NVIDIA RTX: Enabling Ray Tracing in Vulkan

Nuno Subtil, Sr. Devtech Engineer / Eric Werness, Sr. System Software Engineer

March 27, 2018

Overview

Ray tracing vs. rasterization

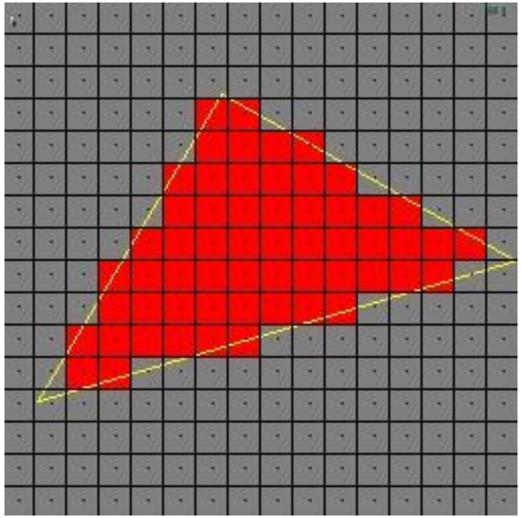
How did we get here?

Real-time ray tracing applications in 2018

Exposing RTX through Vulkan

Ray Tracing vs. Rasterization

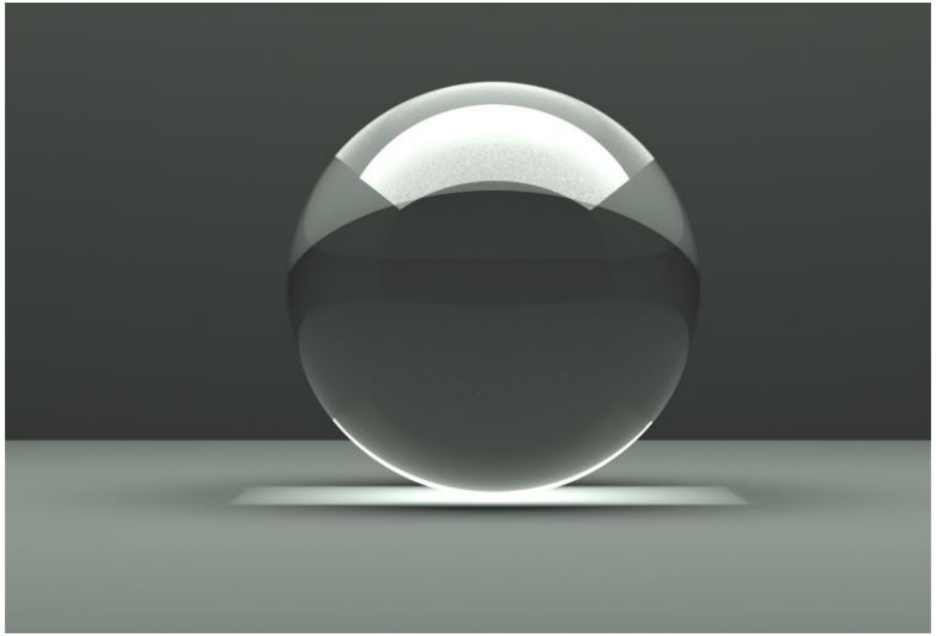
- Rasterization: evaluate one triangle at a time
  - Store and accumulate some of the output data, but discard most of it



