Khronos Connects Software to Silicon

Open interoperability standards to enable software to effectively harness the power of 3D and multiprocessor acceleration

3D graphics, XR, parallel programming, vision acceleration and machine learning

Non-profit, member-driven standards-defining industry consortium

Open to any interested company

All Khronos standards are royalty-free

Well-defined IP Framework protects participant’s intellectual property

Founded in 2000

>150 Members ~ 40% US, 30% Europe, 30% Asia
Khronos Active Initiatives

3D Graphics
Desktop, Mobile, Web
Embedded and Safety Critical
- Vulkan
- ANARI
- OpenGL ES
- OpenGL
- WebGL
- EGL
- OpenVX

3D Assets
Authoring and Delivery
- glTF
- COLLADA

Portable XR
Augmented and Virtual Reality
- OpenXR

Parallel Computation
Vision, Inferencing, Machine Learning
- SPIR
- OpenCL
- SYCL
- NNEF
- OpenVX

Guidelines for creating APIs to streamline system safety certification
Pervasive Vulkan

Desktop and Mobile GPUs

Platforms

- Desktop
- Android (Android 7.0+)
  (Vulkan 1.1 required on Android Q)
- Apple (via porting layers)
- Media Players
- Consoles
- Virtual Reality
- Cloud Services
- Game Streaming
- Embedded

Engines

Cider, Xenko, Godot, Unity, Unreal Engine, CryEngine, Cider, UX3D

Note: The version of Vulkan available will depend on platform and vendor

This work is licensed under a Creative Commons Attribution 4.0 International License

© The Khronos® Group Inc. 2020 - Page 4
Vulkan Roadmap

Vulkan 1.1 Extensions
- Maintenance updates plus additional functionality
- Timeline semaphores
- DX/HLSL compatibility
- Bindless resources
- Reduced precision arithmetic
- Formal memory model
- Buffer references
- SPIR-V 1.5

Roadmap Discussions
- Ray Tracing
- Variable Rate Shading
- Accelerated Video Encode/Decode
- Machine Learning Primitives
- Mesh Shaders

January 2020
Vulkan Ray Tracing

The industry’s first open, cross-vendor, cross-platform standard for ray tracing acceleration

Released November 2020

Coherent ray tracing framework with flexible merging of rasterization and ray tracing

Set of extensions to Vulkan, GLSL and SPIR-V seamlessly integrates ray tracing into Vulkan 1.X

Familiar to users of existing proprietary ray tracing APIs but also introduces new implementation flexibility

Hardware agnostic - can be accelerated on existing GPU compute and dedicated ray tracing cores

Primary focus on meeting desktop market demand for both real-time and offline rendering today - but designed to encourage mobile ray tracing too
Vulkan Ray Tracing Development Schedule

January 2018
Vulkan Ray Tracing Subgroup Created

March 2020
Provisional Extension Specifications
- Extension Specifications
- Press Release
- Blog
- Beta Drivers

Review and Integration of IHV and Developer Feedback
https://khr.io/vkrayprovfeedback
Streamline layering of DXR over Vulkan Ray Tracing
Multiple Usability Tweaks

Conformance Test Development
- Multiple Implementations
- Added support for Provisional Spec to DXC HLSL compiler
- Sample code using Provisional Spec

November 2020
Final Extension Specifications

December 2020
Vulkan SDK with Ray Tracing
Updated Samples
Ray Tracing Refresher

Ray tracing calculates how rays intersect and interact with scene geometry, materials and light sources.

Rasterization and ray tracing can both use triangles to describe scene geometry - but only ray tracing calculates physical phenomena such as shadows, reflections and refractions.

Ray Tracing is a Flexible Technique

Programmers need programmable flexibility to trace rays through scenes for a wide variety of visual effects - some examples...

- Ambient Occlusion
- Reflections
- Shadows
- Global Illumination
Step 1: Create Efficient Scene Geometry

Ray tracing may use a huge numbers of rays
Specialized data structures for interrogating scene geometry are necessary for efficient acceleration

Acceleration Structures
Contains low-level 3D geometry to be ray traced and high-level references into the geometry
Opaque internal organization details
Each vendor can optimize for processing for their hardware
E.g. Bounding Volume Hierarchy (BVH) for rapidly determining if there is any geometry in the path of a ray

Build Acceleration Structure
Vulkan driver integrates supplied geometry into its two-level Acceleration Structure

Using a BVH data structure to enable efficient ray tracing through a 3D scene
Step 2: Traverse Scene with Rays

Two ways to traverse Acceleration Structure
Launching rays into scene to generate results

Ray Tracing Pipelines
A new type of graphics pipeline
Implicit management of ray intersections
Application compiles a set of shaders into the pipeline to provide desired ray and material processing

Ray Queries
Any type of shader can launch a ray at any time
Shader can process intersection data however it wishes
Shader controls how traversal proceeds

Model courtesy of PTC
Traversal with Ray Tracing Pipelines

Implicit Ray and Shader Execution Management

Application compiles collection of shaders to be invoked on ray/geometry intersection into the Ray Tracing Pipeline.

Controlling which shaders are invoked during traversal enables a wide variety of ray tracing techniques.

Hit Shaders can query the materials they intersect e.g. transparent materials can be handled differently than opaque.

Intersection and Hit shaders can control how traversal proceeds.

Shader stages can communicate parameters and results through ray payload structures.

1. Launch 2D/3D grid of rays into scene contained in an Acceleration Structure.

2. ‘Intersection’ Shader computes ray intersections. Ray-triangle intersections are built-in.

