is associated with one or more bones. The influence of each bone is defined by weights, and the vertex position is updated based on these influences.

For each frame, the final transformation of each bone is calculated by combining its local transformation with that of its parent, creating a global transformation. These transformations are passed to the vertex shader for vertex deformation.

72.3 Step 2: Load a Rigged Model

A rigged 3D model includes:

- **Bone Data:** Each bone's offset matrix (initial position relative to the mesh) and parent relationships.
- **Vertex Weights:** The indices of bones affecting each vertex and the corresponding weight values.

The GLTF format is ideal for this purpose, as it supports skeletal animation natively. Libraries like `tinygltf` or `assimp` can load rigged models.

72.4 Step 3: Pass Bone Transformations to the GPU

Create a buffer to store the bone transformations for the current frame. You can use either a uniform buffer (if the number of bones is small) or a storage buffer (for more flexibility).

72.4.1 Uniform Buffer for Bone Data

Define a structure for the bone transformations:

```
1 struct BoneTransforms {
2     glm::mat4 boneMatrices[MAX_BONES];
3 };
```

Create and update the uniform buffer each frame:

```
1 // Create the buffer
2 VkBuffer boneBuffer;
3 VkDeviceMemory boneBufferMemory;
4 CreateBuffer(sizeof(BoneTransforms),
5             VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT,
6             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
7             boneBuffer, boneBufferMemory);
8
9 // Update the buffer with current bone transformations
10 void UpdateBoneBuffer(const std::vector<glm::mat4>& boneMatrices) {
11     void* data;
12     vkMapMemory(device, boneBufferMemory, 0, sizeof(BoneTransforms), 0, &data);
13     memcpy(data, boneMatrices.data(), sizeof(glm::mat4) * boneMatrices.size());
14     vkUnmapMemory(device, boneBufferMemory);
15 }
```

72.5 Step 4: Update the Vertex Shader for Skinning

In the vertex shader, use the bone transformation matrices to deform the vertices based on their weights and bone indices.

72.5.1 Vertex Shader (GLSL)

```

1  #version 450
2
3  layout(location = 0) in vec3 inPosition;
4  layout(location = 1) in vec3 inNormal;
5  layout(location = 2) in ivec4 inBoneIDs; // Bone indices
6  layout(location = 3) in vec4 inWeights; // Bone weights
7
8  layout(set = 0, binding = 0) uniform UniformBufferObject {
9      mat4 model;
10     mat4 view;
11     mat4 proj;
12 } ubo;
13
14 layout(set = 0, binding = 1) uniform BoneTransforms {
15     mat4 boneMatrices[MAX_BONES];
16 } bones;
17
18 layout(location = 0) out vec3 fragNormal;
19
20 void main() {
21     mat4 skinMatrix =
22         inWeights.x * bones.boneMatrices[inBoneIDs.x] +
23         inWeights.y * bones.boneMatrices[inBoneIDs.y] +
24         inWeights.z * bones.boneMatrices[inBoneIDs.z] +
25         inWeights.w * bones.boneMatrices[inBoneIDs.w];
26
27     vec4 worldPosition = ubo.model * skinMatrix * vec4(inPosition, 1.0);
28     gl_Position = ubo.proj * ubo.view * worldPosition;
29
30     mat3 normalMatrix = mat3(ubo.model) * mat3(skinMatrix);
31     fragNormal = normalize(normalMatrix * inNormal);
32 }

```

72.6 Step 5: Update the Rendering Code

During each frame, calculate the final bone transformations on the CPU or GPU. For CPU-side calculations:

- Traverse the bone hierarchy.
- Combine local transformations with parent transformations.
- Multiply by the bone's offset matrix.

Then, update the uniform buffer with the resulting matrices.

```

1  void CalculateBoneTransforms(const Bone& bone,
2                             const glm::mat4& parentTransform,
3                             std::vector<glm::mat4&& finalTransforms) {
4      glm::mat4 globalTransform = parentTransform * bone.localTransform;
5

```

```

6   finalTransforms[bone.index] = globalTransform * bone.offsetMatrix;
7
8   for (const Bone& child : bone.children) {
9       CalculateBoneTransforms(child, globalTransform, finalTransforms);
10  }
11 }

```

72.7 Step 6: Render the Animated Model

Bind the vertex buffer, index buffer, and descriptor sets for the uniform and bone buffers. Draw the rigged model as usual:

```

1  vkCmdBindDescriptorSets(commandBuffer,
2                          VK_PIPELINE_BIND_POINT_GRAPHICS,
3                          pipelineLayout,
4                          0, 1, &descriptorSet,
5                          0, nullptr);
6  vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
7  vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT32);
8  vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);

```

72.8 Step 7: Test and Extend

Start by animating a simple rigged model (e.g., a character with a walking animation). Once successful:

- Optimize the calculations by offloading bone transformations to a compute shader.
- Add interpolation for smooth animations.
- Support multiple animation clips and blending between them.

72.9 Summary

Skeletal animation is a foundational technique for rendering animated 3D characters in Vulkan. By leveraging Vulkan's buffer systems and shader capabilities, you can efficiently implement skinning and render dynamic, lifelike animations. This chapter serves as a gateway to more advanced topics such as GPU-based animation, blending, and procedural animation techniques.

73

Simulate a Water Surface: Use compute shaders for water ripples

73.1 Introduction

Simulating realistic water surfaces is a captivating application of computer graphics and compute shaders. Water ripples can be modeled as a 2D heightfield that evolves over time based on physical principles. Vulkan's compute shaders provide the parallel processing power required to efficiently simulate and render such effects.

In this chapter, you will:

- Learn the basics of water ripple simulation using heightfields.
- Use compute shaders to update the heightfield based on a wave propagation algorithm.
- Render the simulated water surface using Vulkan's graphics pipeline.

73.2 Step 1: Understand the Wave Equation for Water Ripples

The behavior of water ripples can be modeled using the wave equation, which describes how wave heights propagate over time:

$$H(x, y, t + 1) = 2 \cdot H(x, y, t) - H(x, y, t - 1) + c \cdot \Delta^2 H(x, y, t)$$

Here:

- $H(x, y, t)$ is the height of the water at position (x, y) and time t .
- c is a constant that controls wave speed.

- $\Delta^2 H(x, y, t)$ is the Laplacian of the heightfield, representing the curvature of the water surface.

This equation can be discretized and implemented on a 2D grid, where each cell represents a water height value.

73.3 Step 2: Create Buffers for the Heightfield

Allocate two buffers to store the current and previous heightfields. These will be updated alternately to simulate the passage of time.

```

1 // Heightfield resolution
2 const int gridSize = 512;
3
4 // Create buffers for current and previous heightfield states
5 VkBuffer heightfieldBuffers[2];
6 VkDeviceMemory heightfieldMemories[2];
7
8 for (int i = 0; i < 2; ++i) {
9     CreateBuffer(gridSize * gridSize * sizeof(float),
10                 VK_BUFFER_USAGE_STORAGE_BUFFER_BIT,
11                 VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
12                 heightfieldBuffers[i],
13                 heightfieldMemories[i]);
14 }
```

73.4 Step 3: Implement the Compute Shader for Wave Simulation

The compute shader updates the heightfield by calculating the Laplacian for each cell and applying the wave equation.

73.4.1 Compute Shader (GLSL)

```

1 #version 450
2
3 layout(local_size_x = 16, local_size_y = 16) in;
4
5 // Storage buffers for heightfield states
6 layout(set = 0, binding = 0) buffer HeightfieldCurrent {
7     float currentHeight[];
8 };
9
10 layout(set = 0, binding = 1) buffer HeightfieldPrevious {
11     float previousHeight[];
12 };
13
14 layout(set = 0, binding = 2) buffer HeightfieldNext {
15     float nextHeight[];
16 };
17
18 uniform int gridSize;
19 uniform float waveSpeed;
20
21 void main() {
```

```

22 ivec2 coord = ivec2(gl_GlobalInvocationID.xy);
23
24 if (coord.x <= 0 || coord.y <= 0 ||
25     coord.x >= gridSize - 1 || coord.y >= gridSize - 1) {
26     return; // Skip boundaries
27 }
28
29 int index = coord.y * gridSize + coord.x;
30
31 // Calculate Laplacian (average difference)
32 float laplacian =
33     currentHeight[index - 1] +
34     currentHeight[index + 1] +
35     currentHeight[index - gridSize] +
36     currentHeight[index + gridSize] -
37     4.0 * currentHeight[index];
38
39 // Apply wave equation
40 nextHeight[index] = 2.0 * currentHeight[index] - previousHeight[index] + waveSpeed * laplacian;
41 }

```

73.5 Step 4: Dispatch the Compute Shader

Set up a descriptor set to bind the heightfield buffers and dispatch the compute shader to update the water surface.

```

1 // Bind buffers to descriptor set
2 VkDescriptorSet descriptorSet = CreateHeightfieldDescriptorSet(heightfieldBuffers);
3
4 // Record compute shader dispatch
5 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
6 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
7     pipelineLayout, 0, 1, &descriptorSet, 0, nullptr);
8
9 // Dispatch compute shader
10 vkCmdDispatch(commandBuffer, gridSize / 16, gridSize / 16, 1);

```

After dispatching, swap the current and previous buffers for the next frame.

73.6 Step 5: Render the Water Surface

Use a vertex and fragment shader to visualize the heightfield as a water surface. Each vertex position is offset by the height value, and the normals are calculated based on the heightfield to produce realistic lighting effects.

73.6.1 Vertex Shader (GLSL)

```

1 #version 450
2
3 layout(location = 0) in vec2 inPosition; // Grid coordinates
4
5 layout(location = 0) out vec3 fragNormal;
6
7 layout(set = 0, binding = 0) buffer HeightfieldCurrent {
8     float currentHeight[];

```



```

9  };
10
11  uniform int gridSize;
12
13  void main() {
14      int index = int(inPosition.y * gridSize + inPosition.x);
15
16      vec3 pos = vec3(inPosition, currentHeight[index]);
17
18      // Calculate normals using heightfield gradients
19      float dx = currentHeight[index + 1] - currentHeight[index - 1];
20      float dy = currentHeight[index + gridSize] - currentHeight[index - gridSize];
21      fragNormal = normalize(vec3(-dx, -dy, 2.0));
22
23      gl_Position = vec4(pos, 1.0);
24  }

```

73.6.2 Fragment Shader (GLSL)

The fragment shader applies a simple water-like appearance using lighting calculations.

```

1  #version 450
2
3  layout(location = 0) in vec3 fragNormal;
4  layout(location = 0) out vec4 outColor;
5
6  void main() {
7      vec3 lightDir = normalize(vec3(0.5, 0.5, -1.0));
8      float diffuse = max(dot(fragNormal, lightDir), 0.0);
9
10     outColor = vec4(0.0, 0.3, 0.8, 1.0) * diffuse; // Blue water
11 }

```

73.7 Step 6: Add Interactivity

For an engaging simulation, add interaction by injecting disturbances (e.g., ripples) into the heightfield. Update the height of specific grid points based on user input.

```

1  // Example: Add a ripple at a specific grid point
2  void AddRipple(int x, int y, float intensity) {
3      int index = y * gridSize + x;
4      currentHeight[index] += intensity;
5  }

```

73.8 Summary

This chapter introduced the simulation and rendering of a dynamic water surface using Vulkan. By leveraging compute shaders, you efficiently implemented wave propagation, visualized the results, and added interactivity. This technique forms the basis for more advanced fluid simulations and visual effects in real-time applications.

74

Render a Procedural Planet: Generate planets using Vulkan

74.1 Introduction

Rendering procedural planets allows developers to create dynamic and visually captivating celestial objects for space games, simulations, or educational tools. By using Vulkan, you can generate realistic planets with customizable features like terrain, atmosphere, and textures in real time. This chapter will guide you through generating and rendering a procedural planet using Vulkan's compute shaders and graphics pipeline.

In this chapter, you will:

- Create a spherical mesh for the planet.
- Use procedural noise to generate terrain.
- Apply shading and atmospheric effects to enhance realism.

74.2 Step 1: Generate a Spherical Mesh

A planet is represented as a sphere composed of vertices and triangles. Start by defining a spherical mesh, such as an icosphere or a subdivided UV sphere.

74.2.1 UV Sphere Mesh Generation

Create a sphere by mapping a rectangular grid of latitude and longitude coordinates onto a 3D surface. Each vertex is computed using spherical coordinates:

$$x = \sin(\phi) \cdot \cos(\theta)$$

$$y = \sin(\phi) \cdot \sin(\theta)$$

$$z = \cos(\phi)$$

Here, ϕ is the latitude, and θ is the longitude.

```

1 // Generate a UV sphere
2 std::vector<Vertex> generateSphere(float radius, int latitudeSegments, int longitudeSegments) {
3     std::vector<Vertex> vertices;
4     for (int lat = 0; lat <= latitudeSegments; ++lat) {
5         float phi = M_PI * float(lat) / float(latitudeSegments);
6         for (int lon = 0; lon <= longitudeSegments; ++lon) {
7             float theta = 2.0f * M_PI * float(lon) / float(longitudeSegments);
8
9             Vertex vertex;
10            vertex.position = glm::vec3(
11                radius * sin(phi) * cos(theta),
12                radius * sin(phi) * sin(theta),
13                radius * cos(phi)
14            );
15            vertex.normal = glm::normalize(vertex.position);
16            vertices.push_back(vertex);
17        }
18    }
19    return vertices;
20 }

```

74.3 Step 2: Add Procedural Terrain

Procedural terrain generation uses algorithms such as Perlin noise or Simplex noise to add height variations to the sphere's surface. Compute shaders are ideal for parallel generation of these height values.

74.3.1 Compute Shader for Height Displacement

Implement a compute shader that takes vertex positions and applies procedural noise to modify their heights.

```

1 #version 450
2
3 layout(local_size_x = 64) in;
4
5 layout(set = 0, binding = 0) buffer VertexBuffer {
6     vec3 positions[];
7 };
8
9 uniform float time; // Time for animated noise
10 uniform float frequency; // Noise frequency
11 uniform float amplitude; // Noise amplitude
12
13 // Simplex noise function
14 float simplexNoise(vec3 v);
15

```

```

16 void main() {
17     uint id = gl_GlobalInvocationID.x;
18
19     // Apply noise-based height displacement
20     vec3 position = positions[id];
21     float noise = simplexNoise(position * frequency + vec3(time));
22     positions[id] += normalize(position) * noise * amplitude;
23 }

```

74.3.2 Dispatch the Compute Shader

Dispatch the compute shader to update the sphere's vertices.

```

1 // Dispatch compute shader for terrain generation
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, terrainComputePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE,
4                         computePipelineLayout, 0, 1, &descriptorSet, 0, nullptr);
5
6 vkCmdDispatch(commandBuffer, vertexCount / 64, 1, 1);

```

74.4 Step 3: Render the Planet

After generating the terrain, render the sphere using Vulkan's graphics pipeline. Use a vertex shader to transform the vertices and a fragment shader to add shading effects.

74.4.1 Vertex Shader for Spherical Transformations

```

1 #version 450
2
3 layout(location = 0) in vec3 inPosition;
4 layout(location = 1) in vec3 inNormal;
5
6 layout(location = 0) out vec3 fragNormal;
7
8 layout(set = 0, binding = 0) uniform UniformBufferObject {
9     mat4 model;
10    mat4 view;
11    mat4 proj;
12 };
13
14 void main() {
15     fragNormal = mat3(model) * inNormal;
16     gl_Position = proj * view * model * vec4(inPosition, 1.0);
17 }

```

74.4.2 Fragment Shader for Shading and Atmosphere

Simulate planetary lighting with diffuse and specular highlights. Add a simple atmospheric effect using a gradient based on the view angle.

```

1 #version 450
2
3 layout(location = 0) in vec3 fragNormal;

```

```

4 layout(location = 0) out vec4 outColor;
5
6 uniform vec3 lightDir;
7 uniform vec3 planetColor;
8 uniform vec3 atmosphereColor;
9
10 void main() {
11     vec3 normal = normalize(fragNormal);
12     float diffuse = max(dot(normal, -lightDir), 0.0);
13     float specular = pow(max(dot(normal, reflect(lightDir, normal)), 0.0), 16.0);
14
15     // Atmospheric gradient
16     float atmosphereFactor = pow(1.0 - dot(normal, vec3(0.0, 1.0, 0.0)), 2.0);
17
18     vec3 color = mix(atmosphereColor, planetColor, atmosphereFactor) * diffuse + vec3(0.2) * specular;
19     outColor = vec4(color, 1.0);
20 }

```

74.5 Step 4: Add Rotation and Animation

Introduce rotation to the planet for a dynamic effect. Update the model matrix over time.

```

1 // Update rotation in the UBO
2 ubo.model = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));

```

74.6 Step 5: Combine with Stars or a Skybox

Enhance the scene by adding a starry background or a skybox. Use a cubemap to create a realistic space environment.

74.7 Conclusion

This chapter introduced the steps to generate and render a procedural planet with Vulkan. Leveraging compute shaders and procedural noise, you created dynamic terrain, applied realistic shading, and animated the planet. This approach serves as a foundation for building more complex planetary systems in real-time graphics applications.

75

Add an Interactive GUI: Use a library like Dear ImGui

75.1 Introduction

Interactive graphical user interfaces (GUIs) are essential for many applications, especially tools and games, where users need to adjust settings, monitor statistics, or interact with the program. In this chapter, you will learn how to integrate **Dear ImGui**, a popular GUI library, into your Vulkan project. By combining Vulkan's high-performance rendering with Dear ImGui's simplicity, you can create flexible, interactive GUIs.

This chapter covers:

- Setting up Dear ImGui with Vulkan.
- Rendering an ImGui interface.
- Adding interactive elements like sliders, buttons, and checkboxes.

75.2 Step 1: Integrate Dear ImGui

Dear ImGui provides a Vulkan backend that simplifies integration. First, download the Dear ImGui library and include it in your project.

75.2.1 Add ImGui Files

Clone the Dear ImGui repository from GitHub and copy the necessary files into your project:

- `imgui.cpp`
- `imgui.h`
- `imgui_draw.cpp`
- `imgui_widgets.cpp`
- `imgui_impl_vulkan.cpp`
- `imgui_impl_glfw.cpp` (if you are using GLFW for windowing)

Include the Dear ImGui headers in your project:

```
1 #include "imgui.h"
2 #include "imgui_impl_vulkan.h"
3 #include "imgui_impl_glfw.h"
```

75.3 Step 2: Initialize ImGui with Vulkan

Dear ImGui requires Vulkan-specific initialization, including a descriptor pool and a rendering pipeline.

75.3.1 Create a Descriptor Pool for ImGui

Dear ImGui uses Vulkan descriptors to manage resources. Create a descriptor pool tailored to ImGui's needs:

```
1 // Create descriptor pool for ImGui
2 VkDescriptorPoolSize poolSizes[] = {
3     { VK_DESCRIPTOR_TYPE_SAMPLER, 1000 },
4     { VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1000 },
5     { VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 1000 },
6     { VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 1000 },
7     { VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, 1000 },
8     { VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, 1000 },
9     { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1000 },
10    { VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, 1000 },
11    { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, 1000 },
12    { VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, 1000 },
13    { VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, 1000 }
14 };
15
16 VkDescriptorPoolCreateInfo poolInfo = {0};
17 poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
18 poolInfo.flags = VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT;
19 poolInfo.maxSets = 1000 * IM_ARRAYSIZE(poolSizes);
20 poolInfo.poolSizeCount = static_cast<uint32_t>(IM_ARRAYSIZE(poolSizes));
21 poolInfo.pPoolSizes = poolSizes;
22
23 VkDescriptorPool imguiPool;
24 vkCreateDescriptorPool(device, &poolInfo, nullptr, &imguiPool);
```

75.3.2 Initialize Dear ImGui Context

Set up the Dear ImGui context and configure its Vulkan backend:

```

1 // Create Dear ImGui context
2 ImGui_CHECKVERSION();
3 ImGui::CreateContext();
4 ImGuiIO& io = ImGui::GetIO();
5 io.ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard; // Enable keyboard navigation
6 io.ConfigFlags |= ImGuiConfigFlags_NavEnableGamepad; // Enable gamepad navigation
7
8 // Initialize ImGui for Vulkan
9 ImGui_ImplGlfw_InitForVulkan(window, true);
10 ImGui_ImplVulkan_InitInfo initInfo = {};
11 initInfo.Instance = instance;
12 initInfo.PhysicalDevice = physicalDevice;
13 initInfo.Device = device;
14 initInfo.QueueFamily = graphicsQueueFamily;
15 initInfo.Queue = graphicsQueue;
16 initInfo.DescriptorPool = imguiPool;
17 initInfo.MinImageCount = swapchainImageCount;
18 initInfo.ImageCount = swapchainImageCount;
19
20 ImGui_ImplVulkan_Init(&initInfo, renderPass);

```

75.4 Step 3: Render ImGui in Vulkan

Dear ImGui needs to record its rendering commands into Vulkan command buffers.

75.4.1 Update Command Buffers

In your Vulkan rendering loop, update the command buffer to include ImGui's draw data:

```

1 // Start a new ImGui frame
2 ImGui_ImplVulkan_NewFrame();
3 ImGui_ImplGlfw_NewFrame();
4 ImGui::NewFrame();
5
6 // Add ImGui elements
7 ImGui::Begin("Example Window");
8 ImGui::Text("Hello, Vulkan!");
9 ImGui::SliderFloat("Slider", &value, 0.0f, 1.0f);
10 ImGui::Checkbox("Checkbox", &checkboxValue);
11 ImGui::End();
12
13 // Render ImGui
14 ImGui::Render();
15 ImDrawData* drawData = ImGui::GetDrawData();
16 ImGui_ImplVulkan_RenderDrawData(drawData, commandBuffer);

```

75.4.2 Ensure Synchronization

Since ImGui's rendering is part of the main Vulkan rendering flow, ensure synchronization with semaphores and fences.

