Bringing Ray Tracing to Vulkan

Nuno Subtil, NVIDIA
October 2019
Outline

Motivation for Ray Tracing vs. Rasterization

Current Ray Tracing Ecosystem

API Overview
Ray Tracing vs. Rasterization

- Rasterization: evaluate one triangle at a time
  - Store and accumulate some of the output data, but discard most of it

- Computationally cheap, but “local”
- Amenable to hardware implementation
Ray Tracing vs. Rasterization

- Ray tracing: sampling the entire scene one ray at a time
  - Entire scene is evaluated
  - Can visit same primitive many times

- Extremely flexible
- Can be computationally intensive
Ray Traced Shadows

Physically correct shadows
Sharp and Contact Hardening
Ray Traced Ambient Occlusion

Improved ambient occlusion detail with fewer artifacts
Ray Traced Reflections

Physically-correct reflections on arbitrary surfaces
Full Real-Time Path Tracing

Physically-correct light transport in a complex game
Current Ray Tracing Ecosystem

• Existing APIs
  - VK_NV_ray_tracing (NVIDIA only)
  - Microsoft DXR (cross-vendor)

• Ray tracing capable platforms
  - Turing GPU architecture (NVIDIA)
  - Recent market announcements suggest more incoming

• Ray-tracing capable applications showing up
  - Battlefield V (DICE)
  - Wolfenstein: Youngblood (MachineGames)
  - Control (Remedy)
  - JX3 Online (Kingsoft)
  - Quake II (id Software/Bethesda/NVIDIA)
  - ...
Bringing Ray Tracing to Vulkan

(A preview based on VK_NV_ray_tracing)
Bringing Ray Tracing to Vulkan

• Cross-vendor KHR extension under development
  - We can’t share too much detail just yet

• Largely based on and functionally similar to VK_NV_ray_tracing
  - ... but keep in mind KHR hasn’t shipped yet!

• We’ll be presenting concepts from VK_NV_ray_tracing, not a cross-vendor version
  - We expect final KHR version to be a superset
  - The functionality we’re presenting should be very similar between NV and KHR

  - ... but keep in mind KHR hasn’t shipped yet!
Graphics Pipelines

Rasterization

Draw Call → Scheduling → Vertex Shading → Rasterization → Fragment Shading → ROP

Ray Tracing

Ray Generation → Scheduling → Traversal & Intersection → Scheduling → Shading
Ray Tracing Building Blocks

Acceleration Structure
Abstraction for scene geometry, enables efficient scene traversal

Ray tracing shader domains
Handle ray generation and termination, intersections, material shading

Shader binding table
Links shaders and resources to be used during traversal

Ray tracing pipeline state object
Collection of ray tracing shader binaries
Acceleration Structures

Ray tracing: testing one ray against all scene primitives

Acceleration structures required for efficient operation
Acceleration Structures

- Dual-level acceleration structure
- Opaque (implementation-defined) data structures
- Efficient build and update
Building Acceleration Structures
Building Acceleration Structures
Creating Acceleration Structures

- New object type
  VkAccelerationStructureNV

- New descriptor type for binding AS to shaders

```c
struct VkAccelerationStructureInfoNV {
    VkStructureType sType;
    void *pNext;
    // top-level or bottom-level
    VkAccelerationStructureTypeNV type;
    VkAccelerationStructureCreateFlagsNV flags;
    uint32_t instanceCount;
    uint32_t geometryCount;
    const VkGeometryNV* pGeometries;
};

vkCreateAccelerationStructureNV(
    VkDevice device,
    const VkAccelerationStructureCreateInfoNV *info,
    const VkAllocationCallbacks *pAllocator,
    VkAccelerationStructureNV *pAccelerationStruct
);
```
Acceleration Structure Memory Management

- Backing store for AS is managed by application
- Memory requirements will vary

```c
vkGetAccelerationStructureMemoryRequirementsNV(
    VkDevice device,
    const VkAccelerationStructureMemoryRequirementsInfoNV *pInfo,
    VkMemoryRequirements2 *pMemoryRequirements
);

vkBindAccelerationStructureMemoryNV(...);
```
Acceleration Structure Build/Update

- Single command to build or update an AS
- AS updates have restrictions for efficiency

```c
vkCmdBuildAccelerationStructureNV(
    VkCommandBuffer cmdBuf,
    const VKAccelerationStructureInfoNV*,
    VkBuffer instanceData,
    VkDeviceSize instanceOffset,
    VkBool32 update,
    VkAccelerationStructureNV dst,
    VkAccelerationStructureNV src,
    VkBuffer scratch,
    VkDeviceSize scratchOffset
);
```
Acceleration Structure Copy

- Acceleration structures can be copied, optionally compacting during the copy

```c
vkCmdCopyAccelerationStructureNV(
    VkCommandBuffer cmdBuf,
    VkAccelerationStructureNV dst,
    VkAccelerationStructureNV src,
    VkCopyAccelerationStructureModeNV mode
);
```
Ray tracing shader domains

Ray Generation

traceNV()

