EECS 487: Interactive Computer Graphics

Lecture 21:

- Overview of Low-level Graphics API
  Metal, Direct3D 12, Vulkan

Console Games

Why do games look and perform so much better on consoles than on PCs with equivalent specs?

- consoles are closed platforms with long shelf live, programmers can program the hardware directly
- doing away with serialized call-preparation bottleneck allows for better utilization of multiple CPU cores

Motivations for low-level graphics APIs:

- faster graphics from reduced API overhead
- "close-to-metal," direct control of the GPU

[Anandtech:Smith]

Low-Overhead, Low-Level API

Whence the high overhead of graphics API?

- hardware abstractions hide underlying platform diversity, providing programming convenience and flexibility: graphics vs. system programming
- "newby-friendly" safety nets of error checking and state validation

Code gurus (the ones writing game engines and renderers) would rather have performance than hand-holding

[Anandtech:Smith]

Low-Overhead, Low-Level API

Why is this an issue now?

- GPU performance is far outstripping CPU due to the massively parallel nature of graphics rendering: API overhead at the CPU is throttling GPU performance
- serialized command assembly prior to issuing draw calls restricts utilization of multi-core CPU
  - instancing and batching objects into a smaller number of draw calls can only help so much

Another advantage: easier porting of console games to PCs?

[Anandtech:Smith]

How to Improve Performance?

1. Command buffer
   a. reduced draw call overhead
   b. better command submission multi-threading
2. Baked-in states
   a. pipeline state objects
   b. resource binding
3. Pre-compiled shaders

Biggest Source of CPU Overhead

Assembly of command stream prior to issuing a draw call, e.g., the gathering together of

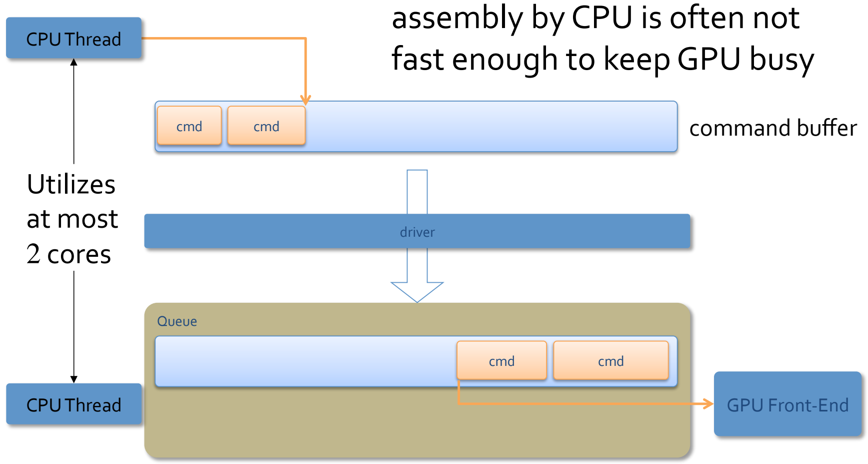
- line mode
- polygon mode
- flat or smooth shading
- texture objects to use
- which vertex array objects
- which vertex buffer objects
- setting vertex attribute pointers
- arguments to draw calls

Done by driver

[Anandtech:Smith]

Single-threaded Job Assembly

Problem: single-threaded job assembly by CPU is often not fast enough to keep GPU busy



[nvidia:Foley]

Command Buffer/List

Developers self-assemble command stream into a command buffer (Vulkan) or command list (D3D12)

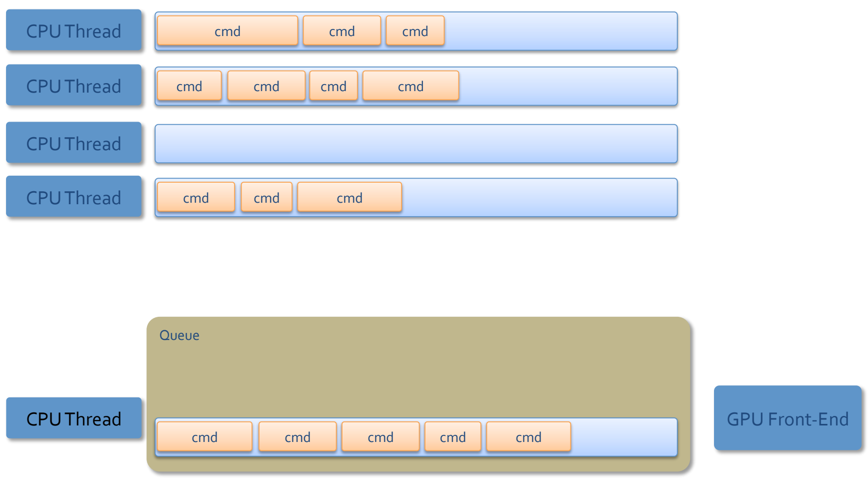
Each command buffer is self-contained, so multiple buffers can be assembled in parallel, each on its own thread/core without extra concurrency work