75.5 Step 4: Customize the GUI

Dear ImGui allows you to add a wide variety of interactive elements. For example:

- **Buttons:** Trigger actions.
- **Sliders:** Adjust parameters like brightness or speed.
- **Graphs:** Visualize real-time data.

```
1 // Example GUI elements
2 if (ImGui::Button("Click Me")) {
3     // Perform an action
4 }
5 ImGui::SliderFloat("Light Intensity", &lightIntensity, 0.0f, 10.0f);
6 ImGui::PlotLines("Frame Times", frameTimes, IM_ARRAYSIZE(frameTimes));
```

75.6 Step 5: Cleanup

When shutting down your Vulkan application, clean up ImGui resources:

```
1 // Cleanup ImGui
2 ImGui_ImplVulkan_Shutdown();
3 ImGui_ImplGlfw_Shutdown();
4 ImGui::DestroyContext();
5 vkDestroyDescriptorPool(device, imguiPool, nullptr);
```

75.7 Summary

This chapter demonstrated how to integrate Dear ImGui with Vulkan to create interactive GUIs. With ImGui, you can quickly add tools and visualizations to your Vulkan applications, making them more user-friendly and engaging. The flexibility of ImGui, combined with Vulkan's power, allows for efficient and dynamic user interfaces that enhance the development and debugging experience.

76

Simulate Fireworks: Create a dynamic fireworks display

76.1 Introduction

Simulating fireworks is an exciting way to explore the use of particle systems. Leveraging Vulkan's compute and graphics capabilities, we can create a visually stunning and highly dynamic display. This chapter will guide you through the steps to simulate fireworks, including initializing a particle system, updating particles with compute shaders, and rendering them with Vulkan.

This chapter covers:

- Initializing a particle system to simulate fireworks.
- Using compute shaders to handle particle physics.
- Rendering particles with Vulkan's graphics pipeline.

76.2 Step 1: Design the Particle System

A particle system is a collection of particles, each with attributes like position, velocity, color, and lifespan. For fireworks:

- Particles are initialized at the launch point of the firework.
- Each particle is given a velocity and a lifespan.
- A compute shader updates the position and color of the particles over time.
- Dead particles are recycled to create new explosions.

76.3 Step 2: Create a Compute Shader for Particle Updates

The compute shader handles the physics and updates for each particle. For example:

- Update positions using velocity and time.
- Reduce lifespan.
- Apply acceleration due to gravity.

Here's an example compute shader in GLSL:

```

1  #version 450
2  layout(local_size_x = 256) in;
3
4  struct Particle {
5      vec4 position; // xyz: position, w: lifetime
6      vec4 velocity; // xyz: velocity, w: unused
7      vec4 color;    // rgba: particle color
8  };
9
10 layout(std430, binding = 0) buffer Particles {
11     Particle particles[];
12 };
13
14 layout(push_constant) uniform PushConstants {
15     float deltaTime;
16     vec3 gravity;
17 } pc;
18
19 void main() {
20     uint idx = gl_GlobalInvocationID.x;
21     Particle p = particles[idx];
22
23     // Update only if the particle is alive
24     if (p.position.w > 0.0) {
25         p.velocity.xyz += pc.gravity * pc.deltaTime;
26         p.position.xyz += p.velocity.xyz * pc.deltaTime;
27         p.position.w -= pc.deltaTime; // Decrease lifetime
28     } else {
29         // Respawn dead particle
30         p.position = vec4(0.0, 0.0, 0.0, 5.0); // New position, reset lifetime
31         p.velocity = vec4(vec3(rand(), rand(), rand()) - 0.5) * 10.0, 0.0);
32         p.color = vec4(rand(), rand(), rand(), 1.0);
33     }
34
35     particles[idx] = p;
36 }

```

76.4 Step 3: Initialize Particles in Vulkan

Create a buffer to store particle data:

```

1  // Particle structure
2  struct Particle {
3      glm::vec4 position; // xyz: position, w: lifetime
4      glm::vec4 velocity; // xyz: velocity, w: unused
5      glm::vec4 color;    // rgba: color
6  };
7

```

```

8 // Create a buffer for particles
9 const size_t particleCount = 10000;
10 VkBuffer particleBuffer;
11 VkDeviceMemory particleMemory;
12
13 // Allocate and bind the buffer
14 createBuffer(device, particleCount * sizeof(Particle),
15             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
16             VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
17             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT |
18             VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
19             particleBuffer, particleMemory);

```

76.5 Step 4: Set Up the Graphics Pipeline

To render particles, use a simple vertex and fragment shader:

- **Vertex Shader:** Transform each particle's position and pass its color to the fragment shader.
- **Fragment Shader:** Use the passed color to render the particle.

Example vertex shader:

```

1 #version 450
2 layout(location = 0) in vec4 inPosition; // xyz: position, w: lifetime
3 layout(location = 1) in vec4 inColor;    // rgba: color
4
5 layout(location = 0) out vec4 fragColor;
6
7 layout(push_constant) uniform PushConstants {
8     mat4 viewProjection;
9 } pc;
10
11 void main() {
12     fragColor = inColor;
13     gl_Position = pc.viewProjection * vec4(inPosition.xyz, 1.0);
14 }

```

Example fragment shader:

```

1 #version 450
2 layout(location = 0) in vec4 fragColor;
3 layout(location = 0) out vec4 outColor;
4
5 void main() {
6     outColor = fragColor;
7 }

```

76.6 Step 5: Simulate Fireworks

Integrate the compute and graphics pipelines:

1. Use the compute pipeline to update particles.
2. Transition the particle buffer to the appropriate usage.

3. Render particles using the graphics pipeline.

```

1 // Record command buffer
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipelineLayout,
4   0, 1, &computeDescriptorSet, 0, nullptr);
5
6 // Dispatch compute shader
7 vkCmdDispatch(commandBuffer, (particleCount + 255) / 256, 1, 1);
8
9 // Transition buffer for graphics
10 VkBufferMemoryBarrier barrier = {};
11 barrier.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
12 barrier.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT;
13 barrier.dstAccessMask = VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT;
14 barrier.buffer = particleBuffer;
15 barrier.size = VK_WHOLE_SIZE;
16
17 vkCmdPipelineBarrier(commandBuffer,
18   VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
19   VK_PIPELINE_STAGE_VERTEX_INPUT_BIT,
20   0, 0, nullptr, 1, &barrier, 0, nullptr);
21
22 // Bind graphics pipeline and draw particles
23 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
24 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &particleBuffer, offsets);
25 vkCmdDraw(commandBuffer, particleCount, 1, 0, 0);

```

76.7 Summary

Simulating fireworks with Vulkan combines the power of compute shaders for physics simulation and the graphics pipeline for rendering. This chapter introduced you to dynamic particle systems, enabling you to create visually rich and interactive applications. With the foundational skills developed here, you can expand the system to include additional effects like trails, smoke, or advanced lighting for more realism.

77

Render a Terrain from Heightmaps: Load heightmap data for terrain rendering

77.1 Introduction

Rendering terrains is an essential technique for many 3D applications, such as games and simulations. This chapter explores how to render a terrain using heightmap data, which encodes elevation information as grayscale values in an image. By converting this data into vertex positions, we can construct a mesh that represents the terrain surface.

This chapter will cover:

- Loading a heightmap texture.
- Generating a terrain mesh based on heightmap data.
- Rendering the terrain using Vulkan's graphics pipeline.

77.2 Step 1: Understanding Heightmaps

A heightmap is a 2D image where the intensity of each pixel represents the elevation of the terrain at that point. For example:

- A white pixel represents the highest point.
- A black pixel represents the lowest point.

By sampling the heightmap, we can generate vertices with varying heights and form a grid-based mesh to represent the terrain.

77.3 Step 2: Load the Heightmap

The heightmap is usually stored as an image file (e.g., PNG, BMP). Use a library like `stb_image` to load the image and access the grayscale data.

```

1  #include <stb_image.h>
2
3  // Load heightmap
4  int width, height, channels;
5  unsigned char* heightmapData = stbi_load("heightmap.png", &width, &height, &channels, STBI_grey);
6
7  if (!heightmapData) {
8      throw std::runtime_error("Failed to load heightmap image");
9  }
10
11 // Ensure dimensions are valid
12 if (width <= 0 || height <= 0) {
13     throw std::runtime_error("Invalid heightmap dimensions");
14 }

```

The grayscale values are now stored in `heightmapData`, ranging from 0 (black) to 255 (white).

77.4 Step 3: Generate Terrain Mesh

To generate the terrain mesh:

1. Create a grid of vertices.
2. Use the heightmap data to set the **y-coordinate** (height) of each vertex.
3. Define indices for rendering the grid as a triangle list.

```

1  // Vertex structure
2  struct Vertex {
3      glm::vec3 position; // x, y, z
4      glm::vec3 normal;   // Normal vector
5      glm::vec2 texCoord; // Texture coordinates
6  };
7
8  // Generate terrain vertices
9  std::vector<Vertex> vertices;
10 std::vector<uint32_t> indices;
11
12 for (int z = 0; z < height; ++z) {
13     for (int x = 0; x < width; ++x) {
14         float heightValue = heightmapData[z * width + x] / 255.0f;
15         vertices.push_back({
16             glm::vec3(x, heightValue * maxHeight, z),
17             glm::vec3(0.0f, 1.0f, 0.0f), // Placeholder normal
18             glm::vec2(x / float(width), z / float(height))
19         });
20     }
21 }
22
23 // Generate indices for triangle list
24 for (int z = 0; z < height - 1; ++z) {
25     for (int x = 0; x < width - 1; ++x) {
26         uint32_t topLeft = z * width + x;
27         uint32_t topRight = topLeft + 1;

```

```

28     uint32_t bottomLeft = (z + 1) * width + x;
29     uint32_t bottomRight = bottomLeft + 1;
30
31     // Two triangles per quad
32     indices.push_back(topLeft);
33     indices.push_back(bottomLeft);
34     indices.push_back(topRight);
35
36     indices.push_back(topRight);
37     indices.push_back(bottomLeft);
38     indices.push_back(bottomRight);
39 }
40 }

```

77.5 Step 4: Create Buffers for the Terrain

The vertex and index data need to be uploaded to Vulkan buffers.

```

1  // Create vertex buffer
2  VkBuffer vertexBuffer;
3  VkDeviceMemory vertexBufferMemory;
4  createBuffer(vertices.size() * sizeof(Vertex),
5              VK_BUFFER_USAGE_VERTEX_BUFFER_BIT,
6              VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
7              vertexBuffer, vertexBufferMemory);
8
9  // Create index buffer
10 VkBuffer indexBuffer;
11 VkDeviceMemory indexBufferMemory;
12 createBuffer(indices.size() * sizeof(uint32_t),
13             VK_BUFFER_USAGE_INDEX_BUFFER_BIT,
14             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
15             indexBuffer, indexBufferMemory);
16
17 // Copy data to buffers
18 copyToBuffer(vertexBuffer, vertices.data(), vertices.size() * sizeof(Vertex));
19 copyToBuffer(indexBuffer, indices.data(), indices.size() * sizeof(uint32_t));

```

77.6 Step 5: Render the Terrain

Set up a basic graphics pipeline for rendering the terrain. The vertex shader transforms the grid positions, and the fragment shader shades the terrain using simple lighting or textures.

Example vertex shader:

```

1  #version 450
2  layout(location = 0) in vec3 inPosition;
3  layout(location = 1) in vec3 inNormal;
4  layout(location = 2) in vec2 inTexCoord;
5
6  layout(location = 0) out vec2 fragTexCoord;
7
8  layout(push_constant) uniform PushConstants {
9      mat4 model;
10     mat4 viewProjection;
11 } pc;
12
13 void main() {

```

```

14     fragTexCoord = inTexCoord;
15     gl_Position = pc.viewProjection * pc.model * vec4(inPosition, 1.0);
16 }

```

Render the terrain with indexed drawing:

```

1  vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
2  vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
3  vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT32);
4  vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 1, 0, 0, 0);

```

77.7 Step 6: Improve Terrain Rendering

Enhance the appearance of the terrain with:

- Normals for realistic lighting.
- A texture map blended with the heightmap.
- Tessellation shaders for smooth surfaces.

77.8 Summary

This chapter demonstrated how to load and use heightmaps for terrain rendering. By combining a heightmap with Vulkan's powerful graphics pipeline, you can create realistic terrains for your 3D scenes. The skills learned here lay the groundwork for advanced techniques like tessellation or procedural terrain generation.

78

Add VR Support: Use Vulkan for VR rendering

78.1 Introduction

Virtual Reality (VR) has transformed the way we interact with 3D environments, offering immersive experiences with stereoscopic rendering and head tracking. Vulkan's low-level API provides the performance and flexibility needed for efficient VR rendering. In this chapter, we will explore how to integrate VR support into your Vulkan application.

Key objectives include:

- Understanding VR rendering requirements.
- Setting up a VR runtime like OpenXR.
- Implementing stereoscopic rendering.
- Synchronizing rendering with head tracking.

78.2 Step 1: Understanding VR Rendering

VR rendering differs from traditional rendering in several ways:

- **Stereoscopic Rendering:** Two images (one for each eye) must be rendered from slightly different perspectives.
- **High Performance:** VR demands high frame rates (90+ FPS) to avoid motion sickness.
- **Head Tracking:** The view must update dynamically based on the user's head movements.

78.3 Step 2: Set Up a VR Runtime

A VR runtime like OpenXR provides the abstraction layer to interface with VR hardware. OpenXR is an open standard that supports multiple VR devices, making it a flexible choice.

First, install and set up the OpenXR SDK. This involves:

1. Including the OpenXR headers and libraries in your project.
2. Linking against the OpenXR loader (`openxr_loader.lib` or equivalent).

Initialize OpenXR and check for a Vulkan-compatible runtime:

```

1  #include <openxr/openxr.h>
2  #include <openxr/openxr_platform.h>
3
4  // Create OpenXR instance
5  XrInstance xrInstance;
6  XrInstanceCreateInfo instanceCreateInfo = { XR_TYPE_INSTANCE_CREATE_INFO };
7  strcpy(instanceCreateInfo.applicationInfo.applicationName, "Vulkan VR App");
8  instanceCreateInfo.applicationInfo.applicationVersion = 1;
9  instanceCreateInfo.applicationInfo.apiVersion = XR_CURRENT_API_VERSION;
10
11 if (xrCreateInstance(&instanceCreateInfo, &xrInstance) != XR_SUCCESS) {
12     throw std::runtime_error("Failed to create OpenXR instance");
13 }
14
15 // Check for Vulkan support
16 XrSystemId xrSystemId;
17 XrSystemGetInfo systemGetInfo = { XR_TYPE_SYSTEM_GET_INFO };
18 systemGetInfo.formFactor = XR_FORM_FACTOR_HEAD_MOUNTED_DISPLAY;
19
20 if (xrGetSystem(xrInstance, &systemGetInfo, &xrSystemId) != XR_SUCCESS) {
21     throw std::runtime_error("Failed to get VR system");
22 }

```

78.4 Step 3: Create Vulkan Instance and Surface for VR

Use the OpenXR extension `XR_KHR_vulkan_enable` to interface Vulkan with the VR runtime. Query the necessary extensions and create a Vulkan instance that is compatible with OpenXR.

```

1  // Query required Vulkan extensions
2  uint32_t extensionCount;
3  xrEnumerateInstanceExtensionProperties(nullptr, 0, &extensionCount, nullptr);
4
5  std::vector<XrExtensionProperties> extensions(extensionCount, { XR_TYPE_EXTENSION_PROPERTIES });
6  xrEnumerateInstanceExtensionProperties(nullptr, extensionCount, &extensionCount, extensions.data());
7
8  // Add required extensions to Vulkan
9  std::vector<const char*> requiredExtensions;
10 for (const auto& extension : extensions) {
11     requiredExtensions.push_back(extension.extensionName);
12 }
13
14 // Create Vulkan instance with required extensions
15 VkInstanceCreateInfo createInfo = {};
16 // Populate createInfo...
17 vkCreateInstance(&createInfo, nullptr, &vulkanInstance);

```

Bind the Vulkan instance to OpenXR:

```
1  XrGraphicsBindingVulkanKHR graphicsBinding = { XR_TYPE_GRAPHICS_BINDING_VULKAN_KHR };
2  graphicsBinding.instance = vulkanInstance;
3  graphicsBinding.physicalDevice = physicalDevice;
4  graphicsBinding.device = logicalDevice;
5  graphicsBinding.queueFamilyIndex = graphicsQueueFamilyIndex;
6  graphicsBinding.queueIndex = 0;
```

78.5 Step 4: Set Up Stereoscopic Rendering

For stereoscopic rendering, the VR runtime will provide two separate framebuffers (one for each eye). You must render each eye's view from its respective perspective.

Set up two Vulkan swapchains, one for each eye:

```
1  // Query VR swapchain formats
2  uint32_t swapchainFormatCount;
3  xrEnumerateSwapchainFormats(xrSession, 0, &swapchainFormatCount, nullptr);
4
5  std::vector<int64_t> swapchainFormats(swapchainFormatCount);
6  xrEnumerateSwapchainFormats(xrSession, swapchainFormatCount, &swapchainFormatCount, swapchainFormats.data());
7
8  // Create swapchains for each eye
9  XrSwapchainCreateInfo swapchainCreateInfo = { XR_TYPE_SWAPCHAIN_CREATE_INFO };
10 swapchainCreateInfo.format = swapchainFormats[0];
11 swapchainCreateInfo.width = eyeWidth;
12 swapchainCreateInfo.height = eyeHeight;
13 swapchainCreateInfo.arraySize = 1;
14 swapchainCreateInfo.mipCount = 1;
15 swapchainCreateInfo.faceCount = 1;
16 swapchainCreateInfo.sampleCount = 1;
17
18 XrSwapchain leftEyeSwapchain, rightEyeSwapchain;
19 xrCreateSwapchain(xrSession, &swapchainCreateInfo, &leftEyeSwapchain);
20 xrCreateSwapchain(xrSession, &swapchainCreateInfo, &rightEyeSwapchain);
```

78.6 Step 5: Synchronize with Head Tracking

The VR runtime provides pose data for each eye, which includes the position and orientation of the headset. Use this data to update the view matrices for rendering.

```
1  // Query head pose for current frame
2  XrViewLocateInfo viewLocateInfo = { XR_TYPE_VIEW_LOCATE_INFO };
3  viewLocateInfo.viewConfigurationType = XR_VIEW_CONFIGURATION_TYPE_PRIMARY_STEREO;
4  viewLocateInfo.displayTime = frameState.predictedDisplayTime;
5  viewLocateInfo.space = xrSpace;
6
7  XrViewState viewState = { XR_TYPE_VIEW_STATE };
8  std::vector<XrView> views(2, { XR_TYPE_VIEW });
9  xrLocateViews(xrSession, &viewLocateInfo, &viewState, views.size(), &viewCount, views.data());
10
11 // Update view matrices for each eye
12 glm::mat4 leftEyeViewMatrix = convertXrPoseToMat4(views[0].pose);
13 glm::mat4 rightEyeViewMatrix = convertXrPoseToMat4(views[1].pose);
```

78.7 Step 6: Render and Present to the VR Display

Render the scene for each eye using the corresponding view and projection matrices. Submit the rendered images to the VR runtime for presentation.

```
1 // Bind left eye framebuffer
2 bindFramebuffer(leftEyeFramebuffer);
3 renderScene(leftEyeViewMatrix, leftEyeProjectionMatrix);
4
5 // Bind right eye framebuffer
6 bindFramebuffer(rightEyeFramebuffer);
7 renderScene(rightEyeViewMatrix, rightEyeProjectionMatrix);
8
9 // Submit images to VR runtime
10 XrSwapchainImageSubmitInfo submitInfo = { XR_TYPE_SWAPCHAIN_IMAGE_SUBMIT_INFO };
11 // Populate submitInfo...
12 xrEndFrame(xrSession, &frameEndInfo);
```

78.8 Summary

This chapter introduced the steps to integrate VR rendering with Vulkan using OpenXR. Leveraging stereoscopic rendering, head tracking, and Vulkan's performance, you can create immersive VR experiences. Advanced topics such as foveated rendering, asynchronous time-warp, and hand tracking can further enhance your VR application.

79

Build an Interactive Solar System: Simulate planetary orbits

79.1 Introduction

In this chapter, we will create an interactive simulation of a solar system using Vulkan. This project involves rendering planets and moons with realistic orbital mechanics and adding interactivity to explore the system. It combines 3D rendering, animation, and user input to create an engaging experience.

Key objectives include:

- Rendering spheres for planets and moons.
- Simulating orbital mechanics for motion.
- Implementing hierarchical transformations for parent-child relationships.
- Adding user controls to explore the system interactively.

79.2 Rendering Spheres for Planets and Moons

Planets and moons are modeled as textured spheres. Use a sphere generation algorithm, such as subdividing an icosahedron or mapping a UV texture onto a unit sphere.

```
1 // Generate a sphere mesh
2 std::vector<Vertex> sphereVertices;
3 std::vector<uint32_t> sphereIndices;
4 generateSphereMesh(radius, sectorCount, stackCount, sphereVertices, sphereIndices);
5
6 // Create Vulkan buffers for the sphere
7 createVertexBuffer(sphereVertices);
```

```
8 createIndexBuffer(sphereIndices);
```

Apply textures to the spheres to represent planetary surfaces. For instance, you can use diffuse maps of planets like Earth, Mars, and Jupiter.

79.3 Simulating Orbital Mechanics

Use Kepler's laws of motion to simulate the orbits of celestial bodies. Each planet and moon has parameters like orbital radius, angular velocity, and inclination.

```
1 // Update orbital positions
2 float deltaTime = getDeltaTime();
3 planetPosition.x = cos(angularVelocity * currentTime) * orbitalRadius;
4 planetPosition.z = sin(angularVelocity * currentTime) * orbitalRadius;
```

Encapsulate the orbital properties in a structure:

```
1 struct OrbitalBody {
2     glm::vec3 position;
3     float orbitalRadius;
4     float angularVelocity;
5     glm::mat4 modelMatrix;
6 };
```

Update the positions of planets and moons dynamically during each frame.

79.4 Hierarchical Transformations

To handle moons orbiting planets, implement hierarchical transformations. Each moon's position is relative to its parent planet.

```
1 // Parent (planet) transformation
2 glm::mat4 planetModelMatrix = glm::translate(glm::mat4(1.0f), planetPosition);
3 planetModelMatrix = glm::rotate(planetModelMatrix, rotationAngle, glm::vec3(0, 1, 0));
4
5 // Child (moon) transformation
6 glm::mat4 moonModelMatrix = planetModelMatrix;
7 moonModelMatrix = glm::translate(moonModelMatrix, moonOffset);
8 moonModelMatrix = glm::rotate(moonModelMatrix, moonRotationAngle, glm::vec3(0, 1, 0));
```

Use a uniform buffer to pass these model matrices to the vertex shader for rendering.