Acceleration Structure Traversal

Hit?

no

Miss

yes

Closest Hit

Any Hit

Intersection
Inter-shader Communication

- **Ray Payload**
  - Application-defined structure to pass data between hit stages and shader stage that spawned a ray
  - Used to return final intersection information to ray generation shader
Inter-shader Communication

- Ray Attributes
  - Application-defined structure to pass intersection information from intersection shader to hit shaders

![Diagram showing ray generation, acceleration structure traversal, and hit determination]

Intersection: write attributes
Any Hit: read attributes

Intersection: write attributes
Closest Hit: read attributes
Ray Generation Shaders

- Starting point for all ray tracing work
- Simple grid of threads launched from host
- Traces rays, writes final output
Intersection Shaders

- Compute ray intersections with app-defined primitives
- Ray-triangle intersections built-in
Any-Hit Shader

- Invoked after an intersection is found
- Multiple intersections invoked in arbitrary order
Closest-Hit Shaders

- Invoked on the closest intersection of the ray
- Can read attributes and trace rays to modify the payload
**Miss Shader**

- Invoked if no hit is found and accepted
- Can trace rays and modify ray payload

[Diagram of ray tracing process]

- Ray Generation
- `traceNV()`
- Acceleration Structure Traversal
- Hit?
  - no → Miss
  - yes → Closest Hit
  - Any Hit
  - Intersection
Mapping to GLSL: Inter-shader Interface

- Ray payload and intersection attributes: new block decorations

  \[
  \text{rayPayloadNV} \{ \ldots \} \\
  \text{intersectionAttributesNV} \{ \ldots \}
  \]

- `traceNV()` builtin may be called with different payload structures in a single shader

  \[
  \text{layout} \ (\text{location}=1) \\
  \text{rayPayloadNV} \text{ firstInterface} \{\ldots\}
  \]

  \[
  \text{layout} \ (\text{location}=2) \\
  \text{rayPayloadNV} \text{ secondInterface} \{\ldots\}
  \]

  \[
  \text{traceNV}(\ldots, 1) \ // \text{firstInterface} \\
  \text{traceNV}(\ldots, 2) \ // \text{secondInterface}
  \]
Mapping to GLSL: Tracing Rays

- **Built-in variable decorations:**
  - ID/index information
  - ray parameters, hit parameters
  - instance transforms

- **New functions:**
  - `traceNV()` traces a ray into the scene
  - `reportIntersectionNV()` outputs intersection information
  - `ignoreIntersectionNV()` rejects an intersection
  - `terminateRayNV()` terminates the current ray

```c
traceNV(
    accelerationStructure topLevel,
    uint rayFlags,
    uint cullMask,
    uint sbtRecordOffset,
    uint sbtRecordStride,
    uint missIndex,
    vec3 origin,
    float tmin,
    vec3 direction,
    float tmax,
    int payload
);

reportIntersectionNV(
    float hit,
    uint hitKind
);

ignoreIntersectionNV();

terminateRayNV();
```
Shader Binding Table

- A given ray query into the scene can hit any object
  - ... with different kinds of material shaders
  - ... requiring different kinds of resources (e.g., textures)
  - ... and different parameters (e.g., transparency value)
  - ... per object instance!
Shader Binding Table

- Array of opaque shader handles + application storage inside VkBuffer

- Indexing used to determine shaders / data used

- App-defined stride for SBT records to accommodate different resource sets
Shader Binding Table

Shader Binding Table Buffer

- Shader Reference
- Custom Data & Indices
- Shader Reference
- Custom Data & Indices

RT Pipeline with RayGeneration

- Descriptor Sets
- Acceleration Structures

Traversal yields instanceOffset

RT Pipeline with ClosestHit

- Descriptor Sets

Shader reacts on hit (can trace)

Textures etc.
Shader Binding Table and `vkCmdTraceRaysNV`

```
vkCmdTraceRaysNV(VkCommandBuffer, VkBuffer shaderBindingTable, raygenIndex, ...)```

1: API invokes RayGeneration Shader

- **Shader Binding Table Buffer**
  - Shader Reference
  - Custom Data & Indices
  - ... (repeated)

2: Shader traces rays

- RT Pipeline with RayGeneration
- Descriptor Sets
- Acceleration Structures

3: Traversal yields `instanceOffset`

- RT Pipeline with ClosestHit
- Descriptor Sets
- Textures etc.

4: Shader reacts on hit (can trace)
Ray Tracing Pipeline Creation

- Ray tracing pipeline contains all shaders used together
- Can have multiple shaders of the same stage
- Facilities provided to enable multi-threaded shader compilation within a ray tracing PSO

```c
struct VkRaytracingPipelineCreateInfoNV {
    VkStructureType sType;
    void *pNext;

    VkPipelineCreateFlags flags;
    uint32_t stageCount;
    const VkPipelineShaderStageCreateInfo *stages;

    uint32_t groupCount;
    const VkRayTracingShaderGroupCreateInfoNV *pGroups;

    uint32_t maxRecursionDepth;
    VkPipelineLayout layout;
    VkPipeline basePipelineHandle;
    int32_t basePipelineIndex;
};

vkCreateRaytracingPipelinesNV(...);
```
Ray Tracing Shader Handles

- Exposes opaque shader handles to be used inside Shader Binding Table