Final submission of the command buffers via the command queue is still serial, but is highly efficient

Metal	D3D12	Vulkan
MTLCommandBuffer()	ID3D12CommandList()	VkCmdBuffer()
MTLCommandQueue()	ID3D12CommandQueue()	VkCmdQueue()

[Microsoft:Sandy]

Multi-Threaded Job Assembly



[nvidia:Foley]

Command Buffer Re-Use

In Vulkan a command buffer can be re-used

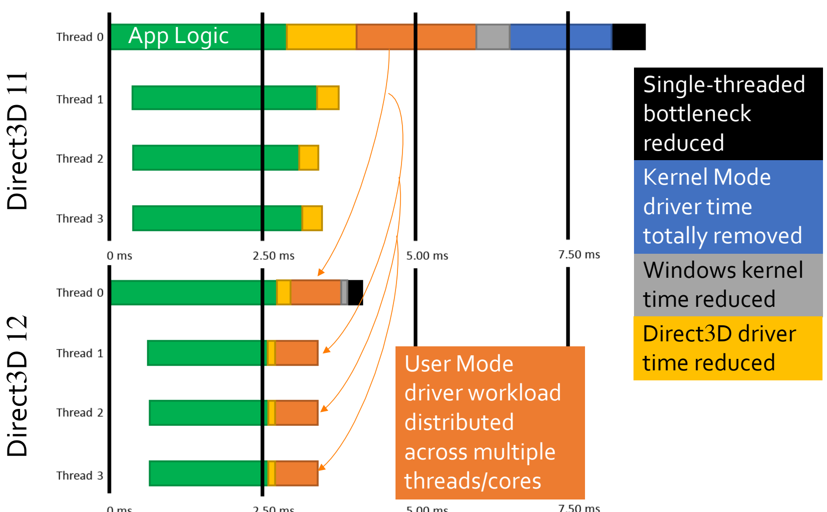
- a "top-level" command buffer can "call" second-level command buffers

In D3D12 a command list "recorded" as a bundle can be submitted once to the GPU but executed multiple times, with different resources, e.g., different textures (much like OpenGL's retained mode display list)

Metal currently doesn't support command buffer re-use

[nvidia:Foley;Microsoft:Sandy]

3DMark – Multi-thread Scaling and 50% Better CPU Utilization

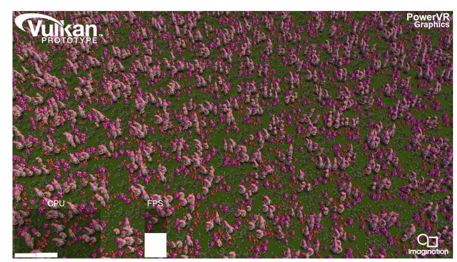


[Microsoft:Sandy; Anandtech:Smith]

Imagination's Gnome Horde

No instancing

Re-use command buffers for each tile:



300 tiles, 13,500 draws/frame, 30 fps, light CPU usage

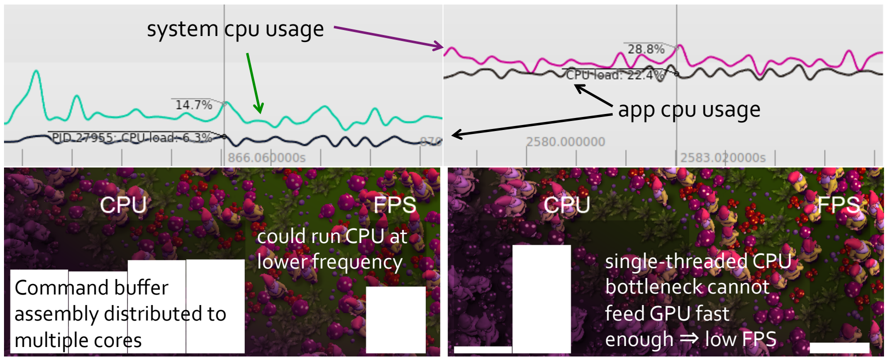
Over 400,000 draw calls/sec, each with a different transformation with many different materials, textures, blend modes, and shaders