79.5 Adding Interactivity

Enable camera controls to explore the solar system interactively. Use mouse and keyboard input to allow the user to rotate, zoom, and pan the camera.

```
1 // Process input for camera movement
2 if (isKeyPressed(KEY_W)) {
3     cameraPosition += cameraSpeed * cameraDirection;
```

```
4 }  
5 if (isKeyPressed(KEY_S)) {  
6     cameraPosition -= cameraSpeed * cameraDirection;  
7 }
```

Allow the user to select celestial bodies and display information such as orbital radius, mass, and rotation speed.

79.6 Rendering the Solar System

Render the sun at the center of the solar system and planets orbiting around it. Add moons orbiting their respective planets. Use a skybox or a starfield texture for the background.

```
1 // Render sun  
2 renderObject(sunModelMatrix, sunTexture);  
3  
4 // Render planets  
5 for (const auto& planet : planets) {  
6     renderObject(planet.modelMatrix, planet.texture);  
7 }  
8  
9 // Render moons  
10 for (const auto& moon : moons) {  
11     renderObject(moon.modelMatrix, moon.texture);  
12 }
```

Use push constants or descriptor sets to pass texture and transformation data to the shaders.

79.7 Summary

This chapter guided you through building an interactive solar system simulation with Vulkan. Combining 3D rendering, hierarchical transformations, and dynamic animation, you can create an engaging representation of planetary systems. As a next step, consider adding advanced features like realistic lighting, orbit prediction, or gravitational interactions for a more sophisticated simulation.

80

Create a Vulkan Screensaver: Use Vulkan for a graphical screensaver

80.1 Introduction

A screensaver is an excellent project to explore Vulkan's rendering capabilities in a creative and visually engaging way. This chapter will guide you through building a simple Vulkan-based screensaver that runs a continuously animated scene. This project emphasizes animation, procedural graphics, and efficient rendering techniques while operating without user interaction.

The screensaver we will create features:

- A procedurally generated scene (e.g., swirling particles or flowing shapes).
- Smooth animations using Vulkan's rendering pipeline.
- Idle mode operation, optimized for minimal CPU and GPU usage.

80.2 Setting Up the Vulkan Application

Like any Vulkan project, we start by initializing a Vulkan instance, creating a window, and setting up a swapchain for rendering. The setup process follows the patterns described in earlier chapters, with the addition of ensuring the application runs fullscreen to simulate screensaver behavior.

```
1 // Create Vulkan instance
2 createVulkanInstance();
3
4 // Setup fullscreen window
5 WindowConfig config;
```

```

6  config.fullscreen = true;
7  createWindow(config);
8
9  // Initialize swapchain
10 initializeSwapchain(window);

```

Ensure that your application captures the "idle" state of the system to mimic a screensaver's purpose.

80.3 Designing the Screensaver Scene

A screensaver often features procedural graphics or abstract animations. In this example, we will create swirling particles that respond to sinusoidal motion.

80.3.1 Particle System Generation

The particle system consists of a set of points, each moving based on a sinusoidal function. The positions and colors of the particles are calculated dynamically in a compute shader or vertex shader.

```

1  // Generate particle data
2  struct Particle {
3      glm::vec3 position;
4      glm::vec3 velocity;
5      glm::vec4 color;
6  };
7  std::vector<Particle> particles;
8  initializeParticles(particles);
9
10 // Create Vulkan buffer for particles
11 createBuffer(VK_BUFFER_USAGE_VERTEX_BUFFER_BIT, particles);

```

80.3.2 Animating Particles

The particle positions are updated on the GPU using a compute shader, which allows for efficient processing of large datasets.

```

1  // Compute shader: Update particle positions
2  layout(std430, binding = 0) buffer Particles {
3      vec4 positions[];
4  };
5  layout(push_constant) uniform SimulationParams {
6      float time;
7      float speed;
8  } params;
9
10 void main() {
11     uint idx = gl_GlobalInvocationID.x;
12     positions[idx].xyz += vec3(
13         sin(params.time + idx) * params.speed,
14         cos(params.time - idx) * params.speed,
15         0.0);
16 }

```

The updated positions are then passed to the graphics pipeline for rendering.

80.4 Rendering the Screensaver

Render the particles as points or small textured quads using the Vulkan graphics pipeline. The vertex shader processes each particle's position, and the fragment shader adds color and effects.

```

1  // Vertex shader: Transform particle positions
2  layout(location = 0) in vec3 inPosition;
3  layout(location = 1) in vec4 inColor;
4  layout(location = 0) out vec4 outColor;
5
6  uniform mat4 projection;
7  uniform mat4 view;
8
9  void main() {
10     gl_Position = projection * view * vec4(inPosition, 1.0);
11     outColor = inColor;
12 }
```

80.4.1 Post-Processing Effects

To enhance visual appeal, add post-processing effects such as bloom, motion blur, or color grading. These effects can be implemented using a separate render pass.

80.5 Idle Mode Optimization

Screensavers often run in the background without user input, so they should consume minimal system resources. Optimize performance by:

- Reducing the particle count or effect complexity when the system is under load.
- Limiting frame rate to conserve GPU power.

```

1  // Implement frame limiter
2  limitFrameRate(30); // Cap FPS to 30
```

80.6 Summary

This chapter demonstrated how to create a Vulkan-based screensaver with smooth animations and procedural graphics. Exploring efficient rendering techniques and GPU-based animations, you can create visually stunning effects suitable for idle-mode applications. Experiment with various effects and particle behaviors to design unique screensaver experiences. This project can also serve as a foundation for dynamic visualization tools or real-time art installations.

Part IX

Tools and Integration Projects

81

Integrate with OpenCV: Process Vulkan-rendered images with OpenCV

81.1 Introduction

Combining Vulkan's high-performance rendering with OpenCV's powerful image processing capabilities opens up exciting possibilities for real-time applications. In this chapter, we will render a scene with Vulkan, retrieve the rendered image, and process it using OpenCV. This integration can be useful for tasks like computer vision, post-processing, or saving rendered frames for later analysis.

Key steps include:

- Rendering a scene with Vulkan.
- Transferring the rendered image from Vulkan to CPU memory.
- Using OpenCV to process the image.

81.2 Capturing Vulkan-Rendered Images

To process rendered images with OpenCV, the rendered frame must be copied from the Vulkan framebuffer to a CPU-accessible format. We use a staging buffer to handle this transfer.

81.2.1 Creating the Staging Buffer

The staging buffer serves as an intermediate storage area for transferring data from the GPU to the CPU.

```

1 // Create a staging buffer
2 VkBuffer stagingBuffer;
3 VkDeviceMemory stagingMemory;
4 createBuffer(imageSize,
5             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
6             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
7             stagingBuffer,
8             stagingMemory);

```

81.2.2 Copying Image Data to the Buffer

After rendering, the image is copied from the Vulkan swapchain image to the staging buffer using a command buffer.

```

1 // Transition image layout to transfer source
2 transitionImageLayout(swapchainImage, VK_IMAGE_LAYOUT_PRESENT_SRC_KHR, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL);
3
4 // Copy image to staging buffer
5 VkCommandBuffer cmdBuffer = beginSingleTimeCommands();
6 VkBufferImageCopy copyRegion = {};
7 copyRegion.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
8 copyRegion.imageSubresource.layerCount = 1;
9 copyRegion.imageExtent.width = swapchainExtent.width;
10 copyRegion.imageExtent.height = swapchainExtent.height;
11 copyRegion.imageExtent.depth = 1;
12
13 vkCmdCopyImageToBuffer(cmdBuffer, swapchainImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, stagingBuffer, 1, &
14     ↳ copyRegion);
15 endSingleTimeCommands(cmdBuffer);

```

The image data is now available in the staging buffer and ready to be mapped to CPU memory.

81.2.3 Mapping the Buffer to CPU Memory

Map the staging buffer to retrieve the image data and convert it into a format compatible with OpenCV.

```

1 // Map the buffer and copy data
2 void* data;
3 vkMapMemory(device, stagingMemory, 0, imageSize, 0, &data);
4 std::vector<uint8_t> imageData(static_cast<uint8_t*>(data), static_cast<uint8_t*>(data) + imageSize);
5 vkUnmapMemory(device, stagingMemory);

```

81.3 Processing with OpenCV

OpenCV operates on images represented as `cv::Mat`. Convert the raw Vulkan image data into this format and apply various OpenCV operations.

81.3.1 Converting Vulkan Image to `cv::Mat`

Ensure that the image format (e.g., `VK_FORMAT_R8G8B8A8_UNORM`) matches OpenCV's expectations. If needed, rearrange channels or change formats.

```

1 // Create an OpenCV Mat from the image data
2 cv::Mat image(swapchainExtent.height, swapchainExtent.width, CV_8UC4, imageData.data());
3
4 // Convert RGBA to BGR (OpenCV uses BGR format by default)
5 cv::cvtColor(image, image, cv::COLOR_RGBA2BGR);

```

81.3.2 Applying OpenCV Processing

With the image in `cv::Mat` format, you can apply any OpenCV function, such as filtering, edge detection, or feature extraction.

```

1 // Apply a Gaussian blur
2 cv::Mat blurredImage;
3 cv::GaussianBlur(image, blurredImage, cv::Size(15, 15), 5.0);
4
5 // Detect edges using Canny
6 cv::Mat edges;
7 cv::Canny(blurredImage, edges, 50, 150);

```

81.3.3 Displaying or Saving the Processed Image

Display the processed image in an OpenCV window or save it to a file.

```

1 // Display the image
2 cv::imshow("Processed Vulkan Image", edges);
3 cv::waitKey(0);
4
5 // Save the image
6 cv::imwrite("processed_image.png", edges);

```

81.4 Integrating the Workflow

To streamline the integration, encapsulate Vulkan rendering and OpenCV processing in separate functions. This modular approach allows easy reuse in various applications.

```

1 void renderWithVulkan() {
2     // Vulkan rendering logic
3 }
4
5 void processWithOpenCV(const std::vector<uint8_t>& imageData) {
6     // OpenCV processing logic
7 }
8
9 // Main loop
10 while (!shouldExit) {
11     renderWithVulkan();
12     std::vector<uint8_t> imageData = captureImageFromVulkan();
13     processWithOpenCV(imageData);
14 }

```

81.5 Summary

Integrating Vulkan and OpenCV provides a powerful combination for real-time rendering and image processing. This chapter demonstrated how to render a scene with Vulkan, retrieve the rendered frame, and process it using OpenCV. With this approach, you can implement advanced computer vision workflows, create artistic post-processing effects, or analyze rendered images. Expand on this example by exploring other OpenCV features, such as machine learning or real-time tracking.

DRAFT
version
0.681

82

Export Frames to Images: Save Vulkan-rendered frames as PNG files

82.1 Introduction

Saving Vulkan-rendered frames as images is a valuable capability for debugging, generating content, or creating videos from rendered scenes. This chapter explains how to capture frames rendered with Vulkan, transfer them to CPU-accessible memory, and save them as PNG files using a library like `stb_image_write` or `libpng`.

Key steps include:

- Capturing rendered images from the Vulkan swapchain.
- Transferring image data from GPU to CPU memory.
- Saving the image in a standard format such as PNG.

82.2 Capturing Rendered Frames

The process of capturing a rendered frame starts by copying the swapchain image to a staging buffer, as described in the previous chapter. For frame exporting, ensure that the rendered images are in a color format compatible with standard image libraries.

82.2.1 Preparing a Staging Buffer

Create a staging buffer to hold the rendered image data before transferring it to the CPU.


```

1 // Create a staging buffer for image transfer
2 VkBuffer stagingBuffer;
3 VkDeviceMemory stagingMemory;
4 createBuffer(imageSize,
5             VK_BUFFER_USAGE_TRANSFER_DST_BIT,
6             VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
7             stagingBuffer,
8             stagingMemory);

```

82.2.2 Copying the Image Data

Transition the swapchain image layout to `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` and copy the image data to the staging buffer using a command buffer.

```

1 // Transition image layout for transfer
2 transitionImageLayout(swapchainImage, VK_IMAGE_LAYOUT_PRESENT_SRC_KHR, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL);
3
4 // Copy image to the staging buffer
5 VkCommandBuffer cmdBuffer = beginSingleTimeCommands();
6 VkBufferImageCopy copyRegion = {};
7 copyRegion.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
8 copyRegion.imageSubresource.layerCount = 1;
9 copyRegion.imageExtent.width = swapchainExtent.width;
10 copyRegion.imageExtent.height = swapchainExtent.height;
11 copyRegion.imageExtent.depth = 1;
12
13 vkCmdCopyImageToBuffer(cmdBuffer, swapchainImage, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, stagingBuffer, 1, &
14     ↪ copyRegion);
15 endSingleTimeCommands(cmdBuffer);

```

82.2.3 Mapping the Staging Buffer

Map the staging buffer to CPU memory to retrieve the image data.

```

1 // Map buffer memory and retrieve image data
2 void* data;
3 vkMapMemory(device, stagingMemory, 0, imageSize, 0, &data);
4 std::vector<uint8_t> imageData(static_cast<uint8_t*>(data), static_cast<uint8_t*>(data) + imageSize);
5 vkUnmapMemory(device, stagingMemory);

```

82.3 Saving Frames as PNG Images

To save the image data, use a library such as `stb_image_write`, which provides a simple interface for writing images in formats like PNG.

82.3.1 Integrating `stb_image_write`

Include the `stb_image_write.h` header in your project. If you don't already have this library, you can download it from <https://github.com/nothings/stb>.

```

1 // Include stb_image_write
2 #define STB_IMAGE_WRITE_IMPLEMENTATION
3 #include "stb_image_write.h"

```

82.3.2 Saving the Image

Convert the image data to the appropriate format if necessary (e.g., from RGBA to RGB) and save it using `stb_image_write`.

```

1 // Save image as PNG
2 int width = swapchainExtent.width;
3 int height = swapchainExtent.height;
4
5 // Convert RGBA to RGB if required
6 std::vector<uint8_t> rgbImageData(width * height * 3);
7 for (size_t i = 0; i < width * height; ++i) {
8     rgbImageData[i * 3 + 0] = imageData[i * 4 + 0]; // R
9     rgbImageData[i * 3 + 1] = imageData[i * 4 + 1]; // G
10    rgbImageData[i * 3 + 2] = imageData[i * 4 + 2]; // B
11 }
12
13 // Write PNG file
14 stbi_write_png("output_frame.png", width, height, 3, rgbImageData.data(), width * 3);

```

82.3.3 Alternative Libraries

You can also use libraries like `libpng` for more control over image saving. While `stb_image_write` is simpler, `libpng` offers advanced features such as metadata embedding and better compression.

82.4 Automating Frame Export

To save frames periodically during the application run, modify the rendering loop to trigger frame captures at desired intervals.

```

1 // Main rendering loop with frame export
2 int frameIndex = 0;
3 while (!shouldExit) {
4     renderFrame();
5
6     // Export frame every 60 frames (e.g., for videos)
7     if (frameIndex % 60 == 0) {
8         std::vector<uint8_t> imageData = captureFrame();
9         saveImage(imageData, "frame_" + std::to_string(frameIndex) + ".png");
10    }
11    ++frameIndex;
12 }

```

82.5 Summary

In this chapter, we demonstrated how to export Vulkan-rendered frames as PNG images using a staging buffer and a simple image-saving library. This capability is useful for debugging, creating static graphics, or generating video sequences. With the described techniques, you can efficiently capture and save rendered frames, enabling new possibilities for both visualization and content creation. Experiment further by exporting sequences or applying real-time filters to the saved frames.

83

Integrate Vulkan with TensorFlow: Use Vulkan for ML model inference

83.1 Introduction

Vulkan's compute capabilities make it a powerful backend for machine learning (ML) tasks, such as model inference. By integrating Vulkan with TensorFlow, you can leverage Vulkan's performance for running ML models on GPUs, particularly in scenarios where custom Vulkan-based workflows are preferred. This chapter demonstrates how to use Vulkan to perform inference tasks in TensorFlow.

83.2 Overview of Vulkan and TensorFlow Integration

TensorFlow is a highly flexible library for ML, and it supports various GPU backends like CUDA and OpenCL. Integrating Vulkan requires the use of TensorFlow's ability to interact with external compute backends. By writing custom TensorFlow operations (ops), you can create a bridge between TensorFlow's computation graph and Vulkan's compute pipelines.

The integration process involves:

- Setting up Vulkan for compute tasks.
- Defining a custom TensorFlow operation that uses Vulkan.
- Passing data between TensorFlow tensors and Vulkan buffers.

83.3 Setting Up Vulkan for TensorFlow

Before integrating with TensorFlow, ensure Vulkan is set up for compute tasks. This includes creating a Vulkan instance, selecting a physical device, and creating a compute pipeline. Refer to the earlier chapters on Vulkan compute pipelines if needed.

83.3.1 Configuring TensorFlow for External Operations

To use Vulkan, you need to implement a custom TensorFlow operation. TensorFlow supports this through its C++ API. Custom ops allow TensorFlow to delegate specific operations to external backends, like Vulkan.

83.4 Writing a Custom TensorFlow Operation

A custom TensorFlow operation consists of a kernel implementation and registration code. The kernel performs the operation using Vulkan, while the registration code integrates it into TensorFlow's computation graph.

83.4.1 Define the Custom Operation

Start by defining the operation in a TensorFlow C++ file.

```
1 // Define a custom TensorFlow op
2 #include "tensorflow/core/framework/op.h"
3 #include "tensorflow/core/framework/op_kernel.h"
4
5 REGISTER_OP("VulkanInference")
6   .Input("input: float")
7   .Output("output: float")
8   .SetShapeFn([](tensorflow::shape_inference::InferenceContext* c) {
9     c->set_output(0, c->input(0)); // Same shape as input
10    return tensorflow::Status::OK();
11  });
```

This operation accepts a tensor as input and produces a tensor as output. The shape inference function ensures the output tensor has the same shape as the input tensor.

83.4.2 Implement the Vulkan Kernel

Next, write the kernel for the operation. This is where the Vulkan code is used.

```
1 class VulkanInferenceOp : public tensorflow::OpKernel {
2 public:
3   explicit VulkanInferenceOp(tensorflow::OpKernelConstruction* context)
4     : OpKernel(context) {
5     // Initialize Vulkan resources (e.g., device, pipeline)
6     initializeVulkan();
7   }
8
9   ~VulkanInferenceOp() {
```

```

10     // Cleanup Vulkan resources
11     cleanupVulkan();
12 }
13
14 void Compute(tensorflow::OpKernelContext* context) override {
15     // Access input tensor
16     const tensorflow::Tensor& input = context->input(0);
17     auto inputTensor = input.flat<float>();
18
19     // Allocate output tensor
20     tensorflow::Tensor* output = nullptr;
21     OP_REQUIRES_OK(context, context->allocate_output(0, input.shape(), &output));
22     auto outputTensor = output->flat<float>();
23
24     // Transfer data to Vulkan buffer
25     uploadToVulkanBuffer(inputTensor.data(), inputTensor.size());
26
27     // Execute Vulkan compute pipeline
28     runVulkanPipeline();
29
30     // Retrieve data from Vulkan buffer
31     downloadFromVulkanBuffer(outputTensor.data(), outputTensor.size());
32 }
33
34 private:
35     void initializeVulkan();
36     void cleanupVulkan();
37     void uploadToVulkanBuffer(const float* data, size_t size);
38     void runVulkanPipeline();
39     void downloadFromVulkanBuffer(float* data, size_t size);
40 };

```

In the kernel, you:

1. Transfer input tensor data to a Vulkan buffer.
2. Execute the Vulkan compute pipeline for inference.
3. Transfer the computed results back to the output tensor.

83.4.3 Register the Operation Kernel

Register the operation kernel with TensorFlow.

```

1 REGISTER_KERNEL_BUILDER(
2     Name("VulkanInference").Device(tensorflow::DEVICE_CPU), VulkanInferenceOp);

```

This binds the VulkanInference operation to the VulkanInferenceOp kernel.

83.5 Integrating with a TensorFlow Model

To use the custom Vulkan operation in a TensorFlow model, you must build TensorFlow with your custom operation and then use it in a Python script.

83.5.1 Building TensorFlow with the Custom Operation

Rebuild TensorFlow from source, ensuring the custom operation is included. Follow TensorFlow's official guide for building custom operations.

83.5.2 Using the Operation in Python

Once the operation is built, you can call it in a TensorFlow graph.

```
1 import tensorflow as tf
2 import my_vulkan_op
3
4 # Create input tensor
5 input_data = tf.constant([1.0, 2.0, 3.0, 4.0], dtype=tf.float32)
6
7 # Call the custom Vulkan operation
8 output_data = my_vulkan_op.vulkan_inference(input_data)
9
10 # Evaluate the operation
11 with tf.Session() as sess:
12     result = sess.run(output_data)
13     print("Inference result:", result)
```

83.6 Summary

Integrating Vulkan with TensorFlow enables leveraging Vulkan's compute power for ML tasks. By writing a custom TensorFlow operation, you can pass data to Vulkan for processing and retrieve the results seamlessly. This chapter provided a foundation for creating Vulkan-accelerated ML workflows. With this knowledge, you can experiment further with advanced inference pipelines or hybrid rendering-ML systems.

84

Build a Vulkan Render Doc Clone: Debug Vulkan calls visually

84.1 Introduction

Debugging Vulkan applications can be challenging due to the complexity of the API and the vast number of operations involved in rendering or compute pipelines. Tools like RenderDoc have set the standard for Vulkan debugging by providing features such as frame capture, API call inspection, and shader debugging. In this chapter, we explore how to build a basic Vulkan debugging tool—a mini RenderDoc clone—that allows you to capture and inspect Vulkan calls and debug frames visually.

84.2 Overview of the Approach

Building a Vulkan debugging tool requires intercepting Vulkan API calls, recording GPU resources (e.g., images, buffers), and visualizing the captured data. This involves:

- Using Vulkan layers to intercept Vulkan calls.
- Recording and storing frame data, such as commands, descriptors, and shaders.
- Creating a user interface to view and inspect the captured frames.

84.3 Setting Up a Vulkan Debug Layer

Vulkan's layer mechanism allows you to intercept and modify Vulkan API calls. Layers operate between your application and the Vulkan driver, making them ideal for debugging.