- 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

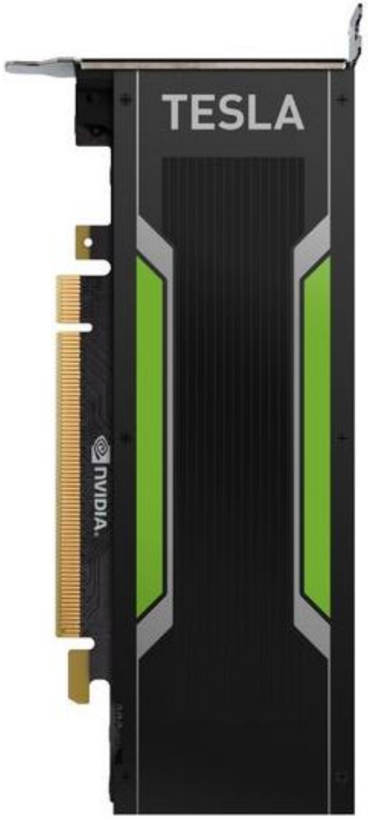


- Extremely flexible
- Can be computationally intensive

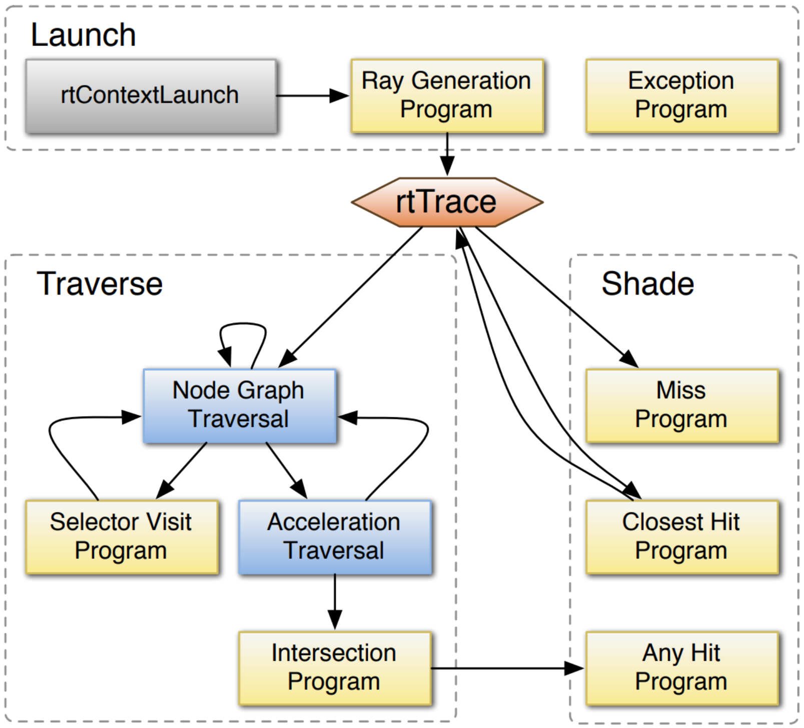
Ray Tracing at NVIDIA

Decade+ of R&D

- Real-time graphics: rasterization is king
  - GPUs evolve to enable powerful rasterization-based algorithms
- 2007: CUDA brings general purpose computing to GPUs
  - Ray tracing research focused on acceleration via GPGPU



OptiX: A General Purpose Ray Tracing Engine




```
RT_PROGRAM void pinhole_camera() {
  Ray ray = PinholeCamera::makeRay( launchIndex );
  UserPayload payload;
  rtTrace( topObject, ray, payload );
  outputBuffer[launchIndex] = payload.result;
}
```

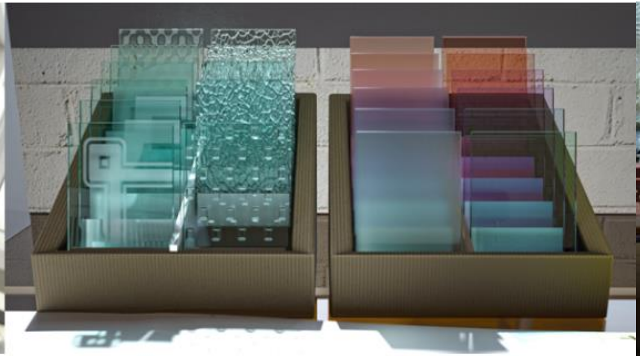
Figure 3: *Example ray generation program (in CUDA C) for a single sample per pixel. The 2-dimensional grid location of the program invocation is given by the semantic variable* launchIndex, *which is used to create a primary ray using a pinhole camera model. Upon tracing a ray, the invoked material hit programs fill the* result *field of the user-defined payload structure. The variable* topObject *refers to the location in the scene hierarchy where ray traversal should start, typically the root of the node graph. At the location specified by* launchIndex, *the result is written to the output buffer to be displayed by the application.*

Parker *et al*, **OptiX: A General Purpose Ray Tracing Engine**, SIGGRAPH 2010

Today: OptiX and Pro GPU Rendering

- **OptiX:** General purpose GPU ray tracing SDK

- **OptiX** abstraction supports future innovation



NVIDIA RTX

"NVIDIA RTX opens the door to make real-time ray tracing a reality!"