3. Invoke ‘Any Hit’ Shader if intersection is found. Multiple intersections possible - arbitrary order.

4. Invoke ‘Closest Hit’ shader on the closest intersection of the ray OR Invoke ‘Miss’ Shader if no hit is found. Can trace more rays.

Ray Tracing Pipeline
Traversing with Ray Queries

Explicit Ray Management within a Single Shader

Any standard shader (e.g. compute, vertex and fragment shaders) can invoke a single ray traversal at any time. Uses an Acceleration Structure and a geometric description of the ray being traced. Shader reads intersection properties during traversal and controls how materials are processed and how the traversal proceeds.

1. Shader launches a single ray into a scene contained in an Acceleration Structure.

2. Shader takes action depending on intersection properties. Can trace more rays.

Graphics, Compute or Ray Tracing Pipeline
Pipeline Libraries

Ray Tracing Pipelines can use many shaders
Potentially orders of magnitude more shaders (1000s) than traditional applications to handle diverse tracing techniques and material types

Compilation Bottleneck
Compiling many shaders into a Ray Tracing Pipeline can be computationally intensive and cause application bottlenecks and stuttering

Vulkan Pipeline Library Extension
Enables a library of SPIR-V shaders to be incrementally compiled into an existing Ray Tracing Pipeline saving significant processing load

Multiple shaders used to build complex lighting in a Quake 2 scene
Host Offload of Setup Operations

Ray tracing setup compute workloads can be intensive
Building Acceleration Structures and compiling Ray Tracing Pipelines
Two Vulkan mechanisms to offload and control setup workloads
on the host CPU(s) for smoother, faster rendering

**Build Acceleration Structure on Host**
Use the host to build Acceleration Structure in host memory and then
copy to the GPU - rather than build directly on the GPU
Final size of Application Structure is known before copying to the
GPU - enabling optimized GPU memory allocation

**Deferred Host Operations**
Driver returns deferred work handle to application for later execution
Application controls work execution and can chose to distribute onto
multiple cores and background threads

Deferred Host Operations can be used to
asynchronously use multiple CPU cores to build
Acceleration Structures on the host

Using Deferred Host Operations to build a
complex Acceleration Structure using
multiple CPU cores to offload the work from
the GPU for faster, smoother framerates
# Vulkan Ray Tracing and DXR

<table>
<thead>
<tr>
<th></th>
<th>Vulkan Ray Tracing</th>
<th>DX12 / DXR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray Tracing Pipelines</td>
<td>At least one must be available</td>
<td>Yes</td>
</tr>
<tr>
<td>Ray Queries</td>
<td>DXR Tier 1.1 Inline ray tracing</td>
<td></td>
</tr>
<tr>
<td>Language for Ray Tracing Shaders</td>
<td>GLSL or HLSL</td>
<td>HLSL</td>
</tr>
<tr>
<td>Pipeline Libraries</td>
<td>Yes</td>
<td>DXR Tier 1.1 AddToStateObject()</td>
</tr>
<tr>
<td>Build Acceleration Structure on Host</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>Deferred Host Operations</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>Capture/Replay Support for Tools (e.g. RenderDoc)</td>
<td>Optional</td>
<td>No</td>
</tr>
</tbody>
</table>

**Porting code between Vulkan Ray Tracing and DXR is straightforward**

Including re-use of ray tracing shaders written in HLSL

**Final extensions designed to enable layering DXR over Vulkan Ray Tracing**

To enable ray tracing in DX12 over Vulkan Layers such as vkd3d
Vulkan Ray Tracing includes GLSL and SPIR-V Extensions

Enabling compiled GLSL/SPIR-V shaders to operate in a Ray Tracing Pipeline - similar to HLSL features used in Direct3D’s DXR

HLSL and Vulkan with DXC
Microsoft’s DXC HLSL compiler was open sourced in Jan 2017
Google and others have added SPIR-V code generation to DXC with Microsoft’s knowledge and approval
Vulkan developers can now choose between GLSL and HLSL!

HLSL for Vulkan Ray Tracing
NVIDIA added code generation to DXC to generate SPIR-V for the Vulkan Ray Tracing extension from HLSL

Developers can port HLSL shaders with minimal changes between Vulkan Ray Tracing and DXR
### API Layering

<table>
<thead>
<tr>
<th>Layers Over</th>
<th>Vulkan</th>
<th>OpenGL</th>
<th>OpenCL</th>
<th>OpenGL ES</th>
<th>DX12</th>
<th>DX9-11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vulkan</td>
<td></td>
<td>Zink</td>
<td>clspv</td>
<td>GLOVE Angle</td>
<td>vkd3d-Proton</td>
<td>DXVK WineD3D</td>
</tr>
<tr>
<td>OpenGL</td>
<td>gfx-rs</td>
<td>Ashes</td>
<td></td>
<td>Angle</td>
<td></td>
<td>WineD3D</td>
</tr>
<tr>
<td>DX12</td>
<td>gfx-rs</td>
<td></td>
<td>Microsoft ‘GLOn12’</td>
<td>Microsoft ‘CLOn12’</td>
<td></td>
<td>Microsoft D3D11On12</td>
</tr>
<tr>
<td>DX9-11</td>
<td>gfx-rs</td>
<td>Ashes</td>
<td></td>
<td>Angle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metal</td>
<td></td>
<td></td>
<td>clspv + SPIRV-Cross?</td>
<td>MoltenGL Angle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- E.g. Vkd3d-Proton used to port DX12 titles to Linux with Valve Proton
- Vulkan is an effective porting target for multiple APIs e.g. for bringing DX12 games on Linux. Layering DX12 DXR over Vulkan Ray Tracing will be needed.
Thank you!
Any questions?