84.3.1 Creating a Custom Vulkan Layer

Start by creating a Vulkan layer to intercept API calls. Vulkan layers are implemented as shared libraries loaded at runtime.

```

1  // Define the Vulkan layer interception
2  #include <vulkan/vulkan.h>
3
4  // Function pointers for the real Vulkan functions
5  static PFN_vkCreateInstance real_vkCreateInstance = nullptr;
6
7  // Intercept vkCreateInstance
8  VKAPI_ATTR VkResult VKAPI_CALL vkCreateInstance(
9      const VkInstanceCreateInfo* pCreateInfo,
10     const VkAllocationCallbacks* pAllocator,
11     VkInstance* pInstance) {
12     // Call the real vkCreateInstance function
13     VkResult result = real_vkCreateInstance(pCreateInfo, pAllocator, pInstance);
14
15     // Log or capture the instance creation parameters
16     printf("Intercepted vkCreateInstance\n");
17
18     return result;
19 }
20
21 // Initialize the layer
22 VKAPI_ATTR PFN_vkVoidFunction VKAPI_CALL vkGetInstanceProcAddr(
23     VkInstance instance,
24     const char* pName) {
25     if (strcmp(pName, "vkCreateInstance") == 0) {
26         if (!real_vkCreateInstance) {
27             real_vkCreateInstance = (PFN_vkCreateInstance)
28                 vkGetInstanceProcAddr(instance, "vkCreateInstance");
29         }
30         return (PFN_vkVoidFunction) vkCreateInstance;
31     }
32
33     // Call the real vkGetInstanceProcAddr
34     return vkGetInstanceProcAddr(instance, pName);
35 }

```

In this example, `vkCreateInstance` is intercepted to log instance creation. Similar hooks can be added for other Vulkan API calls.

84.3.2 Loading the Vulkan Layer

Vulkan layers are loaded via environment variables or application-defined settings. Add your layer to a layer manifest file (`layer.json`) and specify it using the `VK_LAYER_PATH` environment variable.

```

1  {
2      "file_format_version": "1.1.2",
3      "layer": {
4          "name": "VK_LAYER_MY_DEBUG_LAYER",
5          "type": "GLOBAL",
6          "library_path": "my_debug_layer.so",
7          "api_version": "1.3.0",
8          "implementation_version": "1",
9          "description": "A simple Vulkan debugging layer"
10     }
11 }

```

11 }

Set the environment variable:

```
1 export VK_INSTANCE_LAYERS=VK_LAYER_MY_DEBUG_LAYER
2 export VK_LAYER_PATH=/path/to/layer
```

84.4 Recording Frame Data

To debug Vulkan frames, you need to capture and store data related to:

1. Command buffers: Record Vulkan commands for each frame.
2. GPU resources: Save textures, buffers, and shaders for analysis.
3. Frame state: Capture Vulkan pipeline states, descriptors, and other settings.

84.4.1 Intercepting Command Buffers

Intercept `vkBeginCommandBuffer`, `vkEndCommandBuffer`, and `vkCmd*` functions to record frame commands.

```
1 // Intercept vkCmdDraw
2 VKAPI_ATTR void VKAPI_CALL vkCmdDraw(
3     VkCommandBuffer commandBuffer,
4     uint32_t vertexCount,
5     uint32_t instanceCount,
6     uint32_t firstVertex,
7     uint32_t firstInstance) {
8     // Log or store draw call details
9     printf("Draw Call: vertexCount=%u, instanceCount=%u\n", vertexCount, instanceCount);
10
11     // Call the real vkCmdDraw
12     real_vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
13 }
```

84.4.2 Capturing Textures and Buffers

Intercept `vkCreateImage` and `vkCreateBuffer` to track GPU resources. Use `vkCmdCopyImageToBuffer` or similar commands to read data back from GPU memory.

```
1 // Capture texture data
2 void captureTexture(VkImage image, VkDevice device, VkCommandBuffer commandBuffer) {
3     // Transition image to a readable layout
4     // Copy image to a staging buffer
5     // Map and save the data
6 }
```

84.5 Building a Visual Debugger

After recording frame data, create a graphical user interface (GUI) to visualize it. Libraries like Dear ImGui or Qt can be used for this purpose.

84.5.1 Creating the Frame Viewer

Design a viewer that:

- Displays a list of recorded frames.
- Shows command buffer contents, draw calls, and pipeline states.
- Renders captured textures and framebuffer attachments.

Example using Dear ImGui:

```

1 void renderFrameViewer() {
2     ImGui::Begin("Frame Debugger");
3
4     // Display captured frames
5     for (const auto& frame : recordedFrames) {
6         if (ImGui::TreeNode("Frame")) {
7             for (const auto& cmd : frame.commands) {
8                 ImGui::Text("Command: %s", cmd.name.c_str());
9             }
10            ImGui::TreePop();
11        }
12    }
13
14    ImGui::End();
15 }
```

84.5.2 Rendering Captured Textures

Use Vulkan or OpenGL to render captured textures in the debugger. Transfer image data to the debugger's rendering context and display it on the screen.

84.6 Summary

Building a Vulkan debugging tool provides valuable insights into Vulkan's inner workings and helps troubleshoot complex issues in Vulkan applications. By intercepting API calls and recording frame data, you can create a functional RenderDoc-like tool to debug and visualize Vulkan rendering workflows. While this chapter covers the basics, further extensions could include shader debugging, timeline analysis, and advanced GPU profiling.

85

Add Vulkan Support to Blender: Create a plugin for Vulkan rendering

85.1 Introduction

Blender is one of the most popular open-source 3D content creation suites, supporting a wide range of rendering engines. While Blender's built-in rendering engines like Cycles and Eevee are powerful, adding Vulkan support can provide a modern and efficient rendering backend. In this chapter, we will explore how to integrate Vulkan into Blender by creating a custom rendering plugin.

This chapter assumes familiarity with Vulkan basics and the Blender Python API or its C++ source code.

85.2 Overview of Blender Rendering Plugins

Blender supports extensibility through its Python API and, for more advanced rendering engines, direct integration with its source code. We will focus on creating a minimal Vulkan rendering plugin to draw a simple scene from Blender.

The steps include:

1. Setting up Blender's development environment.
2. Creating a Vulkan rendering context within Blender.
3. Implementing scene data export for rendering.
4. Drawing objects with Vulkan.

85.3 Set Up Blender Development Environment

To create a Vulkan rendering plugin, you need a working Blender development environment. Start by downloading the Blender source code from its official repository.

85.3.1 Dependencies and Build Tools

Blender requires several dependencies to build, including Python, CMake, and libraries like OpenGL and SDL. Install the required packages for your platform by following the official Blender build instructions.

For Vulkan, ensure that the Vulkan SDK is installed on your system. This provides the headers, libraries, and validation layers necessary for Vulkan development.

85.3.2 Compiling Blender with Plugin Support

Build Blender with support for external plugins. This involves enabling the necessary flags in Blender's CMake configuration.

```
1 cmake -S . -B build -DWITH_PYTHON=ON -DWITH_MOD_VULKAN=ON
2 make -j$(nproc)
```

85.4 Creating the Vulkan Rendering Context

A rendering context is required to use Vulkan for drawing. In the context of Blender, this involves setting up Vulkan instance creation, physical and logical devices, and swapchain management.

85.4.1 Integrating Vulkan Initialization

Within Blender, modify or extend the rendering backend to include Vulkan. Create a new Vulkan module in the Blender source code:

```
1 // Initialize Vulkan instance
2 VkInstance createVulkanInstance() {
3     VkInstanceCreateInfo instanceCreateInfo = {};
4     instanceCreateInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
5     instanceCreateInfo.pApplicationInfo = nullptr;
6
7     VkInstance instance;
8     VkResult result = vkCreateInstance(&instanceCreateInfo, nullptr, &instance);
9     if (result != VK_SUCCESS) {
10         throw std::runtime_error("Failed to create Vulkan instance");
11     }
12     return instance;
13 }
14
15 // Create swapchain for Blender viewport
16 VkSwapchainKHR createSwapchain(VkDevice device, VkSurfaceKHR surface, VkExtent2D extent) {
```

```

17  VkSwapchainCreateInfoKHR swapchainCreateInfo = {};
18  swapchainCreateInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
19  swapchainCreateInfo.surface = surface;
20  swapchainCreateInfo.minImageCount = 2;
21  swapchainCreateInfo.imageFormat = VK_FORMAT_B8G8R8A8_SRGB;
22  swapchainCreateInfo.imageExtent = extent;
23
24  VkSwapchainKHR swapchain;
25  VkResult result = vkCreateSwapchainKHR(device, &swapchainCreateInfo, nullptr, &swapchain);
26  if (result != VK_SUCCESS) {
27      throw std::runtime_error("Failed to create swapchain");
28  }
29  return swapchain;
30 }

```

85.4.2 Viewport Rendering

Modify Blender's viewport rendering pipeline to use Vulkan for drawing. Hook into the viewport's render callbacks and direct drawing commands to the Vulkan backend.

85.5 Exporting Scene Data for Vulkan Rendering

To render a scene, Blender's data (meshes, materials, and lights) must be exported and translated into Vulkan-compatible formats.

85.5.1 Exporting Mesh Data

Access Blender's geometry data and convert it into vertex and index buffers for Vulkan.

```

1  import bpy
2  import struct
3
4  # Export mesh data for Vulkan
5  def export_mesh_data(obj):
6      vertices = []
7      indices = []
8
9      mesh = obj.data
10     for vertex in mesh.vertices:
11         vertices.append((vertex.co.x, vertex.co.y, vertex.co.z))
12
13     for poly in mesh.polygons:
14         indices.extend(poly.vertices)
15
16     # Save to a binary file
17     with open("mesh_data.bin", "wb") as f:
18         f.write(struct.pack("f" * len(vertices) * 3, *sum(vertices, ())))
19         f.write(struct.pack("I" * len(indices), *indices))
20
21     # Call the export function for an active object
22     export_mesh_data(bpy.context.active_object)

```

85.5.2 Exporting Materials and Textures

Similarly, extract material and texture data. Blender's materials can be converted into Vulkan descriptor sets and passed to shaders.

85.6 Rendering the Scene

With Vulkan initialized and Blender data exported, draw the scene using Vulkan's graphics pipeline. Implement shaders to process vertex data and apply materials.

```

1  // Vertex Shader (GLSL)
2  #version 450
3  layout(location = 0) in vec3 inPosition;
4  layout(location = 1) in vec2 inTexCoord;
5
6  layout(location = 0) out vec2 fragTexCoord;
7
8  void main() {
9      gl_Position = vec4(inPosition, 1.0);
10     fragTexCoord = inTexCoord;
11 }
12
13 // Fragment Shader (GLSL)
14 #version 450
15 layout(location = 0) in vec2 fragTexCoord;
16 layout(location = 0) out vec4 outColor;
17
18 layout(binding = 0) uniform sampler2D textureSampler;
19
20 void main() {
21     outColor = texture(textureSampler, fragTexCoord);
22 }

```

Use Vulkan commands to bind buffers, set descriptor sets, and issue draw calls.

85.7 Testing the Plugin

Once the Vulkan rendering plugin is implemented, test it within Blender by setting it as the active rendering engine. Render simple scenes to verify that Vulkan handles geometry, materials, and viewport updates correctly.

85.8 Summary

Adding Vulkan support to Blender provides a platform to explore modern rendering techniques and optimize for high-performance graphics workflows. While this chapter introduced the basics, you can extend the plugin to support advanced features like ray tracing, post-processing, and GPU-based simulations. By integrating Vulkan into Blender, you unlock the potential for cutting-edge graphics directly within one of the most versatile 3D creation tools.

86

Integrate Vulkan with Qt: Build Vulkan-based GUI applications with Qt

86.1 Introduction

Qt is a powerful framework for building cross-platform GUI applications, and combining it with Vulkan enables the creation of high-performance graphics applications with modern rendering techniques. In this chapter, we will explore how to integrate Vulkan into a Qt application. The focus will be on setting up a Vulkan rendering context within a Qt window and rendering a simple triangle.

86.2 Why Use Qt with Vulkan?

Qt provides a high-level abstraction for GUI development, making it easier to manage application interfaces and user interaction. Vulkan, on the other hand, offers low-level access to GPU features. Integrating the two allows developers to build applications that combine robust GUIs with advanced rendering.

Some potential use cases include:

- Developing graphics editors and visualization tools.
- Creating real-time simulations with user controls.
- Building game engines with integrated UI.

86.3 Setting Up a Qt Project with Vulkan

Before we dive into Vulkan integration, set up a basic Qt application.

86.3.1 Install Qt

Download and install the Qt framework from the official website or your system's package manager. Ensure you have the Qt Creator IDE and necessary development tools.

86.3.2 Create a New Qt Application

In Qt Creator, create a new application:

1. Open Qt Creator and select **File > New File or Project**.
2. Choose **Qt Widgets Application** or **Qt Quick Application**.
3. Follow the setup wizard to create the project.

86.3.3 Add Vulkan SDK to the Project

Integrate the Vulkan SDK into your Qt project. Update the project's `CMakeLists.txt` or `.pro` file to include Vulkan headers and libraries.

For `CMakeLists.txt` :

```
1 find_package(Vulkan REQUIRED)
2
3 target_include_directories(MyApp PRIVATE ${Vulkan_INCLUDE_DIRS})
4 target_link_libraries(MyApp PRIVATE Vulkan::Vulkan)
```

For `.pro` files:

```
1 LIBS += -lvulkan
2 INCLUDEPATH += /path/to/vulkan/sdk/include
```

86.4 Creating a Vulkan Window in Qt

Qt's `QWindow` and `QWidget` classes can be used to host a Vulkan rendering surface. Here, we will create a custom `QVulkanWindow` class.

86.4.1 Define a Custom Vulkan Window

Create a new class `VulkanWindow`, inheriting from `QWindow`. This class will handle Vulkan initialization and rendering.

```

1  #include <QVulkanInstance>
2  #include <QVulkanWindow>
3  #include <vulkan/vulkan.h>
4
5  class VulkanWindow : public QVulkanWindow {
6  public:
7      VulkanWindow() = default;
8
9  protected:
10     void initResources() override {
11         // Initialize Vulkan resources
12         initializeVulkan();
13     }
14
15     void initSwapChainResources() override {
16         // Set up swapchain-dependent resources
17     }
18
19     void releaseSwapChainResources() override {
20         // Clean up swapchain resources
21     }
22
23     void releaseResources() override {
24         // Clean up Vulkan resources
25     }
26
27     void startNextFrame() override {
28         // Render the next frame
29         renderFrame();
30     }
31
32 private:
33     VkInstance vulkanInstance;
34     VkDevice device;
35     VkSwapchainKHR swapchain;
36
37     void initializeVulkan();
38     void renderFrame();
39 };

```

86.4.2 Initialize Vulkan Resources

Implement `initializeVulkan()` to create the Vulkan instance, physical device, and logical device.

```

1  void VulkanWindow::initializeVulkan() {
2      // Create Vulkan instance
3      VkInstanceCreateInfo instanceInfo = {};
4      instanceInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
5
6      if (vkCreateInstance(&instanceInfo, nullptr, &vulkanInstance) != VK_SUCCESS) {
7          throw std::runtime_error("Failed to create Vulkan instance");
8      }
9
10     // Choose a physical device (GPU)
11     uint32_t deviceCount = 0;
12     vkEnumeratePhysicalDevices(vulkanInstance, &deviceCount, nullptr);
13     if (deviceCount == 0) {
14         throw std::runtime_error("No Vulkan-capable GPU found");
15     }
16
17     std::vector<VkPhysicalDevice> devices(deviceCount);

```



```

18 vkEnumeratePhysicalDevices(vulkanInstance, &deviceCount, devices.data());
19 VkPhysicalDevice physicalDevice = devices[0];
20
21 // Create logical device
22 VkDeviceCreateInfo deviceInfo = {};
23 deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
24
25 if (vkCreateDevice(physicalDevice, &deviceInfo, nullptr, &device) != VK_SUCCESS) {
26     throw std::runtime_error("Failed to create Vulkan device");
27 }
28 }

```

86.4.3 Render a Triangle

In `renderFrame()`, submit Vulkan commands to draw a triangle.

```

1 void VulkanWindow::renderFrame() {
2     // Record command buffer for drawing
3     // Submit to the graphics queue
4     // Present the image
5     requestUpdate(); // Schedule the next frame
6 }

```

86.5 Integrating Vulkan Window with Qt Widgets

Embed the Vulkan window in a `QWidget`-based application. Use a `QWidget` container to host the Vulkan rendering surface.

```

1 #include <QWidget>
2 #include <QVBoxLayout>
3 #include <VulkanWindowRenderer>
4
5 int main(int argc, char **argv) {
6     QApplication app(argc, argv);
7
8     QWidget mainWindow;
9     QVBoxLayout layout(&mainWindow);
10
11     VulkanWindow *vulkanWindow = new VulkanWindow;
12     QWidget *container = QWidget::createWindowContainer(vulkanWindow);
13     layout.addWidget(container);
14
15     mainWindow.show();
16     return app.exec();
17 }

```

86.6 Advanced Features

Once the Vulkan integration is set up, you can add advanced features such as:

- **Texturing:** Load textures and bind them to shaders.
- **Post-Processing:** Implement effects like bloom or HDR.
- **GUI Overlays:** Combine Vulkan rendering with Qt's native GUI components.

86.7 Summary

Integrating Vulkan with Qt enables the development of rich, high-performance graphics applications with a user-friendly interface. By combining Vulkan's power with Qt's flexibility, you can create applications that are both visually stunning and highly interactive. This chapter has laid the foundation for such applications; the possibilities for expansion and customization are nearly limitless.

DRAFT
version
0.681

87

Create Vulkan Debugging Tools: Write tools for visualizing Vulkan resources

87.1 Introduction

Developing Vulkan applications can be complex due to its low-level nature. Debugging and profiling Vulkan applications require specialized tools to visualize resources such as buffers, textures, pipelines, and command queues. In this chapter, we will create a basic debugging tool that allows developers to inspect Vulkan resources during runtime.

By integrating a resource visualization tool directly into your application, you can gain real-time insights into your Vulkan rendering pipeline, helping to identify bottlenecks and troubleshoot rendering issues.

87.2 Why Create Debugging Tools?

Existing tools like RenderDoc or Vulkan validation layers are invaluable but are not always sufficient for application-specific debugging. Custom debugging tools tailored to your application can:

- Provide a real-time view of Vulkan resources.
- Allow inspection of application-specific states and data.
- Help with dynamic resource tracking, such as memory usage and pipeline statistics.
- Enable the integration of debugging features directly into the application.

87.3 Setting Up the Debugging Tool Framework

To build a Vulkan debugging tool, we'll start by creating a basic framework. This framework will include:

1. A debug overlay GUI to display resource information.
2. A mechanism to capture Vulkan resource states.
3. A way to visualize the data interactively.

We will use the `Dear ImGui` library for the GUI, as it integrates well with Vulkan and provides a simple interface for building debug overlays.

87.3.1 Integrating Dear ImGui with Vulkan

First, add Dear ImGui to your project. You can download it from the official repository and integrate it into your build system.

```
1 add_subdirectory(imgui)
2 target_link_libraries(MyApp PRIVATE imgui)
```

Next, set up ImGui to render directly on top of your Vulkan swapchain. This involves creating Vulkan resources for ImGui and hooking into your rendering loop.

```
1 // Initialize ImGui for Vulkan
2 ImGui_ImplVulkan_InitInfo init_info = {};
3 init_info.Instance = vulkanInstance;
4 init_info.PhysicalDevice = physicalDevice;
5 init_info.Device = device;
6 init_info.Queue = graphicsQueue;
7 init_info.DescriptorPool = imguiDescriptorPool;
8 ImGui_ImplVulkan_Init(&init_info, renderPass);
```

Render the ImGui interface in the main rendering loop:

```
1 // Start a new ImGui frame
2 ImGui_ImplVulkan_NewFrame();
3 ImGui::NewFrame();
4
5 // Draw the debug UI
6 ImGui::Begin("Vulkan Debugger");
7 ImGui::Text("Resource Information:");
8 ImGui::Text("Frame Time: %.2f ms", frameTime);
9 ImGui::End();
10
11 // Render the ImGui commands
12 ImGui::Render();
13 ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(), commandBuffer);
```

87.4 Displaying Vulkan Resource Information

To debug Vulkan applications, you'll want to visualize key resources like buffers, textures, and pipelines. Below are examples of how to retrieve and display this information.

87.4.1 Buffer Viewer

Capture buffer contents using Vulkan's staging buffers and display them in the debug overlay.

```

1 // Map buffer data to host-visible memory
2 void* data;
3 vkMapMemory(device, bufferMemory, 0, bufferSize, 0, &data);
4
5 // Copy data to a readable format
6 std::vector<float> bufferContents(bufferSize / sizeof(float));
7 memcpy(bufferContents.data(), data, bufferSize);
8
9 // Display the data in the ImGui overlay
10 ImGui::Text("Buffer Contents:");
11 for (size_t i = 0; i < bufferContents.size(); ++i) {
12     ImGui::Text("Index %zu: %.2f", i, bufferContents[i]);
13 }
14
15 vkUnmapMemory(device, bufferMemory);

```

87.4.2 Texture Viewer

To visualize textures, copy the image to a host-visible buffer and render it in the debug overlay.

```

1 // Copy the image to a host-visible buffer
2 void VulkanDebugger::CopyImageToHostBuffer(VkImage image, VkBuffer hostBuffer) {
3     VkCommandBuffer commandBuffer = BeginOneTimeCommands();
4
5     // Transition the image to VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL
6     VkImageMemoryBarrier barrier = {};
7     barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
8     barrier.oldLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
9     barrier.newLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
10    barrier.image = image;
11    vkCmdPipelineBarrier(commandBuffer, ...);
12
13    // Copy the image
14    VkBufferImageCopy copyRegion = {};
15    vkCmdCopyImageToBuffer(commandBuffer, image, VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL, hostBuffer, 1, &
        ↳ copyRegion);
16
17    EndOneTimeCommands(commandBuffer);
18 }
19
20 // Render the texture in ImGui
21 ImGui::Image((ImGuiTextureID)textureId, ImVec2(256, 256));

```

87.4.3 Pipeline Viewer

Display Vulkan pipeline statistics, such as active shaders, vertex input configuration, and blend states.

```

1 // Retrieve pipeline state information
2 VkPipeline pipeline = GetActivePipeline();
3 ImGui::Text("Pipeline State:");
4 ImGui::Text("Vertex Shader: %s", pipelineState.vertexShader.c_str());
5 ImGui::Text("Fragment Shader: %s", pipelineState.fragmentShader.c_str());
6 ImGui::Text("Blend Mode: %s", pipelineState.blendMode.c_str());

```


87.5 Advanced Debugging Features

Enhance your debugging tool with advanced features:

- **Frame Capture:** Save Vulkan commands and resource states for offline analysis.
- **Memory Tracking:** Monitor GPU memory usage for buffers, images, and descriptor sets.
- **Command Buffer Inspection:** Visualize command buffer contents, including draw calls and barriers.
- **Performance Metrics:** Integrate Vulkan timestamp queries to measure GPU execution times.