[Imagination: Smith]

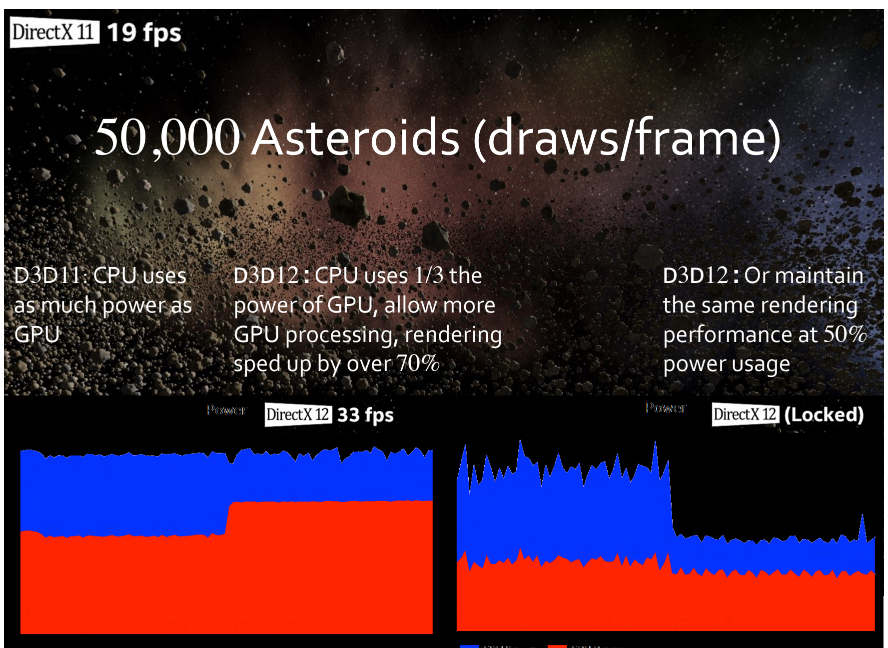
Fast Moving Camera

Command buffers need to be regenerated very frequently

Vulkan | OpenGL ES



[Imagination: Smith]



[Intel:Lauritzen]

Power Efficiency

When CPU and GPU have to share power and thermal budget

- lower CPU usage allows more power and thermal budget to go to GPU
- spreading workload across more CPU cores allow each to run at a lower clock speed, further reducing power usage as compared to running a single thread at a high frequency (to feed the GPU)

[Intel;Lauritzen;Anandtech:Barrett&Smith]

Pipeline State Objects (PSOs)

Problem: draw-time validation of shader states delays hardware setup and reduces the number of draw calls per frame

Solution: bake (compile and validate) pipeline states into PSOs that are finalized on creation, switching PSOs have lower overhead than computing hardware state on the fly

[Microsoft:Sandy]

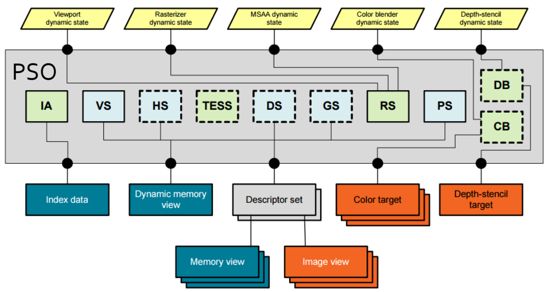
Pipeline State Objects

Contains all static state for entire 3D pipeline

- shaders, vertex attribute formats, rasterization, color blend, depth stencil, etc.

Created outside of the performance critical paths

PSO can be cached for re-use, even saved to disk/ cloud for re-use across app runs

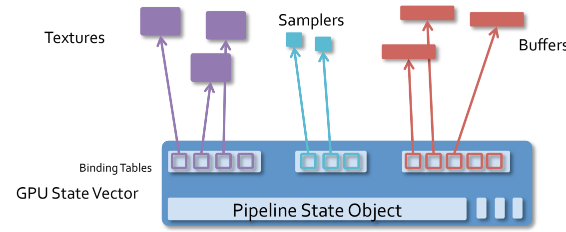


[nvidia:Daniell;AMD]

What Doesn't Go into a PSO?