```c
vkGetRaytracingShaderGroupHandlesNV(
    VkDevice device,
    VkPipeline pipeline,
    uint32_t firstGroup,
    uint32_t groupCount,
    size_t dataSize,
    void *pData
);
```
vkCmdTraceRaysNV

- Entry point to invoke ray tracing work
- Uses existing vkCmdBindPipeline and vkCmdBindDescriptorSets with new bind point

```c
vkCmdTraceRaysNV(
    VkCommandBuffer cmdBuf,
    VkBuffer raygenSBT,
    VkDeviceSize raygenSBTOffset,
    VkBuffer missSBT,
    VkDeviceSize missSBTOffset,
    VkDeviceSize missSBTStride,
    VkBuffer hitSBT,
    VkDeviceSize hitSBTOffset,
    VkDeviceSize hitSBTStride,
    VkBuffer callableSBT,
    VkDeviceSize callableSBTOffset,
    VkDeviceSize callableSBTStride,
    uint32_t width,
    uint32_t height,
    uint32_t depth
);
```
Example Ray Generation Shader

// Binding for the acceleration structure
layout(set = 0, binding = 0) accelerationStructureNV scene;
layout(set = 1, binding = 0) image2D framebufferImage;

layout(set = 2, binding = 0, std140) uniform traceParameters {
    uint sbtMiss;
    uint sbtOffset;
    uint sbtStride;
    vec3 origin;
    vec3 dir;
}

layout(location = 1) rayPayloadNV primaryPayload {
    vec4 color;
}

void main() {
    primaryPayload.color = vec4(0.0, 0.0, 0.0, 0.0);

    traceNV(scene, 0, 0, sbtMiss, sbtOffset, sbtStride, traceParameters.origin,
            0.0, computeDir(gl_GlobalInvocationIDNV.xy, traceParameters.dir),
            1.e9, 1);

    imageStore(framebufferImage, gl_GlobalInvocationIDNV.xy, primaryPayload.color);
}
Example Closest Hit Shader

// Closest hit shader
//
// Closest hit shaders only have one payload -
// for the incoming ray
rayPayloadInNV primaryPayload {
    vec4 color;
}

layout(set = 3, binding = 0) sampler3D solidMaterialSampler;

void main() {
    vec3 pos = gl_WorldRayOriginNV + gl_hitTNV * gl_WorldRayDirectionNV;

    primaryPayload.color = texture(
        solidMaterialSampler, pos);
}
Iterative Loop Path Tracing in RayGen Shader

```
vec3 color = 0.0f;
Float sample_pdf = 1.0f;
vec3 path_throughput = 1.0f;

// GBufferEntry contains all necessary info to reconstruct the hit and local brdf
GBufferEntry gbuffer_entry = imageLoad(gbuffer, gl_GlobalInvocationID.xy);

Ray ray = reconstruct_ray( gbuffer_entry, pixel ); // reconstruct the first ray

for (uint32 bounce = 0; bounce < max_bounces; ++bounce) {
    // unpack the brdf
    Brdf brdf = get_brdf( gbuffer_entry );

    // accumulate emission of the local surface
    color += path_throughput * local_emission( gbuffer_entry, brdf, sample_pdf );

    // perform next-event estimation (casts one or more shadow rays, getting visibility as a return value)
    color += path_throughput * next_event_estimation( gbuffer_entry, brdf, sample_pdf );

    // sample the brdf
    vec3 sample_throughput;
    ray = sample_brdf( gbuffer_entry, brdf, sample_throughput, sample_pdf );

    // perform Russian-Roulette to terminate path
    if (random() >= max(sample_throughput))
        break;

    // accumulate the path throughput
    path_throughput *= sample_throughput / max(sample_throughput);

    // trace a ray and get a brdf as a return value
    gbuffer_entry = trace(..., ray.origin, ray.direction, ...);
}
```
HLSL Support

Mainline DXC can compile DXR shaders into SPIR-V

- Currently targeting NV vendor extension, but likely easy to convert to KHR
- NVIDIA intends to contribute support for KHR version once it is final

Resource binding model is different between DXR and Vulkan

- DXR: SBT contains local root signature, used by the bytecode compiler to map resources
- VK_NV_ray_tracing: SBT is just memory, used by the shader directly
  - Intent is for app to use EXT_descriptor_indexing to make resources available, then store indices in the SBT record
  - EXT_device_buffer_address allows buffer pointers in SBT

Shader Binding Table exposed as a constant-buffer-like binding

- General problem of mapping SBT to a DXR local root signature can be solved, but no implementation work has been done
- Developer feedback suggests this is enough in practice
Conclusion

• Ray tracing is coming to Vulkan!
  - Driven by industry interest
  - Enables growing ecosystem of ray tracing capable apps to target Vulkan
  - Current design largely compatible with shipping NV extension