87.6 Summary

Creating a custom debugging tool, you gain greater insight into the inner workings of your Vulkan application. While third-party tools like RenderDoc are invaluable, a tailored solution can provide additional flexibility and help debug application-specific issues. This chapter has laid the groundwork for building a Vulkan debugging tool, enabling you to visualize and analyze your resources effectively.

Part X

Miscellaneous Projects

88

Render a Wireframe Model: Add wireframe rendering

88.1 Introduction

Rendering a model in wireframe mode is a common technique used in 3D applications for debugging, visualization, or stylistic purposes. Vulkan provides direct support for wireframe rendering through its rasterization state settings. In this chapter, we will explore how to render a 3D model in wireframe mode using Vulkan. This process involves configuring the graphics pipeline and optionally toggling between wireframe and solid rendering modes dynamically.

88.2 Setting Up Wireframe Rendering

Wireframe rendering in Vulkan is achieved by configuring the rasterization state to render triangle edges instead of filled surfaces. This is controlled using the `VkPolygonMode` setting in the rasterization state of the graphics pipeline.

88.2.1 Configuring the Rasterization State

To enable wireframe rendering, modify the pipeline's rasterization state as follows:

```
1 // Configure rasterization state for wireframe rendering
2 VkPipelineRasterizationStateCreateInfo rasterizationState = {};
3 rasterizationState.sType = VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
4 rasterizationState.polygonMode = VK_POLYGON_MODE_LINE; // Wireframe mode
5 rasterizationState.cullMode = VK_CULL_MODE_BACK_BIT; // Cull back-facing polygons
6 rasterizationState.frontFace = VK_FRONT_FACE_CLOCKWISE; // Define front face
```

```
7 rasterizationState.lineWidth = 1.0f; // Line width for edges
```

Wireframe mode requires the GPU to support the `VK_POLYGON_MODE_LINE` feature. Ensure your physical device has this feature enabled:

```
1 // Check if wireframe mode is supported
2 VkPhysicalDeviceFeatures supportedFeatures;
3 vkGetPhysicalDeviceFeatures(physicalDevice, &supportedFeatures);
4 if (!supportedFeatures.fillModeNonSolid) {
5     throw std::runtime_error("Wireframe rendering not supported on this device.");
6 }
```

88.2.2 Recreating the Graphics Pipeline

Since the rasterization state is part of the pipeline, changes to this setting require recreating the graphics pipeline. Define two separate pipelines for solid and wireframe rendering, or dynamically toggle the `polygonMode` setting during pipeline creation.

```
1 // Create pipeline for wireframe rendering
2 VkGraphicsPipelineCreateInfo pipelineInfo = {};
3 pipelineInfo.pRasterizationState = &rasterizationState;
4 vkCreateGraphicsPipelines(device, pipelineCache, 1, &pipelineInfo, nullptr, &wireframePipeline);
```

88.3 Rendering in Wireframe Mode

To render a wireframe model, bind the appropriate pipeline and issue the draw calls as usual.

```
1 // Bind wireframe pipeline
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, wireframePipeline);
3
4 // Bind vertex and index buffers
5 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
6 vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT32);
7
8 // Issue draw call
9 vkCmdDrawIndexed(commandBuffer, indexCount, 1, 0, 0, 0);
```

88.4 Dynamic Wireframe Toggling

For applications requiring both wireframe and solid rendering, provide a toggle mechanism. Use a user interface, such as `Dear ImGui`, to switch between modes dynamically.

```
1 // ImGui toggle for wireframe mode
2 static bool wireframeMode = false;
3 ImGui::Checkbox("Wireframe Mode", &wireframeMode);
4
5 // Bind the appropriate pipeline based on user input
6 VkPipeline selectedPipeline = wireframeMode ? wireframePipeline : solidPipeline;
7 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, selectedPipeline);
```

88.5 Adjusting Line Width

Vulkan allows you to control the thickness of wireframe edges using the `lineWidth` property in the rasterization state. Note that some GPUs require enabling the `wideLines` feature to support line widths greater than 1.0.

```
1 // Check if wide lines are supported
2 if (supportedFeatures.wideLines) {
3     rasterizationState.lineWidth = 2.0f; // Set a thicker line width
4 } else {
5     rasterizationState.lineWidth = 1.0f; // Fall back to default
6 }
```

88.6 Practical Applications of Wireframe Rendering

Wireframe rendering is useful for:

- **Debugging Geometry:** Identifying missing or incorrectly connected vertices.
- **Visualizing Mesh Topology:** Understanding the structure of a model's geometry.
- **Low-Detail Rendering:** Rendering models in a simplified manner for non-photorealistic effects.

88.7 Summary

Wireframe rendering is a valuable feature for debugging and visualization in 3D applications. By leveraging Vulkan's rasterization state, you can easily render models in wireframe mode and provide users with dynamic toggling capabilities. This chapter has covered the steps to implement wireframe rendering, from configuring the rasterization state to integrating it into your rendering loop.

89

Add Tessellation Shaders: Implement tessellation for smoother models

89.1 Introduction

Tessellation shaders are a powerful feature in modern graphics pipelines that allow for dynamically increasing or decreasing the detail level of a 3D model. By dividing (or "tessellating") a model's surfaces into smaller primitives, you can achieve smoother curves and finer details without altering the original geometry. This chapter will guide you through implementing tessellation shaders in Vulkan, enabling dynamic control over model detail based on performance and visual fidelity requirements.

89.2 Overview of Tessellation Stages

Vulkan's tessellation pipeline consists of three programmable stages:

- **Tessellation Control Shader (TCS):** Controls the level of tessellation for each primitive.
- **Tessellation Evaluation Shader (TES):** Calculates the positions of the subdivided vertices.
- **Fixed-Function Tessellation Unit:** Performs the actual subdivision based on the control shader's output.

These stages work together to process patches of primitives and generate finely detailed surfaces.

89.3 Enabling Tessellation in Vulkan

To use tessellation shaders, you must enable support for tessellation in your Vulkan pipeline and provide shader modules for the TCS and TES stages.

89.3.1 Updating the Pipeline State

The pipeline configuration must include the tessellation state and shader stages for tessellation. The tessellation state determines the number of control points per patch.

```
1 // Configure the tessellation state
2 VkPipelineTessellationStateCreateInfo tessellationState = {};
3 tessellationState.sType = VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO;
4 tessellationState.patchControlPoints = 3; // For triangular patches
```

Include the tessellation shader stages in the pipeline's shader stage configuration:

```
1 // Load and set tessellation control and evaluation shaders
2 VkPipelineShaderStageCreateInfo tcsStage = {};
3 tcsStage.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
4 tcsStage.stage = VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT;
5 tcsStage.module = tessellationControlShaderModule;
6 tcsStage.pName = "main";
7
8 VkPipelineShaderStageCreateInfo tesStage = {};
9 tesStage.sType = VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
10 tesStage.stage = VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT;
11 tesStage.module = tessellationEvaluationShaderModule;
12 tesStage.pName = "main";
```

Finally, add these shader stages and tessellation state to the pipeline creation info:

```
1 // Add tessellation stages to the pipeline creation
2 VkGraphicsPipelineCreateInfo pipelineInfo = {};
3 pipelineInfo.pStages = shaderStages; // Include TCS and TES
4 pipelineInfo.pTessellationState = &tessellationState;
```

89.3.2 Specifying Patch Topology

In Vulkan, tessellation operates on patches instead of individual primitives. Specify the input topology as `VK_PRIMITIVE_TOPOLOGY_PATCH_LIST`:

```
1 // Set input topology to patches
2 VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};
3 inputAssembly.sType = VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
4 inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_PATCH_LIST;
5 inputAssembly.primitiveRestartEnable = VK_FALSE;
```

89.4 Writing Tessellation Shaders

The tessellation control and evaluation shaders are written in GLSL and compiled to SPIR-V. Below are examples of a simple TCS and TES for rendering a smooth triangle.

89.4.1 Tessellation Control Shader (TCS)

The TCS controls the tessellation level for each edge of the patch.

```

1 // tessellation_control.glsl
2 #version 450
3 layout(vertices = 3) out;
4
5 void main() {
6     // Pass through vertex data
7     gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
8
9     // Set tessellation levels (outer and inner levels)
10    if (gl_InvocationID == 0) {
11        gl_TessLevelOuter[0] = 4.0;
12        gl_TessLevelOuter[1] = 4.0;
13        gl_TessLevelOuter[2] = 4.0;
14        gl_TessLevelInner[0] = 3.0;
15    }
16 }
```

89.4.2 Tessellation Evaluation Shader (TES)

The TES calculates the position of each tessellated vertex.

```

1 // tessellation_evaluation.glsl
2 #version 450
3 layout(triangles, equal_spacing, cw) in;
4
5 void main() {
6     vec3 p0 = gl_in[0].gl_Position.xyz;
7     vec3 p1 = gl_in[1].gl_Position.xyz;
8     vec3 p2 = gl_in[2].gl_Position.xyz;
9
10    // Compute barycentric coordinates
11    vec3 barycentric = vec3(gl_TessCoord.x, gl_TessCoord.y, gl_TessCoord.z);
12
13    // Interpolate vertex position
14    vec3 position = barycentric.x * p0 + barycentric.y * p1 + barycentric.z * p2;
15
16    gl_Position = vec4(position, 1.0);
17 }
```

89.5 Drawing Tessellated Geometry

To draw tessellated geometry, bind the pipeline with tessellation shaders and issue a draw call:

```
1 // Bind the tessellation pipeline
```

```

2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, tessellationPipeline);
3
4 // Bind vertex buffer and issue draw call
5 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
6 vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);

```

89.6 Dynamic Tessellation Levels

Tessellation levels can be adjusted dynamically by passing uniform data to the TCS. This allows you to control the detail level based on camera distance or other factors.

```

1 // Uniform data for tessellation levels
2 layout(binding=0) uniform TessellationUniforms {
3     float tessLevelOuter;
4     float tessLevelInner;
5 };
6
7 void main() {
8     if (gl_InvocationID == 0) {
9         gl_TessLevelOuter[0] = tessLevelOuter;
10        gl_TessLevelOuter[1] = tessLevelOuter;
11        gl_TessLevelOuter[2] = tessLevelOuter;
12        gl_TessLevelInner[0] = tessLevelInner;
13    }
14 }

```

89.7 Summary

Tessellation shaders offer a powerful way to increase the visual fidelity of 3D models by dynamically subdividing geometry into smaller primitives. This chapter covered the configuration of Vulkan's tessellation pipeline, the writing of control and evaluation shaders, and the process of rendering tessellated geometry. By incorporating tessellation into your Vulkan projects, you can achieve smoother surfaces and more detailed models while maintaining efficient performance.

90

Render Text with Vulkan: Use a font library for text rendering

90.1 Introduction

Rendering text in Vulkan requires combining efficient texture management with accurate glyph placement. While Vulkan doesn't have built-in support for text rendering, integrating a font library like **FreeType** simplifies the process by providing the tools to load and rasterize fonts. In this chapter, we will explore how to render text using Vulkan by leveraging a font library and creating a texture atlas for glyphs.

90.2 Overview of Text Rendering

Text rendering involves several steps:

1. **Font Loading:** Load a font file and extract individual glyphs using a font library like FreeType.
2. **Texture Atlas Creation:** Combine all glyphs into a single texture for efficient rendering.
3. **Glyph Placement:** Calculate screen positions for glyphs to form words and sentences.
4. **Rendering Pipeline:** Use Vulkan's rendering pipeline to draw textured quads representing the glyphs.

90.3 Setting Up FreeType for Font Loading

FreeType is a widely used library for loading and rasterizing fonts. Begin by including FreeType in your project and initializing it:

```

1  // Initialize FreeType library
2  FT_Library ft;
3  if (FT_Init_FreeType(&ft)) {
4      throw std::runtime_error("Failed to initialize FreeType library");
5  }
6
7  // Load a font face
8  FT_Face face;
9  if (FT_New_Face(ft, "path/to/font.ttf", 0, &face)) {
10     throw std::runtime_error("Failed to load font");
11 }
12
13 // Set font size
14 FT_Set_Pixel_Sizes(face, 0, 48);

```

Each glyph in the font is rasterized into a bitmap, which can then be used to generate textures.

90.4 Creating a Glyph Texture Atlas

A texture atlas consolidates all the glyphs into a single texture, reducing the number of draw calls and texture bindings.

90.4.1 Generating the Atlas

Iterate over the desired characters and rasterize each glyph. Copy the glyph bitmaps into a single larger texture:

```

1  std::map<char, Glyph> glyphs;
2  int atlasWidth = 0, atlasHeight = 0;
3
4  // Pre-calculate texture size
5  for (char c = 32; c < 128; c++) {
6      if (FT_Load_Char(face, c, FT_LOAD_RENDER)) continue;
7      atlasWidth += face->glyph->bitmap.width;
8      atlasHeight = std::max(atlasHeight, face->glyph->bitmap.rows);
9  }
10
11 // Create a Vulkan texture
12 VkImage glyphAtlasImage = createImage(atlasWidth, atlasHeight, VK_FORMAT_R8_UNORM);
13
14 // Copy glyphs into the texture
15 int offsetX = 0;
16 for (char c = 32; c < 128; c++) {
17     if (FT_Load_Char(face, c, FT_LOAD_RENDER)) continue;
18
19     // Copy the glyph bitmap into the texture at the appropriate position
20     uploadGlyphToAtlas(glyphAtlasImage, face->glyph->bitmap, offsetX, 0);
21
22     // Store glyph metadata
23     glyphs[c] = { offsetX, face->glyph->bitmap.width, face->glyph->bitmap.rows,
24                  face->glyph->bitmap_left, face->glyph->bitmap_top };

```

```

25     offsetX += face->glyph->bitmap.width;
26 }

```

90.4.2 Sampling the Texture in Shaders

Bind the texture atlas in your Vulkan pipeline and sample it in the fragment shader:

```

1  // text_fragment.glsl
2  #version 450
3  layout(binding = 0) uniform sampler2D glyphAtlas;
4
5  layout(location = 0) in vec2 texCoords;
6  layout(location = 0) out vec4 outColor;
7
8  void main() {
9      float alpha = texture(glyphAtlas, texCoords).r;
10     outColor = vec4(1.0, 1.0, 1.0, alpha); // White text
11 }

```

90.5 Placing Glyphs on the Screen

Each glyph is rendered as a textured quad. Compute the position and size of each quad based on the glyph metrics provided by FreeType.

```

1  // Render text
2  float x = startX, y = startY;
3  for (char c : text) {
4      Glyph& g = glyphs[c];
5
6      float xpos = x + g.bearingX;
7      float ypos = y - (g.height - g.bearingY);
8      float w = g.width;
9      float h = g.height;
10
11     // Define quad vertices
12     vertices.push_back({ xpos,      ypos + h, g.offsetX, 0 });
13     vertices.push_back({ xpos + w, ypos + h, g.offsetX + w, 0 });
14     vertices.push_back({ xpos,      ypos,      g.offsetX, h });
15
16     vertices.push_back({ xpos + w, ypos,      g.offsetX + w, h });
17     vertices.push_back({ xpos,      ypos,      g.offsetX, h });
18     vertices.push_back({ xpos + w, ypos + h, g.offsetX + w, 0 });
19
20     x += g.advanceX;
21 }

```

Upload the vertex data to a Vulkan buffer and issue a draw call.

90.6 Integrating with Vulkan

To render the text, bind the pipeline, descriptor sets (containing the glyph atlas), and vertex buffer, then issue the draw commands:

```
1 // Bind pipeline and descriptor sets
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, textPipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, 1, &descriptorSet, 0
  ↪ , nullptr);
4
5 // Bind vertex buffer and draw text
6 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
7 vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);
```

90.7 Summary

Rendering text in Vulkan involves several steps, from loading font data to creating a texture atlas and rendering textured quads. Integrating a font library like FreeType, you can manage font glyphs efficiently and render high-quality text. This chapter provides a foundation for text rendering in Vulkan, which you can extend with advanced features such as dynamic fonts, colored text, and styling.

91

Add Compute-Based Occlusion Culling: Optimize scene rendering

91.1 Introduction

As scenes in modern applications grow more complex, rendering every object can become computationally expensive. **Occlusion culling** is a technique to improve performance by skipping the rendering of objects not visible in the scene. This chapter demonstrates how to use Vulkan's compute shaders to implement occlusion culling efficiently. By leveraging the GPU, we can parallelize visibility tests and reduce the burden on the graphics pipeline.

91.2 What is Occlusion Culling?

Occlusion culling determines whether an object is visible or hidden behind other objects in the scene. If an object is not visible (occluded), it can be excluded from further processing, saving GPU and CPU resources. Compute-based occlusion culling involves the following steps:

1. Perform visibility tests in a compute shader.
2. Use a bounding volume (e.g., a bounding box or sphere) for each object.
3. Write the results to a buffer, which is then used to filter draw calls.

91.3 Overview of the Implementation

The process involves:

1. **Frustum Culling:** Remove objects outside the camera's view.
2. **Depth Buffer Sampling:** Compare object bounding volumes against the depth buffer to determine occlusion.
3. **Cull Results Storage:** Store the results of visibility tests in a buffer.
4. **Draw Command Filtering:** Use the cull results to selectively issue draw commands.

91.4 Preparing the Data

Each object in the scene requires a bounding volume and transformation data. The bounding volumes are uploaded to a storage buffer accessible by the compute shader:

```

1 // Define bounding sphere data
2 struct BoundingSphere {
3     glm::vec3 center;
4     float radius;
5 };
6
7 // Create a storage buffer for bounding spheres
8 VkBuffer sphereBuffer;
9 VkDeviceMemory sphereMemory;
10 createBuffer(device, sizeof(BoundingSphere) * objectCount, VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, &sphereBuffer, &
    ↳ sphereMemory);
11
12 // Map the buffer and upload sphere data
13 BoundingSphere* spheres;
14 vkMapMemory(device, sphereMemory, 0, VK_WHOLE_SIZE, 0, (void*)&spheres);
15 for (int i = 0; i < objectCount; i++) {
16     spheres[i] = computeBoundingSphere(objects[i]);
17 }
18 vkUnmapMemory(device, sphereMemory);

```

91.5 Writing the Compute Shader

The compute shader performs depth tests against the bounding volumes to determine visibility:

```

1 // occlusion_culling.comp
2 #version 450
3
4 layout(local_size_x = 64) in;
5
6 // Input buffers
7 layout(binding = 0) readonly buffer BoundingSpheres {
8     vec4 spheres[]; // xyz = center, w = radius
9 };
10 layout(binding = 1) readonly uniform sampler2D depthBuffer;
11
12 // Output buffer
13 layout(binding = 2) writeonly buffer VisibilityResults {
14     uint visible[];
15 };
16
17 // Camera properties
18 uniform mat4 viewProjection;
19
20 void main() {

```

```

21     uint index = gl_GlobalInvocationID.x;
22
23     // Fetch bounding sphere
24     vec4 sphere = spheres[index];
25     vec4 centerClip = viewProjection * vec4(sphere.xyz, 1.0);
26
27     // Perform frustum culling
28     if (abs(centerClip.x) > centerClip.w || abs(centerClip.y) > centerClip.w || centerClip.z > centerClip.w ||
        ↪ centerClip.z < 0.0) {
29         visible[index] = 0;
30         return;
31     }
32
33     // Sample depth buffer to check occlusion
34     vec2 screenCoords = centerClip.xy / centerClip.w * 0.5 + 0.5;
35     float depth = texture(depthBuffer, screenCoords).r;
36     if (centerClip.z > depth) {
37         visible[index] = 0;
38     } else {
39         visible[index] = 1;
40     }
41 }

```

This shader performs frustum and occlusion culling and writes visibility results to the output buffer.

91.6 Dispatching the Compute Shader

The compute shader is dispatched with enough workgroups to cover all objects:

```

1 // Bind resources
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipelineLayout, 0, 1, &descriptorSet, 0,
    ↪ nullptr);
4
5 // Dispatch compute shader
6 vkCmdDispatch(commandBuffer, (objectCount + 63) / 64, 1, 1);

```

After the compute shader executes, the visibility results can be used to filter draw calls.

91.7 Using the Results to Filter Draw Commands

The visibility results buffer is read by the CPU to dynamically build a list of objects to render:

```

1 // Map visibility results buffer
2 uint* visibilityResults;
3 vkMapMemory(device, visibilityMemory, 0, VK_WHOLE_SIZE, 0, (void*)&visibilityResults);
4
5 // Filter objects based on visibility
6 std::vector<uint> visibleObjects;
7 for (uint i = 0; i < objectCount; i++) {
8     if (visibilityResults[i] == 1) {
9         visibleObjects.push_back(i);
10    }
11 }
12 vkUnmapMemory(device, visibilityMemory);

```



```
13
14 // Issue draw commands for visible objects
15 for (uint i : visibleObjects) {
16     vkCmdDrawIndexed(commandBuffer, objects[i].indexCount, 1, 0, 0, i);
17 }
```

91.8 Integrating with Vulkan Pipelines

The occlusion culling pipeline includes:

1. A compute pipeline for visibility tests.
2. A graphics pipeline that renders only visible objects.

Descriptor sets bind the buffers for spheres, depth textures, and visibility results. Synchronization ensures the compute shader completes before rendering begins.

91.9 Summary

Compute-based occlusion culling is a powerful optimization for complex scenes. By leveraging Vulkan's compute shaders, we can offload visibility determination to the GPU, minimizing unnecessary draw calls. This technique is especially beneficial for applications with a high object count, ensuring efficient resource utilization and improved performance.

92

Render Procedural Clouds: Create dynamic, realistic clouds

92.1 Introduction

Cloud rendering is an essential feature for many applications, from games to simulations. Procedural clouds offer a way to generate realistic, dynamic cloud formations without requiring pre-generated textures. This chapter explores creating volumetric procedural clouds using Vulkan, combining 3D noise generation and raymarching techniques.

92.2 Understanding Procedural Clouds

Procedural clouds rely on algorithms like Perlin or Worley noise to simulate the density and structure of clouds. These noise functions are used to represent the density field of the clouds in a 3D space. A common approach to render procedural clouds is to use raymarching, where rays are cast into the 3D density field to determine how light interacts with the cloud layers.