Resource bindings

- the actual vertex, index, constant buffers
- textures, samplers, etc.

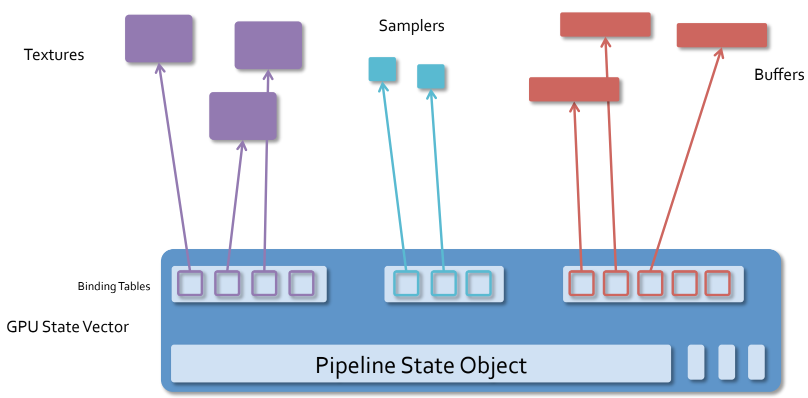


Fixed-function states that do not cause shader recompilation: viewport, color blend constants, polygon offset, scissor, stencil masks and refs, etc.

[nvidia:Foley]

Descriptor Tables and Pool/Heap

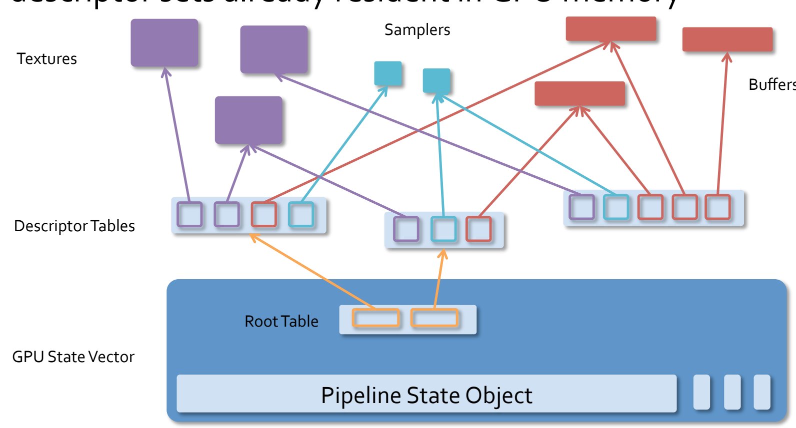
Problem: to use different resources, e.g., texture, an app must bind and rebind them to fixed and limited bind slots (descriptors) and issue multiple draw calls



[Microsoft:Sandy;nvidia:Foley]

Descriptor Table and Heap/Pool

Solution: pre-write multiple sets of descriptors to descriptor heap; changing resources simply switches descriptor sets already resident in GPU memory



[nvidia:Foley;Microsoft:Sandy]

GPU Memory Management

With high-level API, to pass data from app to GPU, first allocate a driver-managed buffer and copy the data before passing the data to the shader $\Rightarrow$ CPU overhead

With low-level API, a developer simply maps the GPU memory address and writes to that memory location directly, no CPU intervention

[Imagination:Smith]

Pre-Compiled Shader

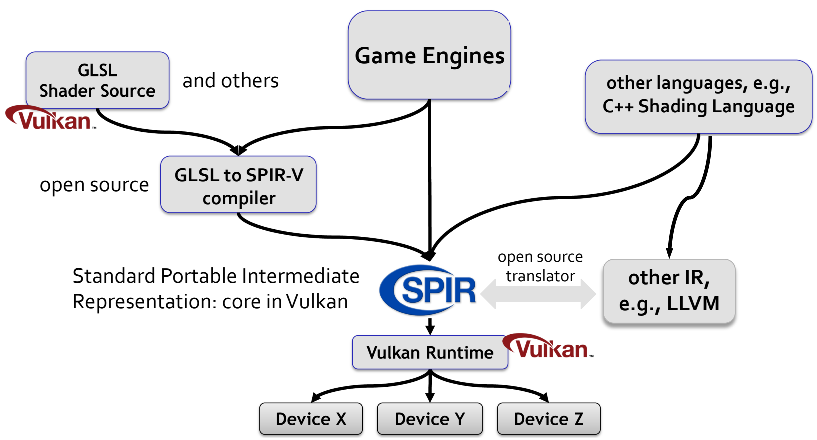
Vulkan:

- pre-compiles shaders into a common intermediate representation
  - provide some IP protection, developers can distribute shaders in a compiled intermediate representation instead of in source
  - pre-compiled shaders also speed up draw calls

Metal also pre-compiles shaders

[Anandtech: Smith]

Vulkan Shader Programming



[Khronos]

More Predictable Performance

Previously: app submits a draw call, maps a buffer, etc.
Driver might (GPU dependent):

- compile shaders
- insert synchronization fences into GPU schedule
- flush caches
- allocate memory

With low-level API all the above must be done by the app itself, but driver performance across vendors becomes more predictable

[nvidia:Foley]

Why Vulkan is Not for Beginners

Must handle multi-threading and concurrency/ synchronization

Must manage memory allocation and usage

These are optional in OpenGL, but mandatory in Vulkan

Summary of Features

Tech	Metal	Direct3D12	Vulkan
command buffer	✔	✔	✔
pipeline state objects	✔	✔	✔
descriptor table	✗	✔	✔
tile-based render pass	✔	✗	✔
multi-adapter	✗	✔	✔?

Vulkan and D3D12:

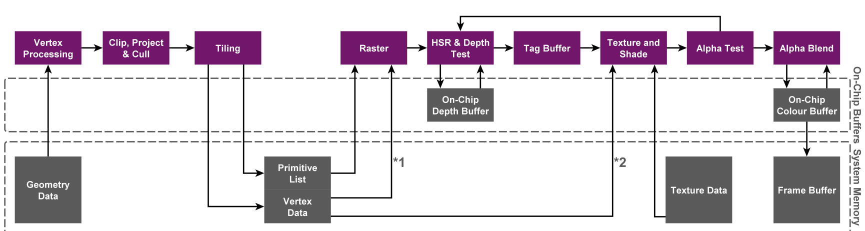
- both similar to Mantle to start with
  - Mantle supports multi-GPU
- not as low level, to be cross-vendor and cross-platform

[nvidia:Foley;Anandtech:Smith]

Tile-based Architectures

"Mobile GPU" usually means "tile-based GPU"

- most Android and all iOS devices use tile-based rendering
  - Vulkan and Metal have support for tile-based architecture, but not Direct3D 12
- tiling reduces use of expensive off-chip memory bandwidth

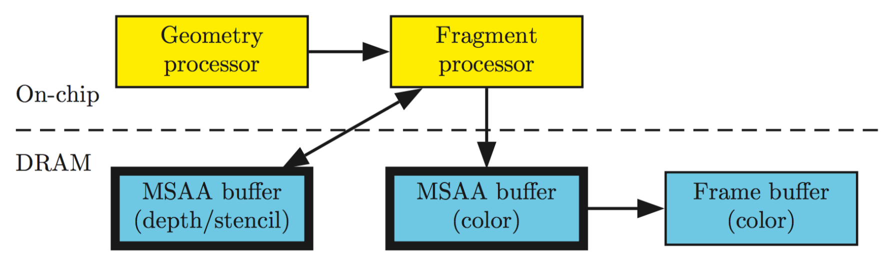


[Google:Hall;Imagination:Sommefeldt]

Immediate-Mode Rendering

Fragment shading, including texture sampling, performed even on fragments that will eventually fail the depth test

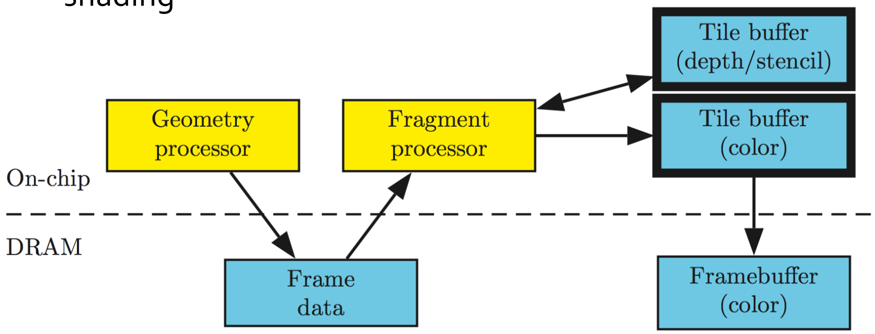
- requires accessing off-chip memory
- inefficient use of off-chip bandwidth



