Vulkan Ray Tracing Development Timeline

- **January 2018**: Vulkan Ray Tracing Subgroup Created
- **March 2020**: Provisional Extension Specifications
- **November 2020**: Final Extension Specifications
- **December 2020**: Vulkan SDK with Ray Tracing

**Key Events**:
- **Review and Integration of IHV and Developer Feedback**
- **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**
- **Updated Samples**
Vulkan Ray Tracing Design

- The industry’s first open, cross-vendor, cross-platform standard for ray tracing acceleration
  - Coherent ray tracing framework with flexible merging of rasterization and ray tracing
  - Set of extensions to Vulkan, SPIR-V, and GLSL
  - 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
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
VK_KHR_acceleration_structure

- Create, build, and manage acceleration structures (AS)
- Opaque, implementation-defined data structures, placed on a buffer
- Two-level acceleration structure
  - Bottom-level contains triangles or AABBs
  - Top-level refers to bottom-level
- Reference to bottom-level AS includes transform and shading properties
- Top-level AS is accessed from the shader either as a descriptor binding or by device address
Step 2: Scene Traversal

- **Ray tracing pipelines**
  - Application compiles a set of shaders into the pipeline to provide desired ray and material processing
  - Application launches grid of rays into scene
  - Implicit management of ray intersections

- **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

---

**Ray Tracing Pipeline**

- RayGen Shader
- Acc. Structure Traversal
- Ray Tracing Pipeline

**Shader Binding Table (SBT) Buffers**

- Shading Info
- Transform & Properties

**Top Level Acceleration Structure (AS)**

- Closest Hit

**Bottom Level AS**

- Vertex & Index Buffers
- Instance Buffer

**Acc. Structure Hierarchy**

- Intersection
- Hit?
- Any Hit
- Recursion
- Miss

**Graphics, Compute or Ray Tracing Pipeline**

- RayQuery
- Proceed?
- Handle Result
- Confirm Hit, Generate Details, or Terminate

**Acceleration Structure Hierarchy**

- Explicit Ray Management within Single Shader
VK_KHR_ray_tracing_pipeline

- Provides ray tracing shaders and pipelines
  - SPV_KHR_ray_tracing / GLSL_EXT_ray_tracing
    - Support for ray tracing shader stages, builtins, storage classes, and instructions

- **vkCmdTraceRaysKHR** launches ray generation shader with launch grid and shader addresses
  - `traceRayEXT(...)` from a shader starts acceleration structure traversal

- **Traversal** invokes intersection and any hit shaders as needed to control traversal
  - Upon completion, traversal invokes a miss or closest hit shader
  - Hit attributes store hit information between intersection and hit shaders
  - Ray payloads store control and result information

- **Shader binding table provides shading info**
  - Shader group handle for each type of shader
  - Optional shader buffer record for instance-specific data (e.g., buffer device addresses or descriptor indices)
VK_KHR_ray_query

- Enables ray query intrinsics for all shader stages (graphics, compute, ray tracing)
  - SPV_KHR_ray_query / GLSL_EXT_ray_query
  - Ray query shader instructions

- Execute ray traversal in any shader

- `rayQueryInitializeEXT(...)` from a shader initializes a query with an AS and specific traversal properties

- Call `rayQueryProceedEXT(..)` to start or continue traversal

- Shader must explicitly examine potential intersections (type, geometry, location), and specify how it should be handled (generate intersection details, confirm hit, terminate ray)
VK_KHR_pipeline_library

• 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 in a ray tracing pipeline can be computationally intensive and cause application bottlenecks and stuttering

• Pipeline library
  - 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 workloads can be compute intensive
  - AS builds and compiling ray tracing pipelines
  - Two 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 AS in host memory and then copy to the GPU - rather than build directly on the GPU
  - Final size of AS is known before copying to the GPU - enabling optimized GPU memory allocation

- **VK_KHR_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
  - Can 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
Developer Vulkan Ray Tracing Resources

Production Vulkan Drivers with Vulkan Ray Tracing are shipping

- AMD Radeon Adrenalin 20.11.3 drivers for Radeon RX 6000 Series
- NVIDIA R460 drivers for All RTX GPUs
- GeForce GTX 1660 with 6GB+ of memory
- GeForce GTX 1060+ with 6GB+ of memory
- Intel Xe-HPG GPUs, available in 2021

Vulkan Ray Tracing Samples
Vulkan Ray Tracing Guide
How to use the Vulkan Ray Tracing extensions
Exploring deeper technical details of the Vulkan Ray Tracing specifications
Best practices for blending Vulkan rasterization and ray tracing techniques

Khronos Member Materials
Deep dive Vulkan Ray Tracing Tutorial
How to use Vulkan Ray Tracing to Create a complete mini-path tracer
A Vulkan-based glTF ray tracing viewer with open source on GitHub

New textbook, and here, on principles and history of ray tracing by Jon Peddie

Khronos welcomes developer feedback on Vulkan GitHub issues tracker