Key Concepts

1. **3D Noise Generation:** Generate noise in three dimensions to simulate cloud density.
2. **Raymarching:** Traverse the density field along rays cast from the camera to calculate color and opacity.
3. **Lighting:** Simulate how light interacts with clouds for realistic shading.

92.3 Setting Up the Compute Pipeline for 3D Noise

The first step is to generate a 3D noise texture that represents cloud density. This is accomplished using a compute shader:

```

1  // compute_cloud_density.comp
2  #version 450
3
4  layout(local_size_x=8, local_size_y=8, local_size_z=8) in;
5
6  // Output 3D texture for cloud density
7  layout(binding=0, rgba32f) writeonly uniform image3D densityTexture;
8
9  float hash(vec3 p) {
10     return fract(sin(dot(p, vec3(12.9898, 78.233, 45.164))) * 43758.5453);
11 }
12
13 // Simple Perlin-like noise function
14 float noise(vec3 p) {
15     vec3 i = floor(p);
16     vec3 f = fract(p);
17     float a = hash(i);
18     float b = hash(i + vec3(1.0, 0.0, 0.0));
19     float c = hash(i + vec3(0.0, 1.0, 0.0));
20     float d = hash(i + vec3(1.0, 1.0, 0.0));
21     vec2 u = f.xy * f.xy * (3.0 - 2.0 * f.xy);
22     return mix(a, b, u.x) + (c - a) * u.y;
23 }
24
25 void main() {
26     ivec3 pos = ivec3(gl_GlobalInvocationID.xyz);
27     vec3 p = vec3(pos) / vec3(imageSize(densityTexture));
28     float value = noise(p * 10.0); // Scale the noise
29     imageStore(densityTexture, pos, vec4(value));
30 }

```

Dispatching the Compute Shader

The compute shader is dispatched to generate the 3D noise texture:

```

1  // Create a 3D image for cloud density
2  VkImage densityImage;
3  VkDeviceMemory densityImageMemory;
4  create3DImage(device, VK_FORMAT_R32_SFLOAT, cloudResolution, &densityImage, &densityImageMemory);
5
6  // Create a compute pipeline
7  VkPipeline computePipeline = createComputePipeline(device, "compute_cloud_density.comp");
8
9  // Bind resources and dispatch
10 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
11 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipelineLayout, 0, 1, &descriptorSet, 0,
    ↪ nullptr);
12 vkCmdDispatch(commandBuffer, cloudResolution.x / 8, cloudResolution.y / 8, cloudResolution.z / 8);

```

92.4 Raymarching to Render Clouds

After generating the density field, the next step is to raymarch through it to render the clouds. This is implemented in a fragment shader:

```

1 // raymarch_clouds.frag
2 #version 450
3
4 layout(location = 0) out vec4 fragColor;
5
6 // Input 3D density texture
7 layout(binding = 0) uniform sampler3D densityTexture;
8
9 // Camera parameters
10 uniform vec3 cameraPosition;
11 uniform mat4 viewProjection;
12
13 // Raymarching parameters
14 const int MAX_STEPS = 100;
15 const float STEP_SIZE = 0.01;
16
17 vec3 rayDirection(vec2 uv) {
18     vec4 clip = vec4(uv, 1.0, 1.0);
19     vec4 eye = inverse(viewProjection) * clip;
20     return normalize(eye.xyz / eye.w - cameraPosition);
21 }
22
23 void main() {
24     vec2 uv = gl_FragCoord.xy / vec2(viewportSize);
25     vec3 rayDir = rayDirection(uv);
26     vec3 pos = cameraPosition;
27
28     vec4 accumulatedColor = vec4(0.0);
29     for (int i = 0; i < MAX_STEPS; ++i) {
30         float density = texture(densityTexture, pos).r;
31         vec4 color = vec4(vec3(density), density); // Basic shading
32         accumulatedColor = mix(accumulatedColor, color, density);
33         pos += rayDir * STEP_SIZE;
34
35         // Stop if fully opaque
36         if (accumulatedColor.a > 0.99) {
37             break;
38         }
39     }
40
41     fragColor = accumulatedColor;
42 }

```

92.5 Lighting the Clouds

To achieve realistic cloud shading, implement simple lighting based on a single directional light:

```

1 vec3 lightDirection = normalize(vec3(-1.0, -1.0, -1.0));
2 float lightIntensity = max(dot(lightDirection, normal), 0.0);
3 color *= lightIntensity;

```

92.6 Integrating into the Vulkan Pipeline

The final Vulkan pipeline involves:

1. **Compute Pass:** Generate the 3D noise texture.
2. **Render Pass:** Use the fragment shader to render the raymarched clouds.

92.7 Summary

Procedural clouds add depth and realism to scenes without requiring heavy texture assets. This chapter demonstrated how to use Vulkan's compute shaders and raymarching techniques to create dynamic, visually striking clouds. Experiment with different noise algorithms and lighting techniques to enhance the visual quality of your procedural clouds.

DRAFT
version
0.681

93

Create a Vulkan Benchmark Tool: Measure Vulkan performance on hardware

93.1 Introduction

Understanding the performance of a Vulkan-based application is critical for optimizing rendering pipelines and ensuring smooth experiences across various hardware. A Vulkan benchmark tool provides valuable insights by measuring frame times, GPU utilization, and other performance metrics. In this chapter, we will build a basic Vulkan benchmarking tool to analyze and visualize these metrics.

93.2 Purpose of a Vulkan Benchmark Tool

A benchmarking tool serves multiple purposes:

- **Evaluate Performance:** Measure how efficiently your application uses the GPU.
- **Optimize Pipelines:** Identify bottlenecks in rendering or compute workloads.
- **Compare Hardware:** Test and compare Vulkan performance across devices.
- **Monitor Real-Time Metrics:** Display frame rate (FPS), GPU utilization, and frame times in real-time.

93.3 Framework Overview

Our Vulkan benchmark tool will:

1. Record frame rendering times using timestamp queries.
2. Calculate average and peak frame times.
3. Display metrics such as FPS and GPU time.
4. Optionally write results to a file for later analysis.

93.4 Setting Up Timestamp Queries

Vulkan provides [timestamp queries](#) to measure GPU execution times. These queries capture timestamps at specific points in the command buffer.

Creating a Query Pool

The first step is to create a query pool for the timestamp queries:

```
1 // Create a query pool
2 VkQueryPoolCreateInfo queryPoolInfo{};
3 queryPoolInfo.sType = VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO;
4 queryPoolInfo.queryType = VK_QUERY_TYPE_TIMESTAMP;
5 queryPoolInfo.queryCount = 2; // Start and end timestamps
6
7 VkQueryPool queryPool;
8 vkCreateQueryPool(device, &queryPoolInfo, nullptr, &queryPool);
```

Adding Timestamps to the Command Buffer

Insert ‘vkCmdWriteTimestamp’ calls into the command buffer to capture start and end times for a frame:

```
1 // Begin recording
2 vkCmdWriteTimestamp(commandBuffer, VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, queryPool, 0);
3
4 // Rendering commands
5 drawFrame();
6
7 // End recording
8 vkCmdWriteTimestamp(commandBuffer, VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, queryPool, 1);
```

Retrieving Query Results

After submission, retrieve the timestamps using ‘vkGetQueryPoolResults’:

```
1 uint64_t timestamps[2];
2 vkGetQueryPoolResults(
3     device,
4     queryPool,
5     0, 2,
6     sizeof(timestamps),
7     timestamps,
8     sizeof(uint64_t),
9     VK_QUERY_RESULT_64_BIT
10 );
11
12 uint64_t startTime = timestamps[0];
13 uint64_t endTime = timestamps[1];
```

```

14
15 // Convert to nanoseconds using the timestamp period
16 double frameTimeNs = (endTime - startTime) * deviceProperties.limits.timestampPeriod;
17 double frameTimeMs = frameTimeNs / 1e6; // Convert to milliseconds

```

93.5 Calculating FPS and Metrics

To calculate FPS, average frame time, and peak frame time:

- **FPS:** `1000 / frameTimeMs`
- **Average Frame Time:** Maintain a moving average of frame times.
- **Peak Frame Time:** Track the highest frame time over a session.

```

1 // Update metrics
2 double fps = 1000.0 / frameTimeMs;
3 updateMetrics(frameTimeMs);

```

93.6 Visualizing the Metrics

Display the metrics in real-time using text rendering or an overlay GUI like [Dear ImGui](#):

```

1 // Example ImGui overlay
2 ImGui::Begin("Performance Metrics");
3 ImGui::Text("FPS: %.2f", fps);
4 ImGui::Text("Frame Time: %.2f ms", frameTimeMs);
5 ImGui::Text("Peak Frame Time: %.2f ms", peakFrameTime);
6 ImGui::End();

```

93.7 Exporting Results

Optionally, export the metrics to a CSV file for further analysis:

```

1 std::ofstream file("benchmark_results.csv");
2 file << "Frame,Time (ms),FPS\n";
3 for (const auto& frame : frames) {
4     file << frame.index << "," << frame.timeMs << "," << frame.fps << "\n";
5 }
6 file.close();

```

93.8 Advanced Features

You can enhance the benchmark tool with the following features:

- **Per-Pipeline Timing:** Measure execution times for specific pipeline stages using additional queries.
- **Hardware-Specific Metrics:** Use Vulkan extensions like `VK_KHR_performance_query` for detailed hardware counters.

- **Interactive Benchmarking:** Allow users to adjust rendering parameters dynamically.

93.9 Summary

Creating a Vulkan benchmark tool provides valuable insights into application performance and hardware capabilities. By leveraging timestamp queries and visualizing real-time metrics, you can identify bottlenecks and optimize your Vulkan pipelines. This chapter lays the foundation for building more advanced benchmarking tools tailored to your application's needs.

DRAFT
version
0.681

94

Render a Vulkan-Based UI Library: Develop a custom UI library

94.1 Introduction

Developing a Vulkan-based UI library from scratch offers a unique challenge and opportunity to understand Vulkan's flexibility and power in rendering complex 2D elements. This chapter focuses on creating a basic UI library capable of rendering elements such as buttons, sliders, and text boxes using Vulkan.

94.2 Goals of a Custom UI Library

A UI library is essential for applications requiring user interaction. While libraries like Dear ImGui provide ready-made solutions, building your own library offers complete control over the design and functionality. The primary goals of this chapter include:

- **Understand Vulkan Rendering for UI:** Learn how to efficiently render 2D elements with Vulkan.
- **Handle Input Events:** Capture mouse and keyboard inputs for interactive components.
- **Create Scalable UI Elements:** Design a system to dynamically adapt to varying screen resolutions.

94.3 Framework Overview

The framework for the Vulkan-based UI library involves:

1. Rendering 2D primitives (rectangles, lines, and text) using Vulkan.
2. Managing UI state (e.g., button presses or slider values).
3. Handling user inputs and integrating them with the Vulkan render loop.

94.4 Setting Up the UI Rendering System

Designing the UI Element Structure

Each UI element (e.g., buttons, sliders) can be represented as a structured object with properties such as position, size, and color.

```
1 struct UIElement {
2     glm::vec2 position;
3     glm::vec2 size;
4     glm::vec4 color;
5     bool isHovered = false;
6     bool isPressed = false;
7 };
```

Creating Vertex Buffers for 2D Primitives

To render a UI element, define its geometry in a vertex buffer. For example, a rectangle can be represented by two triangles:

```
1 // Vertices for a rectangle
2 std::vector<Vertex> vertices = {
3     {{position.x, position.y}, color},
4     {{position.x + size.x, position.y}, color},
5     {{position.x, position.y + size.y}, color},
6     {{position.x + size.x, position.y + size.y}, color},
7 };
```

Store these vertices in a Vulkan vertex buffer for efficient rendering.

```
1 // Create vertex buffer
2 VkBuffer vertexBuffer;
3 VkDeviceMemory vertexBufferMemory;
4 // Code to create and bind the Vulkan buffer
```

Rendering the UI Elements

Render the UI elements by recording draw commands in the command buffer:

```
1 // Bind pipeline and buffers
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, uiPipeline);
3 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
4
5 // Draw each UI element
6 vkCmdDraw(commandBuffer, vertices.size(), 1, 0, 0);
```

94.5 Handling Input Events

Input events such as mouse clicks and movements are crucial for interactivity. Use a windowing library like GLFW or SDL2 to capture input and map it to UI element actions.

Detecting Hover States

Check if the mouse position is within a UI element's bounds:

```
1 bool isHovered(const glm::vec2& mousePos, const UIElement& element) {
2     return mousePos.x > element.position.x &&
3         mousePos.x < element.position.x + element.size.x &&
4         mousePos.y > element.position.y &&
5         mousePos.y < element.position.y + element.size.y;
6 }
```

Update the hover state during the input handling phase:

```
1 element.isHovered = isHovered(mousePosition, element);
```

Handling Button Presses

Use mouse button events to detect clicks:

```
1 if (mouseButton == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
2     if (element.isHovered) {
3         element.isPressed = true;
4     }
5 }
```

94.6 Advanced Features for the UI Library

Once the basic rendering and input handling are in place, expand the UI library with additional features:

Text Rendering

Integrate a font library like `stb_truetype` or `Freetype` for rendering text. Use Vulkan's texture sampling to draw text glyphs.

```
1 // Load font texture and render glyphs
2 VkImage fontTexture;
3 VkSampler fontSampler;
4 // Code to create and use the font texture
```

Scalable Layouts

Support screen resolution independence by implementing scalable UI layouts using relative coordinates or scaling factors.


```
1 element.position *= scaleFactor;
2 element.size *= scaleFactor;
```

Animations

Add smooth animations for hover effects or transitions by interpolating between states.

```
1 // Simple hover animation
2 if (element.isHovered) {
3     element.color = glm::mix(element.color, hoverColor, animationSpeed);
4 }
```

94.7 Performance Considerations

Rendering multiple UI elements can impact performance. Optimize the UI library with:

- **Batch Rendering:** Group draw calls to minimize pipeline changes.
- **Dynamic Buffers:** Use Vulkan's dynamic uniform buffers to update per-frame data.
- **Efficient Input Handling:** Avoid redundant state checks for non-interactive elements.

94.8 Summary

Building a Vulkan-based UI library provides a deeper understanding of 2D rendering techniques and Vulkan's flexibility. The concepts explored here—such as vertex buffers, input handling, and scalable layouts—form the foundation of a customizable and interactive UI system. Expand this basic library into a full-fledged solution tailored to your application needs.

95

Create a Vulkan Music Visualizer: Sync graphics with music

95.1 Introduction

Music visualizers are a popular and engaging way to combine audio and graphics. By syncing graphics to audio, you can create dynamic and captivating visual effects that react in real time to the music's beat, frequency, and amplitude. In this chapter, you will learn how to build a Vulkan-based music visualizer by analyzing audio input and using it to influence rendered graphics.

95.2 Overview of the Music Visualizer

The music visualizer will:

- **Analyze Audio Data:** Extract amplitude and frequency information from a music track.
- **Sync Graphics to Music:** Use audio data to drive the transformation, color, or position of graphical elements.
- **Render Real-Time Effects:** Display fluid animations and effects based on music dynamics.

95.3 Audio Analysis

Using an Audio Processing Library

To analyze the music data, use an audio library such as **FFmpeg** or **PortAudio** to read and process audio files. These libraries can provide raw audio samples or frequency domain data using techniques like the Fast Fourier Transform (FFT).

Extracting Frequency Data

Use FFT to split the audio into different frequency bands (e.g., bass, midrange, treble). This data will influence specific visual effects.

```

1 // Example: Using FFTW to process audio samples
2 #include <fftw3.h>
3
4 void processAudio(const std::vector<float>& audioSamples, std::vector<float>& frequencyBands) {
5     fftw_complex* in = fftw_alloc_complex(audioSamples.size());
6     fftw_complex* out = fftw_alloc_complex(audioSamples.size());
7
8     // Copy audio data to FFT input
9     for (size_t i = 0; i < audioSamples.size(); ++i) {
10         in[i][0] = audioSamples[i]; // Real part
11         in[i][1] = 0.0;           // Imaginary part
12     }
13
14     // Perform FFT
15     fftw_plan plan = fftw_plan_dft_1d(audioSamples.size(), in, out, FFTW_FORWARD, FFTW_ESTIMATE);
16     fftw_execute(plan);
17
18     // Extract frequency bands
19     for (size_t i = 0; i < frequencyBands.size(); ++i) {
20         frequencyBands[i] = sqrt(out[i][0] * out[i][0] + out[i][1] * out[i][1]); // Magnitude
21     }
22
23     fftw_destroy_plan(plan);
24     fftw_free(in);
25     fftw_free(out);
26 }
```

95.4 Integrating Audio with Vulkan Graphics

Passing Audio Data to the GPU

Use a uniform buffer or push constants to send audio data (e.g., frequency bands) to the Vulkan shaders.

```

1 // Uniform buffer to hold frequency data
2 VkBuffer audioBuffer;
3 VkDeviceMemory audioBufferMemory;
4
5 // Map and update audio buffer with frequency data
6 void updateAudioBuffer(const std::vector<float>& frequencyBands) {
7     void* data;
8     vkMapMemory(device, audioBufferMemory, 0, bufferSize, 0, &data);
9     memcpy(data, frequencyBands.data(), frequencyBands.size() * sizeof(float));
10 }
```

```

10 vkUnmapMemory(device, audioBufferMemory);
11 }

```

Updating Shaders

Modify the vertex or fragment shaders to use the audio data. For example, use the bass frequency to scale objects and the treble frequency to change colors.

```

1 // Vertex shader example
2 layout(set = 0, binding = 0) uniform AudioData {
3     float frequencyBands[64];
4 };
5
6 layout(location = 0) in vec3 inPosition;
7 layout(location = 1) out vec3 fragColor;
8
9 void main() {
10     float scale = 1.0 + frequencyBands[0] * 0.1; // Use bass for scaling
11     gl_Position = vec4(inPosition * scale, 1.0);
12
13     fragColor = vec3(frequencyBands[10], frequencyBands[20], frequencyBands[30]); // Use mid/treble for color
14 }

```

95.5 Rendering Visual Effects

Simple Visualizer: Pulsating Circles

Render a series of circles or spheres that pulse with the beat of the music.

1. Generate a circle using vertex data for a triangle fan.
2. Scale the circle's size based on the amplitude of a specific frequency band.

```

1 // Update circle size based on bass frequency
2 float bassAmplitude = frequencyBands[0];
3 float circleSize = 1.0f + bassAmplitude * 0.5f;
4
5 // Update uniform data
6 updateAudioBuffer(frequencyBands);

```

Advanced Visualizer: Frequency Bars

Render vertical bars for each frequency band, scaling their height dynamically.

```

1 // Generate bar positions
2 for (size_t i = 0; i < frequencyBands.size(); ++i) {
3     float barHeight = frequencyBands[i] * scaleFactor;
4     vertices.push_back(glm::vec2(i * barWidth, 0.0f), color);
5     vertices.push_back(glm::vec2(i * barWidth, barHeight), color);
6 }

```

95.6 Performance Considerations

Real-time audio visualization can be resource-intensive. Optimize your implementation by:

- **Minimizing Data Transfers:** Send only necessary frequency data to the GPU.
- **Using Compute Shaders:** Perform audio processing directly on the GPU for higher efficiency.
- **Batching Draw Calls:** Render all visualizer elements in a single pass.

95.7 Expanding the Visualizer

Add Particle Effects

Enhance the visualizer with particle effects that react to the beat or treble frequencies.

```
1 // Emit particles based on treble frequency
2 if (frequencyBands[20] > threshold) {
3     emitParticles(position, color);
4 }
```

Synchronize with Playback

Use a media library like SDL2 to synchronize audio playback with rendering, ensuring that visual effects match the music in real time.

95.8 Summary

Creating a Vulkan-based music visualizer is a rewarding project that combines real-time audio analysis and dynamic graphics rendering. This chapter introduced the fundamental concepts, including audio processing, data synchronization, and interactive visual effects. Experiment with different visual styles, such as waveforms, spectrograms, and particle systems, to create a unique and immersive experience.

96

Simulate Explosions: Render dynamic explosion effects

96.1 Introduction

Explosion effects are a staple of many visual simulations, games, and animations. They are visually striking and provide a sense of impact and energy in a scene. In this chapter, we will explore how to create dynamic explosion effects in Vulkan by combining particle systems, procedural shaders, and compute-based simulation. This project will challenge you to integrate various Vulkan techniques, such as compute shaders, dynamic vertex buffers, and fragment effects, to simulate and render explosions in real time.

96.2 Overview of Explosion Effects

An explosion effect typically involves:

- **Particle Emission:** Generating multiple particles to simulate debris, smoke, and fire.
- **Dynamic Motion:** Applying physics to particles, such as velocity, acceleration, and damping.
- **Visual Effects:** Using shaders to simulate fire, smoke, and glowing debris.
- **Fade-Out and Dissipation:** Gradually reducing particle visibility to simulate the explosion dissipating over time.