[Imagination:Sommefeldt;Merry]

Tile-based Architectures

Tile-based rendering splits framebuffer up into tiles (e.g., 16×16 or 32×32 pixels) and sort all triangles on tile using on-chip storage before fragment shading



[Merry]

Multi-Adapter Support

PCs can contain multiple graphics cards

Apps can enumerate graphics cards

- can create a device abstraction for each

Some graphics cards have multiple GPUs

- each with its own engines and memory

Apps should be able to assign work to any GPU on any graphics card

- create queues on any engine and submit command buffers
- allocate resources in memory associated with any GPU

[Microsoft:Boyd]

Multi-Adapter Support

Options:

- Alternate-frame rendering (AFR): frame pacing becomes an issue if the GPUs are of different performance
- Split-frame rendering
- Work sharing of individual frames

D3D12 Explicit Multi-Adapter (EMA) mode allows exchange of multiple data types between GPUs, beyond just finished, rendered images

But transferring data over PCIe bus is slow and with high latency!

[Anandtech:Smith]

Feature Sets/Levels

Hardware feature scoping

- can be defined for different platforms or versions of the API
- all features listed in a set must be supported
- developers can develop against Feature Sets
- features enabled at device creation time

Vulkan Feature Set Definitions	
Platform	**Expected To Be Defined By**
Android	Platform Holder - Google
SteamOS	Platform Holder - Valve?
Linux	Khronos?
Windows	Khronos (Platform holder Microsoft is anticipated to decline)
Apple	Apple? Imagination? Khronos? Locked out?

[Microsoft:Sandy;Anandtech:Smith]

References

Smith, R., "Microsoft Announces DirectX12," *Anandtech*, Mar. 24, 2014
Smith, R., "Understanding AMD Mantle," *Anandtech*, Sep. 26, 2013
Smith, R., "Some Thoughts on Apple's Metal API," *Anandtech*, Jun. 3, 2014
Smith, R., "Khronos Announces Next Generation OpenGL Initiative," *Anandtech*, Aug. 11, 2014
Chester, B., "Comparing OpenGL ES to Metal on iOS," *Anandtech*, Jun. 15, 2015
Sandy, M., "DirectX 12," *DirectX Developer Blog*, Mar. 20, 2014
Lauritzen, A., "DirectX 12 on Intel," Aug. 11, 2014
Yeung, A., "DirectX 12 – Looking back at GDC 2015," Mar. 9, 2015
Langley, B., "Windows 10 and DirectX 12 Released!," *DirectX Developer Blog*, Jul. 29, 2015
Smith, R., "Microsoft Details Direct3D 11.3 & 12 New Rendering Features," *Anandtech*, Sep. 18, 2014
Smith, R., "The DirectX 12 Performance Preview," *Anandtech*, Feb. 6, 2015
Smith, R. and Cutress, I., "Exploring DirectX 12: 3DMark API Overhead Feature Test," *Anandtech*, Mar. 27, 2015
Smith, R., "Next Generation OpenGL Becomes Vulkan," *Anandtech*, Mar. 3, 2015
Smith, A., "Trying out the new Vulkan graphics API on PowerVR GPUs," *Imagination PowerVR Graphics Blog*, Mar. 3, 2015
Smith, A., "Gnomes per second in Vulkan and OpenGL ES," *Imagination PowerVR Graphics Blog*, Aug. 10, 2015
Foley, T., "Next-Generation Graphics APIs: Similarities and Differences," *ACM SIGGRAPH 2015*
Sellers, G., "A Whirlwind Tour of Vulkan," *ACM SIGGRAPH 2015*
Hall, J., "Vulkan on Android," *ACM SIGGRAPH 2015*
Hall, J., "Using Next-Generation APIs on Mobile GPUs," *ACM SIGGRAPH 2015*
Daniell, P., "Vulkan on NVIDIA GPUs," *ACM SIGGRAPH 2015*
Boyd, C., "Direct3D 12," *ACM SIGGRAPH 2015*
Yeung, A., "DirectX 12 Multiadapter," *DirectX Developer Blog*, Apr. 30, 2015
Smith, R., "GeForce+Radeon: Previewing DirectX 12 Multi-Adapter," *Anandtech*, Oct. 26, 2015
Merry, B., "Performance Tuning for Tile-Based Architecture," *OpenGL Insights*, eds. Cozzi, P. and Riccio, C., 2012