96.3 Creating a Particle System for Explosions

Particle Initialization

Each particle in the explosion will be represented by attributes such as position, velocity, color, and lifetime. The particles will be initialized with random directions and velocities to simulate an outward burst.

```

1 // Particle structure
2 struct Particle {
3     glm::vec3 position;
4     glm::vec3 velocity;
5     glm::vec4 color;
6     float lifetime;
7 };
8
9 // Initialize particles
10 void initializeParticles(std::vector<Particle>& particles, int count) {
11     for (int i = 0; i < count; ++i) {
12         Particle particle;
13         particle.position = glm::vec3(0.0f); // Origin
14         particle.velocity = glm::sphericalRand(1.0f) * explosionSpeed; // Random direction
15         particle.color = glm::vec4(1.0, 0.5, 0.0, 1.0); // Bright orange
16         particle.lifetime = initialLifetime; // Time to live
17         particles.push_back(particle);
18     }
19 }

```

Dynamic Vertex Buffer

Store particle data in a Vulkan vertex buffer, and update it each frame based on the simulation.

```

1 // Create a Vulkan buffer for particle data
2 VkBuffer particleBuffer;
3 VkDeviceMemory particleBufferMemory;
4
5 // Update particle buffer with new positions
6 void updateParticleBuffer(const std::vector<Particle>& particles) {
7     void* data;
8     vkMapMemory(device, particleBufferMemory, 0, particles.size() * sizeof(Particle), 0, &data);
9     memcpy(data, particles.data(), particles.size() * sizeof(Particle));
10    vkUnmapMemory(device, particleBufferMemory);
11 }

```

96.4 Simulating Particle Dynamics

Compute Shader for Physics

Use a compute shader to update particle attributes such as position and velocity. This offloads the simulation from the CPU to the GPU.

```

1 // Compute shader to update particle positions
2 layout(set = 0, binding = 0) buffer ParticleData {
3     struct {
4         vec3 position;

```

```

5     vec3 velocity;
6     vec4 color;
7     float lifetime;
8 } particles[];
9 };
10
11 layout(push_constant) uniform SimulationParams {
12     float deltaTime;
13     vec3 gravity;
14 } params;
15
16 void main() {
17     uint id = gl_GlobalInvocationID.x;
18     particles[id].velocity += params.gravity * params.deltaTime; // Apply gravity
19     particles[id].position += particles[id].velocity * params.deltaTime; // Update position
20     particles[id].lifetime -= params.deltaTime; // Decrease lifetime
21
22     // Fade out particles based on remaining lifetime
23     particles[id].color.a = max(0.0, particles[id].lifetime / initialLifetime);
24 }

```

Dispatching Compute Work

Launch the compute shader with a workgroup size matching the particle count.

```

1 // Dispatch compute shader
2 vkCmdDispatch(commandBuffer, particleCount / workgroupSize, 1, 1);

```

96.5 Rendering Explosion Effects

Particle Rendering

Render particles as point sprites or billboards, applying a fire-like texture to simulate the explosion visually.

```

1 // Fragment shader for fire effect
2 layout(location = 0) in vec4 particleColor;
3 layout(location = 0) out vec4 outColor;
4
5 void main() {
6     // Simulate glowing fire effect
7     float intensity = 1.0 - length(gl_PointCoord - vec2(0.5));
8     outColor = particleColor * intensity;
9 }

```

Blending and Transparency

Enable blending in Vulkan to achieve the smooth, glowing appearance of explosions.

```

1 // Configure blend state
2 VkPipelineColorBlendAttachmentState blendAttachment = 0;
3 blendAttachment.blendEnable = VK_TRUE;
4 blendAttachment.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;
5 blendAttachment.dstColorBlendFactor = VK_BLEND_FACTOR_ONE;
6 blendAttachment.colorBlendOp = VK_BLEND_OP_ADD;
7 blendAttachment.srcAlphaBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA;

```

```
8 blendAttachment.dstAlphaBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA;  
9 blendAttachment.alphaBlendOp = VK_BLEND_OP_ADD;
```

96.6 Expanding the Effect

Smoke Trails

Add secondary particles for smoke trails that emerge from the explosion. Use a gray, semi-transparent texture and slower motion for these particles.

Lighting and Shadows

Enhance realism by adding dynamic lighting that brightens the scene during the explosion. Use a light source that follows the explosion's origin and fades over time.

Multiple Explosions

Create multiple explosions by spawning particles at different locations. Use a timer or user interaction to trigger new explosions dynamically.

96.7 Performance Optimization

Simulating and rendering hundreds or thousands of particles can be computationally expensive. Optimize by:

- Using compute shaders for efficient parallel updates.
- Batching particle rendering into a single draw call.
- Using Vulkan's pipeline cache to reduce state change overhead.

96.8 Summary

Simulating explosions in Vulkan involves a combination of particle systems, compute shaders, and advanced rendering techniques. This chapter has shown how to create dynamic, visually compelling explosions while optimizing performance. With these skills, you can extend the project to include more complex visual effects, such as debris, shockwaves, and post-processing effects.

97

Add Vulkan Networking: Sync data across networks for multiplayer apps

97.1 Introduction

Vulkan is a powerful API for rendering, but when developing multiplayer applications or distributed simulations, you need networking to synchronize data across different clients or devices. While Vulkan itself does not provide networking capabilities, it can be combined with networking libraries like **Enet**, **Boost.Asio**, or even **WebSockets** to synchronize rendering-related data, such as player positions, game states, or shared 3D scenes.

This chapter explores integrating networking with Vulkan to create a multiplayer-capable application. We'll cover the architecture for syncing data across networks and how to ensure smooth rendering without performance bottlenecks.

97.2 Networking Basics

Networking involves sending and receiving data packets over a communication channel. For multiplayer Vulkan apps, you typically:

1. **Establish a Connection:** Use a networking library to connect clients and servers.
2. **Sync State:** Transfer state information, such as object positions or actions.
3. **Handle Latency:** Implement interpolation or prediction to smooth out delays.
4. **Secure Communication:** Ensure data integrity and security if needed.

97.3 Choosing a Networking Library

Several libraries can handle networking for Vulkan applications:

- **Enet**: Lightweight, UDP-based networking library for low-latency applications.
- **Boost.Asio**: Powerful library for synchronous and asynchronous I/O.
- **WebSockets**: Ideal for browser-based Vulkan apps or cross-platform multiplayer.

For this chapter, we'll use **Enet**, which is efficient and easy to integrate.

97.4 Setting Up the Networking Layer

Initialize Enet

Begin by initializing the Enet library and creating a host.

```

1 // Initialize Enet
2 if (enet_initialize() != 0) {
3     throw std::runtime_error("Failed to initialize Enet!");
4 }
5 atexit(enet_deinitialize);
6
7 // Create a server or client host
8 ENetAddress address;
9 ENetHost* host;
10
11 address.host = ENET_HOST_ANY;
12 address.port = 12345;
13
14 // Create the server
15 host = enet_host_create(&address, /* max clients */ 32, /* channels */ 2, 0, 0);
16 if (!host) {
17     throw std::runtime_error("Failed to create Enet server!");
18 }

```

Handle Networking Events

Listen for incoming packets or events and process them accordingly.

```

1 // Poll for networking events
2 ENetEvent event;
3 while (enet_host_service(host, &event, 1000) > 0) {
4     switch (event.type) {
5         case ENET_EVENT_TYPE_CONNECT:
6             printf("A new client connected.\n");
7             break;
8         case ENET_EVENT_TYPE_RECEIVE:
9             printf("Received packet: %s\n", event.packet->data);
10            enet_packet_destroy(event.packet);
11            break;
12         case ENET_EVENT_TYPE_DISCONNECT:
13             printf("A client disconnected.\n");
14             break;
15     }
16 }

```

97.5 Synchronizing Vulkan Data

Structuring Data for Network Sync

Create a lightweight structure to represent the game or simulation state. This might include positions, orientations, or actions.

```
1 // Structure for syncing player data
2 struct PlayerState {
3     float position[3];
4     float orientation[3];
5     int id;
6 };
```

Sending and Receiving Data

Serialize the data and send it across the network. Use Enet's packet creation functions.

```
1 // Send data to all clients
2 PlayerState state = { {1.0f, 2.0f, 3.0f}, {0.0f, 1.0f, 0.0f}, 42 };
3 ENetPacket* packet = enet_packet_create(&state, sizeof(state), ENET_PACKET_FLAG_RELIABLE);
4 enet_host_broadcast(host, 0, packet);
```

On the receiving side, decode the data and update the Vulkan scene.

```
1 // Receive player state
2 PlayerState* receivedState = (PlayerState*)event.packet->data;
3 updateVulkanScene(receivedState);
```

Updating Vulkan Scene with Synced Data

In your Vulkan rendering loop, use the received data to update object positions or states.

```
1 // Update object position in Vulkan
2 void updateVulkanScene(PlayerState* state) {
3     glm::vec3 position = glm::make_vec3(state->position);
4     updateObjectPosition(objectId, position);
5 }
```

97.6 Handling Latency

Networking introduces latency, which can cause noticeable delays or stuttering. To mitigate this:

1. **Interpolate Positions:** Smooth transitions between the last known and current positions.
2. **Predict Movement:** Extrapolate object positions based on velocity and recent state.
3. **Timestamp Packets:** Use timestamps to synchronize updates accurately.

97.7 Debugging and Optimization

- **Packet Size:** Keep packets as small as possible to reduce latency.
- **Compression:** Compress large state data if bandwidth is a concern.
- **Threading:** Run the networking layer in a separate thread to avoid blocking the rendering loop.

97.8 Summary

Integrating Vulkan with a networking library, you can create powerful multiplayer applications or distributed simulations. This chapter has demonstrated how to set up a networking layer, sync Vulkan data, and manage latency. The techniques covered here can be extended to more complex scenarios, such as real-time strategy games, collaborative design tools, or virtual reality experiences.

98

Render Vulkan-Based Fur Simulation: Create realistic furry objects

98.1 Introduction

Fur simulation is a visually appealing effect used in games and animation to render realistic furry objects, such as animals or textiles. With Vulkan's efficient rendering capabilities and compute power, it is possible to simulate and render fur with high performance. This chapter walks you through the process of creating a basic fur simulation using geometry instancing and compute shaders to model and render thousands of fur strands.

98.2 Overview of Fur Simulation

Rendering fur involves two main components:

1. **Fur Generation:** Simulating the strands of fur or hair based on a base object's geometry.
2. **Rendering Fur Layers:** Rendering multiple layers of fur to achieve the appearance of thickness and softness.

Each fur strand can be represented as a series of connected points or a small set of triangles, and these are distributed across the surface of the base object.

98.3 Step 1: Fur Geometry Generation

Creating Fur Shells

A common approach to fur simulation is the **shell-based method**, where multiple layers of the base geometry are offset along the surface normal to create the illusion of volume.

1. Start with the base geometry (e.g., a sphere or cube).
2. Generate multiple “shells” by duplicating the base geometry and offsetting the vertices outward along their normals.

```

1 // Generate fur shells by duplicating and offsetting the base mesh
2 std::vector<Vertex> generateFurShells(const std::vector<Vertex>& baseMesh, int layers, float offset) {
3     std::vector<Vertex> furShells;
4     for (int i = 0; i < layers; ++i) {
5         float layerOffset = i * offset / layers;
6         for (const Vertex& vertex : baseMesh) {
7             Vertex furVertex = vertex;
8             furVertex.position += vertex.normal * layerOffset;
9             furShells.push_back(furVertex);
10        }
11    }
12    return furShells;
13 }

```

Using a Compute Shader for Strand Dynamics

A compute shader can simulate the dynamics of fur strands, such as bending and movement due to wind or interaction. Each strand can be represented as a set of connected points or a simple curve.

```

1 // Fur dynamics compute shader
2 #version 450
3
4 layout(local_size_x = 256) in;
5
6 struct Strand {
7     vec3 position;
8     vec3 velocity;
9 };
10
11 layout(std430, binding = 0) buffer FurStrands {
12     Strand strands[];
13 };
14
15 void main() {
16     uint idx = gl_GlobalInvocationID.x;
17     strands[idx].velocity += vec3(0.0, -0.01, 0.0); // Gravity effect
18     strands[idx].position += strands[idx].velocity; // Update position
19 }

```

98.4 Step 2: Rendering Fur Layers

Fur layers are rendered using instanced rendering to draw multiple shells efficiently. Each instance corresponds to a fur layer.

```

1 // Create instance buffers for fur layers
2 VkBuffer instanceBuffer;
3 // Populate instance data (transformations, layer-specific offsets)
4 std::vector<InstanceData> instanceData = generateInstanceData(furShells);
5
6 // Record draw commands
7 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, furPipeline);
8 vkCmdBindVertexBuffers(commandBuffer, 0, 1, &vertexBuffer, offsets);
9 vkCmdBindVertexBuffers(commandBuffer, 1, 1, &instanceBuffer, offsets);
10 vkCmdDraw(commandBuffer, vertexCount, instanceCount, 0, 0);

```

Texturing the Fur

To enhance realism, use a fur texture or procedural noise to vary the color and density across the fur. A texture map can also control the fur length.

```

1 // Fragment shader for fur rendering
2 layout(location = 0) in vec3 FragPosition;
3 layout(location = 1) in float LayerIndex;
4
5 layout(binding = 1) uniform sampler2D FurTexture;
6
7 layout(location = 0) out vec4 FragColor;
8
9 void main() {
10     float furDensity = texture(FurTexture, FragPosition.xy).r;
11     if (furDensity < LayerIndex / maxLayers) discard;
12     FragColor = vec4(1.0, 0.8, 0.6, 1.0); // Fur color
13 }

```

98.5 Step 3: Adding Dynamics and Interactivity

Animating Fur with Physics

Integrate physics into the fur strands using a compute shader, applying forces such as wind or user interaction.

```

1 // Wind effect in compute shader
2 vec3 windForce = vec3(sin(time), cos(time), 0.0) * 0.05;
3 strands[idx].velocity += windForce;

```

Interactive Fur Simulation

Allow user interaction by detecting collisions or applying forces to fur strands based on mouse or touch input. Pass interaction data to the compute shader as uniform variables.

98.6 Optimizing Performance

- **Level of Detail (LOD):** Reduce the number of fur layers or strands for distant objects.
- **Batching:** Use instanced rendering to minimize draw calls.
- **Compute Optimization:** Optimize compute shaders by minimizing memory accesses and using shared memory.

98.7 Summary

Fur simulation adds a layer of realism to your Vulkan applications, making characters and objects visually dynamic and engaging. By combining geometry generation, compute shaders for dynamics, and efficient rendering techniques, you can create visually stunning furry objects. Experiment with textures, lighting, and physics to further enhance the effect.

DRAFT
version
0.681

99

Integrate Vulkan with WebAssembly: Run Vulkan in a web browser (experimental)

99.1 Introduction

Running Vulkan applications in a web browser is a challenging yet rewarding endeavor that can expand the accessibility of your graphics applications. By integrating Vulkan with WebAssembly (WASM), you can harness the power of low-level graphics programming on the web, enabling high-performance rendering in browser environments. This chapter provides a step-by-step guide to compiling Vulkan-based applications to WebAssembly and running them in a web browser.

99.2 Overview of WebAssembly

WebAssembly is a binary instruction format designed for web performance. It allows you to compile applications written in C, C++, or Rust into a format that can run on the web with near-native speed. To use Vulkan with WebAssembly, you must rely on libraries like `emscripten` and a Vulkan-compatible web implementation such as `WebGPU` or `MoltenVK` (for platforms that map Vulkan calls to Metal).

99.3 Step 1: Set Up the Development Environment

To compile Vulkan applications for WebAssembly, ensure your development environment includes the following:

- **Emscripten SDK**: A toolchain for compiling C/C++ to WebAssembly.
- **Vulkan SDK**: For Vulkan development and headers.
- **HTML and JavaScript**: For embedding the WebAssembly module and rendering in the browser.

Install Emscripten by following its official setup guide:

```
1 # Install and activate Emscripten
2 git clone https://github.com/emscripten-core/emsdk.git
3 cd emsdk
4 ./emsdk install latest
5 ./emsdk activate latest
6 source ./emsdk_env.sh
```

99.4 Step 2: Adjust Your Vulkan Application for WebAssembly

Ensure Portability

Vulkan applications often rely on platform-specific features such as windowing systems. Replace or abstract these features to use WebAssembly-compatible libraries, such as `SDL2` or `GLFW`, for creating a window and handling input.

Use Vulkan Subsets or Extensions

The Vulkan implementation for WebAssembly might only support a subset of Vulkan features. For instance, consider using `WebGPU` as a target since it provides a GPU abstraction layer optimized for browsers. Alternatively, leverage `MoltenVK` for Vulkan-to-Metal translation.

99.5 Step 3: Compile the Application to WebAssembly

Modify your Vulkan application to use Emscripten's toolchain for compilation. The following example demonstrates a basic Vulkan application's compilation:

```
1 emcc -O3 -s WASM=1 -s USE_WEBGPU=1 -o vulkan_app.html \
2     -I/path/to/vulkan/include -L/path/to/vulkan/lib \
3     vulkan_app.cpp -lvulkan
```

Key flags include:

- `-s WASM=1` : Enable WebAssembly output.
- `-s USE_WEBGPU=1` : Use the WebGPU backend for Vulkan.
- `-o vulkan_app.html` : Specify the HTML wrapper for the output.

99.6 Step 4: Create the HTML Wrapper

The compiled WebAssembly module requires an HTML file for rendering. This file includes JavaScript to load the module and a canvas for displaying Vulkan output:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Vulkan WebAssembly</title>
5 </head>
6 <body>
7   <canvas id="vulkanCanvas" width="800" height="600"></canvas>
8   <script>
9     const canvas = document.getElementById('vulkanCanvas');
10    const Module = {
11      canvas: canvas
12    };
13  </script>
14  <script src="vulkan_app.js"></script>
15 </body>
16 </html>

```

99.7 Step 5: Run and Test in a Browser

Serve the compiled files using a local web server to avoid browser security restrictions:

```
1 python3 -m http.server
```

Navigate to `http://localhost:8000/vulkan_app.html` in your browser to view the Vulkan-rendered content.

99.8 Optimizing for WebAssembly

- **Minimize File Size:** Use Emscripten's optimization flags (`-O3` , `-s ENVIRONMENT=web`).
- **Asynchronous Compilation:** Enable asynchronous WebAssembly compilation for better performance.
- **Memory Management:** Configure Emscripten's memory allocation to match your application's needs.

99.9 Summary

Integrating Vulkan with WebAssembly extends the reach of your applications to web browsers, enabling high-performance graphics on a universal platform. While challenging, this approach opens doors to new audiences and interactive experiences. Experiment with optimization techniques, such as leveraging WebGPU's API and fine-tuning Emscripten settings, to achieve the best results.

100

Simulate Cloth Physics: Use compute shaders for cloth simulation

100.1 Introduction

Cloth simulation is a fascinating and visually engaging application of physics in computer graphics. By using Vulkan compute shaders, you can efficiently simulate the complex interactions of particles to emulate the realistic motion of cloth. This chapter guides you through creating a cloth simulation pipeline, focusing on leveraging compute shaders to handle physics calculations in parallel on the GPU.

100.2 Understanding Cloth Simulation

Cloth simulation involves representing a piece of cloth as a grid of interconnected particles, each of which interacts with its neighbors. The system is governed by physics constraints such as:

- **Structural Constraints:** Maintain the cloth's grid structure by keeping particles connected.
- **Shear Constraints:** Preserve the diagonal relationships in the grid to prevent distortion.
- **Bend Constraints:** Ensure smoothness by limiting how much the grid can bend.

Gravity, wind, and collisions with external objects add further complexity to the simulation.

100.3 Step 1: Define the Particle Grid

The cloth is represented as a grid of particles connected by springs. Each particle's position, velocity, and mass are stored in a buffer, which will be processed in the compute shader.

```

1  // Define particle structure
2  struct Particle {
3      glm::vec3 position;
4      glm::vec3 velocity;
5      float mass;
6  };
7
8  // Initialize grid
9  std::vector<Particle> particles(gridWidth * gridHeight);
10 // Set positions and initialize velocities
11 for (int y = 0; y < gridHeight; ++y) {
12     for (int x = 0; x < gridWidth; ++x) {
13         int index = y * gridWidth + x;
14         particles[index].position = glm::vec3(x * spacing, -y * spacing, 0.0f);
15         particles[index].velocity = glm::vec3(0.0f);
16         particles[index].mass = 1.0f; // Uniform mass for simplicity
17     }
18 }

```

Upload this data to a Vulkan buffer, ensuring it is accessible to the compute shader.

100.4 Step 2: Write the Compute Shader

The compute shader processes each particle independently, applying forces and resolving constraints in parallel. A simplified example of a compute shader might look like this:

```

1  // Compute shader for cloth physics
2  #version 450
3  layout(local_size_x=16, local_size_y=16) in;
4
5  struct Particle {
6      vec3 position;
7      vec3 velocity;
8      float mass;
9  };
10
11 layout(std430, binding=0) buffer Particles {
12     Particle particles[];
13 };
14
15 uniform float deltaTime;
16 uniform vec3 gravity;
17
18 void main() {
19     uint id = gl_GlobalInvocationID.x + gl_GlobalInvocationID.y * gridWidth;
20
21     // Apply gravity
22     particles[id].velocity += gravity * deltaTime;
23
24     // Update position
25     particles[id].position += particles[id].velocity * deltaTime;
26
27     // TODO: Implement constraint resolution and collision handling
28 }

```

100.5 Step 3: Handle Constraints

To simulate realistic cloth behavior, constraints must be resolved iteratively. For each particle, ensure that the distance to its neighbors does not exceed a predefined limit. This requires adjusting the positions of neighboring particles to maintain the cloth's integrity.

In practice, constraints are resolved by repeatedly calculating adjustments for all particle pairs connected by springs.

100.6 Step 4: Integrate with Vulkan Pipeline

After setting up the compute shader, integrate it into your Vulkan pipeline. Allocate and bind the necessary buffers for particle data, and dispatch the compute shader with a grid size matching the cloth's dimensions.

```
1 // Bind resources
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, computePipeline);
3 vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_COMPUTE, pipelineLayout, 0, 1, &descriptorSet, 0,
  ↳ nullptr);
4
5 // Dispatch compute work
6 vkCmdDispatch(commandBuffer, gridWidth / 16, gridHeight / 16, 1);
```

100.7 Step 5: Render the Cloth

To visualize the simulated cloth, render the particle positions as a wireframe or textured surface. Use vertex buffers to upload particle positions to the GPU and draw them as triangles or lines.

```
1 // Update vertex buffer with particle positions
2 for (int i = 0; i < particles.size(); ++i) {
3     vertices[i].position = particles[i].position;
4 }
5
6 // Render the cloth
7 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
8 vkCmdDraw(commandBuffer, vertexCount, 1, 0, 0);
```

100.8 Step 6: Add Interactivity and Forces

Enhance the simulation by adding user interactions, such as clicking and dragging the cloth or applying wind forces. Update the compute shader to respond to external forces dynamically.

```
1 // Apply wind force
2 vec3 windForce = vec3(0.0, 0.0, 10.0);
3 particles[id].velocity += windForce * deltaTime;
```


100.9 Summary

Simulating cloth physics with Vulkan compute shaders demonstrates the power and flexibility of GPU computing. Iteratively solving constraints and leveraging parallel processing, you can create visually compelling and physically accurate simulations. Experiment with additional forces, collision detection, and rendering techniques to refine your simulation further.

DRAFT
version
0.681

101

Build a Vulkan-Based Ray Tracing Engine: Fully utilize Vulkan's ray tracing features

101.1 Introduction

Building a ray tracing engine is a significant milestone in computer graphics and one of the most exciting applications of Vulkan's advanced features. Vulkan's support for ray tracing provides a robust foundation for implementing realistic lighting, shadows, reflections, and other effects that enhance visual fidelity. This chapter walks you through the essential steps to create a Vulkan-based ray tracing engine, from acceleration structures to shading and rendering.

101.2 Overview of Vulkan Ray Tracing

Vulkan's ray tracing pipeline extends the traditional graphics pipeline with new programmable stages and constructs:

- **Acceleration Structures:** Hierarchical representations of scene geometry optimized for ray traversal.
- **Ray Generation Shaders:** Initiate ray tracing operations and manage ray logic.
- **Intersection Shaders:** Handle custom intersection tests.
- **Closest Hit and Any Hit Shaders:** Compute shading for rays and manage hit interactions.
- **Miss Shaders:** Define behavior when no geometry is hit.

101.3 Step 1: Initialize Ray Tracing Extensions

To utilize Vulkan's ray tracing capabilities, enable the required extensions and features during instance and device creation.

```

1  // Enable ray tracing extensions
2  const std::vector<const char*> deviceExtensions = {
3      VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
4      VK_KHR_RAY_TRACING_PIPELINE_EXTENSION_NAME,
5      VK_KHR_DEFERRED_HOST_OPERATIONS_EXTENSION_NAME,
6      VK_EXT_DESCRIPTOR_INDEXING_EXTENSION_NAME,
7      VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME,
8      VK_KHR_SPIRV_1_4_EXTENSION_NAME,
9      VK_KHR_SHADER_FLOAT_CONTROLS_EXTENSION_NAME
10 };
11
12 // Specify required device features
13 VkPhysicalDeviceRayTracingPipelineFeaturesKHR rayTracingFeatures{};
14 rayTracingFeatures.sType = VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_RAY_TRACING_PIPELINE_FEATURES_KHR;
15 rayTracingFeatures.rayTracingPipeline = VK_TRUE;
16 rayTracingFeatures.rayTraversalPrimitiveCulling = VK_TRUE;
17
18 deviceCreateInfo.pNext = &rayTracingFeatures;

```

101.4 Step 2: Build Acceleration Structures

Acceleration structures are the backbone of Vulkan's ray tracing engine. They allow efficient ray traversal and intersection queries.

- **Bottom-Level Acceleration Structures (BLAS):** Represent individual geometry such as triangles or meshes.
- **Top-Level Acceleration Structures (TLAS):** Organize instances of BLAS, representing the entire scene.

Creating a BLAS: Allocate a buffer for vertex and index data, then build a BLAS.

```

1  // Define geometry for BLAS
2  VkAccelerationStructureGeometryKHR geometry{};
3  geometry.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
4  geometry.geometryType = VK_GEOMETRY_TYPE_TRIANGLES_KHR;
5  geometry.flags = VK_GEOMETRY_OPAQUE_BIT_KHR;
6  geometry.geometry.triangles.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_TRIANGLES_DATA_KHR;
7  geometry.geometry.triangles.vertexFormat = VK_FORMAT_R32G32B32_SFLOAT;
8  geometry.geometry.triangles.vertexData.deviceAddress = vertexBufferAddress;
9  geometry.geometry.triangles.indexType = VK_INDEX_TYPE_UINT32;
10 geometry.geometry.triangles.indexData.deviceAddress = indexBufferAddress;
11
12 // Build BLAS
13 VkAccelerationStructureBuildGeometryInfoKHR buildInfo{};
14 buildInfo.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
15 buildInfo.type = VK_ACCELERATION_STRUCTURE_TYPE_BOTTOM_LEVEL_KHR;
16 buildInfo.flags = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
17 buildInfo.geometryCount = 1;
18 buildInfo.pGeometries = &geometry;
19
20 VkAccelerationStructureBuildRangeInfoKHR buildRange{};
21 buildRange.primitiveCount = indexCount / 3;
22

```

```

23 VkAccelerationStructureBuildRangeInfoKHR* pBuildRanges[] = { &buildRange };
24 vkCmdBuildAccelerationStructuresKHR(commandBuffer, 1, &buildInfo, pBuildRanges);

```

Creating a TLAS: Use the built BLAS instances to create the TLAS, defining their transformations and instance data.

```

1  // Define instance data
2  VkAccelerationStructureInstanceKHR instance0;
3  instance.transform = glmToVkTransform(matrix);
4  instance.instanceCustomIndex = 0;
5  instance.mask = 0xFF;
6  instance.instanceShaderBindingTableRecordOffset = 0;
7  instance.flags = VK_GEOMETRY_INSTANCE_TRIANGLE_FACING_CULL_DISABLE_BIT_KHR;
8  instance.accelerationStructureReference = blasDeviceAddress;
9
10 // Build TLAS
11 VkAccelerationStructureGeometryInstancesDataKHR instancesData0;
12 instancesData.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_INSTANCES_DATA_KHR;
13 instancesData.data.deviceAddress = instanceBufferAddress;
14
15 VkAccelerationStructureGeometryKHR tlasGeometry0;
16 tlasGeometry.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_GEOMETRY_KHR;
17 tlasGeometry.geometryType = VK_GEOMETRY_TYPE_INSTANCES_KHR;
18 tlasGeometry.geometry.instances = instancesData;
19
20 // TLAS build info
21 VkAccelerationStructureBuildGeometryInfoKHR tlasBuildInfo0;
22 tlasBuildInfo.sType = VK_STRUCTURE_TYPE_ACCELERATION_STRUCTURE_BUILD_GEOMETRY_INFO_KHR;
23 tlasBuildInfo.type = VK_ACCELERATION_STRUCTURE_TYPE_TOP_LEVEL_KHR;
24 tlasBuildInfo.flags = VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_KHR;
25 tlasBuildInfo.geometryCount = 1;
26 tlasBuildInfo.pGeometries = &tlasGeometry;
27
28 // Build TLAS
29 vkCmdBuildAccelerationStructuresKHR(commandBuffer, 1, &tlasBuildInfo, pBuildRanges);

```

101.5 Step 3: Write Ray Tracing Shaders

Ray tracing shaders define how rays interact with the scene. At a minimum, implement the following shader types:

- **Ray Generation Shader:** Spawns rays and initiates traversal.
- **Closest Hit Shader:** Calculates shading at the intersection point.
- **Miss Shader:** Defines behavior for rays that miss all geometry.

```

1  // Ray Generation Shader
2  #version 460
3  #extension GL_EXT_ray_tracing : require
4
5  layout(location = 0) rayPayloadEXT vec3 payload;
6
7  void main() {
8      vec3 origin = vec3(0.0, 0.0, 0.0);
9      vec3 direction = normalize(vec3(gl_LaunchIDEXT.xy, 1.0));
10     traceRayEXT(tlas, RAY_FLAG_NONE, 0xFF, 0, 0, 0, origin, 0.0, direction, 100.0, payload);
11 }
12
13 // Closest Hit Shader
14 #version 460
15 #extension GL_EXT_ray_tracing : require

```

```

16
17 layout(location = 0) rayPayloadInEXT vec3 payload;
18
19 void main() {
20     payload = vec3(1.0, 0.0, 0.0); // Set hit color
21 }
22
23 // Miss Shader
24 #version 460
25 #extension GL_EXT_ray_tracing : require
26
27 layout(location = 0) rayPayloadInEXT vec3 payload;
28
29 void main() {
30     payload = vec3(0.0, 0.0, 1.0); // Set miss color
31 }

```

101.6 Step 4: Build the Ray Tracing Pipeline

Create a ray tracing pipeline by linking the shaders and specifying the Shader Binding Table (SBT).

```

1 // Define ray tracing pipeline stages
2 VkPipelineShaderStageCreateInfo stages[] = {
3     { /* Ray Generation Shader Info */ },
4     { /* Closest Hit Shader Info */ },
5     { /* Miss Shader Info */ },
6 };
7
8 // Create pipeline
9 VkRayTracingPipelineCreateInfoKHR pipelineInfo{};
10 pipelineInfo.sType = VK_STRUCTURE_TYPE_RAY_TRACING_PIPELINE_CREATE_INFO_KHR;
11 pipelineInfo.stageCount = sizeof(stages) / sizeof(stages[0]);
12 pipelineInfo.pStages = stages;
13 pipelineInfo.maxPipelineRayRecursionDepth = 1;
14
15 VkPipeline rayTracingPipeline;
16 vkCreateRayTracingPipelinesKHR(device, nullptr, nullptr, 1, &pipelineInfo, nullptr, &rayTracingPipeline);

```

101.7 Step 5: Dispatch Rays and Render

Bind the ray tracing pipeline and dispatch rays.

```

1 // Bind ray tracing pipeline
2 vkCmdBindPipeline(commandBuffer, VK_PIPELINE_BIND_POINT_RAY_TRACING_KHR, rayTracingPipeline);
3
4 // Trace rays
5 vkCmdTraceRaysKHR(commandBuffer, &raygenRegion, &missRegion, &hitRegion, nullptr, width, height, 1);

```

101.8 Step 6: Add Advanced Features

Enhance the ray tracing engine with advanced features such as:

- **Reflection and Refraction:** Add secondary ray tracing.
- **Soft Shadows:** Implement area lights and shadow sampling.
- **Global Illumination:** Use recursive ray tracing for indirect lighting.

101.9 Summary

Following the steps in this chapter, you'll be able to build a basic ray tracing engine using Vulkan's powerful ray tracing features. Experiment with additional effects and optimizations to push the visual and performance boundaries of your engine further.

DRAFT
version
0.681



Appendix

A.1 Further Reading

Official Documentation

- **Vulkan API**
<https://www.vulkan.org>
- **GLSL Shader Language (GLSL)**
<https://registry.khronos.org/OpenGL/specs/gl/>

Examples/Tutorials

- **XBDev Vulkan/GLSL Tutorials & Information**
<https://xbdev.net/vulkan/>
- **VulkanGPU Lab (100+ Code Samples & Examples)**
<https://vulkanlab.xbdev.net/>

A.2 Vulkan Compute Shader Key Functions/Members

Listing A.1: Vulkan Compute Shader key functions/members.

<code>vkCreateShaderModule,</code>	<code>vkCreatePipelineLayout,</code>	<code>vkCreateComputePipelines,</code>
<code>vkCmdDispatch,</code>	<code>vkCmdDispatchIndirect,</code>	<code>vkCreateBuffer,</code>
<code>vkBindBufferMemory,</code>	<code>vkCreateDescriptorSetLayout,</code>	<code>vkCreateDescriptorPool,</code>
<code>vkAllocateDescriptorSets,</code>	<code>vkUpdateDescriptorSets,</code>	<code>vkCmdBindPipeline,</code>
<code>vkCmdBindDescriptorSets,</code>	<code>vkCmdPushConstants,</code>	<code>vkCmdPipelineBarrier,</code>
<code>vkMapMemory,</code>	<code>vkUnmapMemory,</code>	<code>vkCmdCopyBuffer</code>

A.3 GLSL Compute Shader Reserved Words

A reserved word is a token that is reserved and cannot be used as an identifier in GLSL shaders, including Vulkan compute shaders.

Listing A.2: GLSL reserved words for Vulkan compute shaders.

in	out	uniform	shared	buffer	coherent
volatile	restrict	readonly	writeonly	subgroup	subgroupBarrier
workgroupBarrier	memoryBarrier	atomic_uint	subgroupMemoryBarrier		

A.4 GLSL Compute Shader Keywords

Vulkan compute shaders share many types with OpenGL compute shaders, but these are handled through Vulkan-specific APIs.

Listing A.3: GLSL compute shader keywords in Vulkan.

uint	ivec2	ivec3	ivec4	uvec2	uvec3	uvec4
image1D	image2D	image3D	image2DArray	imageCube	imageBuffer	
samplerBuffer	atomic_uint	buffer	sampler	storageBuffer		

A.5 GLSL Compute Shader Built-in Variables

In Vulkan, GLSL compute shaders have built-in variables to manage thread dispatch, workgroup IDs, and memory synchronization. These variables are used for coordinating the execution of the compute shaders.

Listing A.4: GLSL compute shader built-in variables in Vulkan.

gl_GlobalInvocationID	// Global thread index (uvec3)
gl_LocalInvocationID	// Local index within the workgroup (uvec3)
gl_WorkGroupID	// Workgroup index (uvec3)
gl_LocalInvocationIndex	// Linear index within the workgroup (uint)
gl_NumWorkGroups	// Total number of workgroups (uvec3)
gl_WorkGroupSize	// Size of each workgroup (uvec3)

A.6 GLSL Compute Shader Syntactic Tokens

A list of arithmetic, bitwise, and synchronization operators available in Vulkan compute shaders. These are key to efficient parallel computation and memory synchronization.

Listing A.5: GLSL compute shader syntactic tokens in Vulkan.

+	-	*	/	%	&		^	~
&&		<<	>>	+=	-=	*=	/=	%=
&=	=	^=	<<=	>>=				

A.7 GLSL Compute Shader Built-in Functions

Vulkan compute shaders provide several useful built-in functions for synchronization and atomic operations.

Listing A.6: GLSL compute shader built-in functions in Vulkan.

```
atomicAdd(atomic_uint mem, uint data)    // Atomic add operation
atomicMin(atomic_uint mem, uint data)    // Atomic minimum operation
atomicMax(atomic_uint mem, uint data)    // Atomic maximum operation
barrier()                                // Synchronize workgroup threads
memoryBarrier()                          // Synchronize memory accesses across stages
groupMemoryBarrier()                     // Synchronize memory accesses within a workgroup
subgroupBarrier()                         // Synchronize threads in a subgroup
```

A.8 GLM Library

<https://github.com/g-truc/glm> GLM (OpenGL Mathematics) is a C++ mathematics library specifically designed for graphics software. It provides classes and functions for handling vectors, matrices, and other mathematical operations commonly used in OpenGL and Vulkan graphics programming.

When developing your compute applications, you might want to take advantage of external libraries like GLM for smaller tasks such as matrix transformations, vector math, physics calculations, or even animation systems.



Index

2d, 167, 301
2d game, 301
3d, 45, 305, 309

acceleration structures, 223
advanced features, 213
android, 269
animate, 59
anit-aliasing, 87
app, 269, 273
asynchronous compute, 197
asynchronous graphics, 197

background, 121
batch descriptor, 189
benchmark tools, 399
binding, 51
blender, 363
bloom, 125
blur, 143
bounding volume hierar-
chies (bvh), 223
buffering, 45
buffers, 79
bvh, 223

caching, 177
calls, 359
cellular automata, 151
characters, 309
clear screen, 33
clearing, 33

cloth physics, 427
clouds, 395
color, 33, 37
command buffer, 181
command buffers, 181
commands, 33
compute, 139, 197, 391
compute shader, 135, 171
compute shaders, 139, 143,
147, 151, 155, 159,
313, 427

constants, 63
content, 247
cpu, 79
create, 7, 21
cross-compile, 269, 277
cross-platform, 261
cube map shadows, 117
cubemap, 121
custom ui library, 403

data, 63, 71
data transfer, 79
debug, 17
debugging tools, 373
deferred, 101
dependencies, 97
depth buffer, 45
descriptor set, 51
descriptor sets, 71
descriptor updates, 189

development environment,
281
device extensions, 213
devices, 21
display, 325
draw, 75, 185
draw calls, 185
draw multiple objects, 185
draw quad, 41
draw textured triangle, 55
draw triangle, 37
draw triangles, 75
dynamic buffers, 193
dynamic lighting, 105

effect, 143
effects, 125, 411
efficiency, 181
enable, 17
engine core, 295
enumerate, 13
explosion, 411
export frames, 351
export png, 351
extensions, 213

features, 213
firework display, 325
fireworks, 325
fluid dynamics, 167
font library, 387
fps, 209

- fractals, 139
- frame, 209
- frame graph, 97
- framebuffer, 93
- fur, 419
- fur simulation, 419
- furry objects, 419

- game, 301
- game engine, 295
- game of life, 151
- gaussian blur, 143
- generate, 83, 317
- generated gradient, 83
- glass, 231
- glass-like materials, 231
- glfw, 261
- glm library, 439
- global illumination, 239
- glsl compute shader key-words, 438
- glsl reserved words, 438
- gltf, 305
- gpu, 13, 79, 255
- graph, 97
- graphics, 197
- graphics pipeline, 29
- gui, 321, 367

- handles, 21
- hardware, 399
- hdr, 247
- heightmap, 329
- hello compute, 135
- high-dynamic-range (hdr), 247

- image, 25
- image blur, 143
- images, 347
- imgui, 321
- inference, 355
- instance, 7
- instancing, 185
- integrate, 45, 265, 289
- interactive, 321
- interactive solar system, 337

- layers, 17
- library, 321, 403
- lighting, 105, 239
- lighting effects, 239
- lighting quality, 101
- limit fps, 209
- limiter, 209
- linux, 281
- list, 13
- load models, 305
- logical device, 21

- machine learning (ml), 355
- macos, 273

- manage, 7
- manage data, 71
- mandelbrot set, 139
- materials, 231
- measure, 399
- measure performance, 201
- memory usage, 193
- mesh shader, 251
- mobile gpu, 255
- model, 379
- moltenvk, 273
- msaa, 87
- multiplayer apps, 415
- multiple light, 105
- multiple objects, 71, 75
- multiple passes, 101
- multisample, 87
- music, 407
- music visualizer, 407

- networking, 415
- networks, 415
- next-gen mesh shaders, 251
- noise textures, 155

- obj, 305
- occlusion culling, 391
- off-screen, 93
- omnidirectional shadows, 117
- opencv, 347
- optimize, 391
- optimize buffer, 193
- optimzie, 205
- optimize, 189
- orbits, 337
- overhead, 177

- parallel sorting, 159
- particle system, 129, 163
- particles, 129
- pass data, 67
- path tracing, 235
- path-traced, 235
- pbr, 109
- performance, 201
- perlin noise, 155
- physical device, 13
- physically based rendering (pbr), 109
- physics, 147
- physics engine, 147
- physics simulate, 427
- pipeline, 29, 177
- planetary, 337
- planets, 317
- plugin, 363
- png, 351
- point light, 117
- pong, 301
- post-processing, 125
- present, 25

- procedural clouds, 395
- procedural gradient, 83
- procedural heightmap, 171
- procedural planet, 317
- push constant, 63

- qt, 367
- quad, 41
- quads, 75
- queries, 201
- queue, 21

- rasberry pi, 277
- ray tracing, 219
- ray tracing engine, 431
- ray tracing features, 431
- ray-traced, 227
- realistic clouds, 395
- realistic fur, 419
- realistic lighting, 239
- record command buffers, 181
- reflection, 227
- refractions, 231
- render, 41, 75, 83, 101, 121, 219, 227, 235, 247, 305, 309, 317
- render doc clone, 359
- render passes, 205
- render to texture, 93
- render triangle, 83
- rendering, 109, 113, 255, 333, 379
- resources, 373

- save png frames, 351
- scene, 105, 219, 235, 391
- screen, 33
- screensaver, 341
- sdl2, 265
- seperate queues, 197
- set updates, 189
- sets, 51, 71
- setup, 7, 29
- shader, 67
- shaders, 63, 109, 251
- shadow, 113
- shadow mapping, 113
- shadows, 117
- shadows omnidirectional, 117
- simulate, 129, 167, 239, 313, 325, 337, 411
- simulate physics, 147
- simulation, 151
- single pass, 75
- single triangle, 37
- skeletal animation, 309
- skybox, 121
- smoother models, 383
- solar system, 337
- solid color, 37

sort data, 159
source, 105
staging, 79
staging buffers, 79
subpasses, 205
swapchain, 25
sync data, 415
sync graphics, 407
system, 13, 129

tensorflow, 355
terrain, 329
terrains, 171
tessellation shaders, 383
test, 281
text, 387

texture, 93
texture binding, 51
textured triangle, 55
tile-based, 255
tiled, 255
tiled rendering, 255
time, 201
timestamp, 201
transformations, 67
triangle, 37, 55, 59
triangles, 41, 75

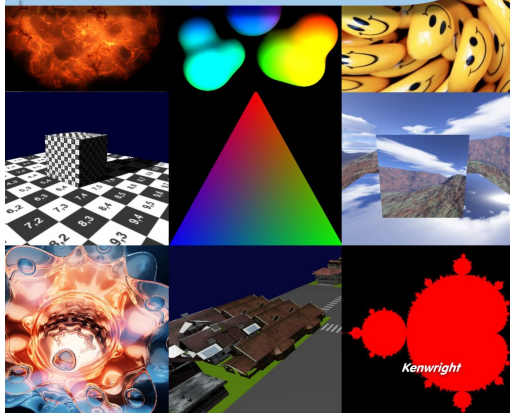
ui library, 403
uniform buffer, 67
unity, 285
unreal engine, 289

validation layers, 17
vertex shader, 129
video decoding, 243
vignette, 125
vr, 333
vulkan, 7

water ripples, 313
water surface, 313
web browser, 423
webassembly, 423
window, 25
windowing, 261, 265
windows, 261
wireframe, 379
write tools, 373

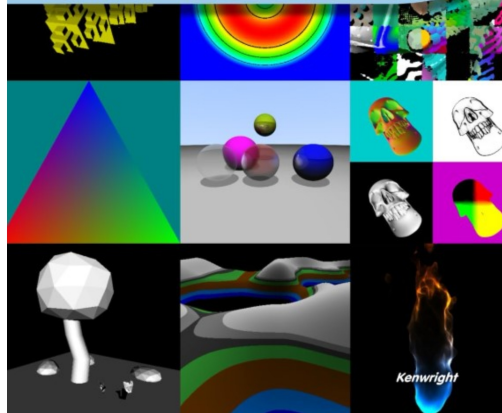
VULKAN Projects

A Hands-On Journey Through 101 Vulkan Programming Examples



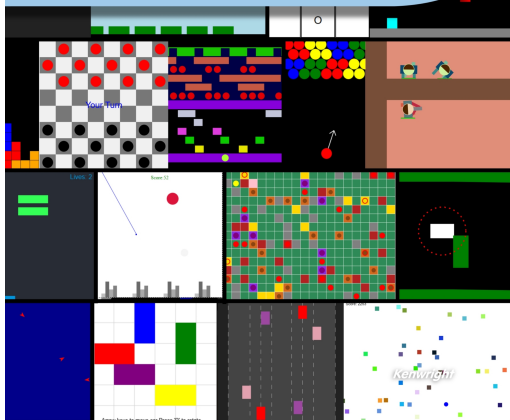
WEBGPU & WGSL Projects

A Hands-On Journey Through 101 WebGPU & WGSL Programming Project Examples



GAMES RETRO ARCADE Projects

A Hands-On Journey Through 101 Game Programming Project Examples



FRACTAL Projects

A Hands-On Journey Through 101 Fractal Programming Project Examples