— Kim Libreri, CTO, Epic Games

"With NVIDIA GV100 GPUs and RTX, we can now do real-time ray tracing. It's just fantastic!"

— Sébastien Guichou, CTO of Isotropix

NVIDIA RTX

Applications

NVIDIA DesignWorks | NVIDIA GameWorks

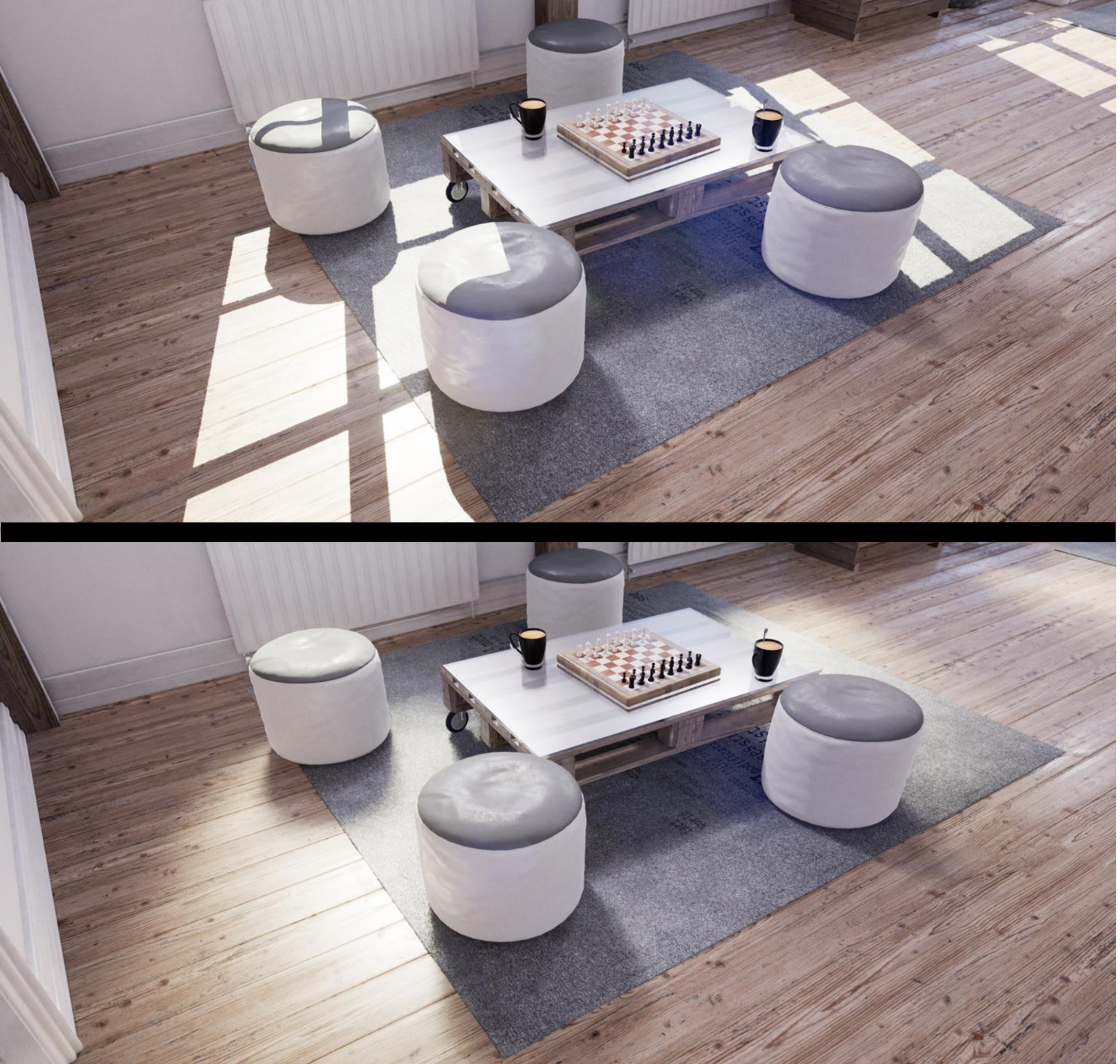
OptiX
for CUDA

Raytracing Extensions
for Vulkan

DirectX Raytracing
for Microsoft DX12

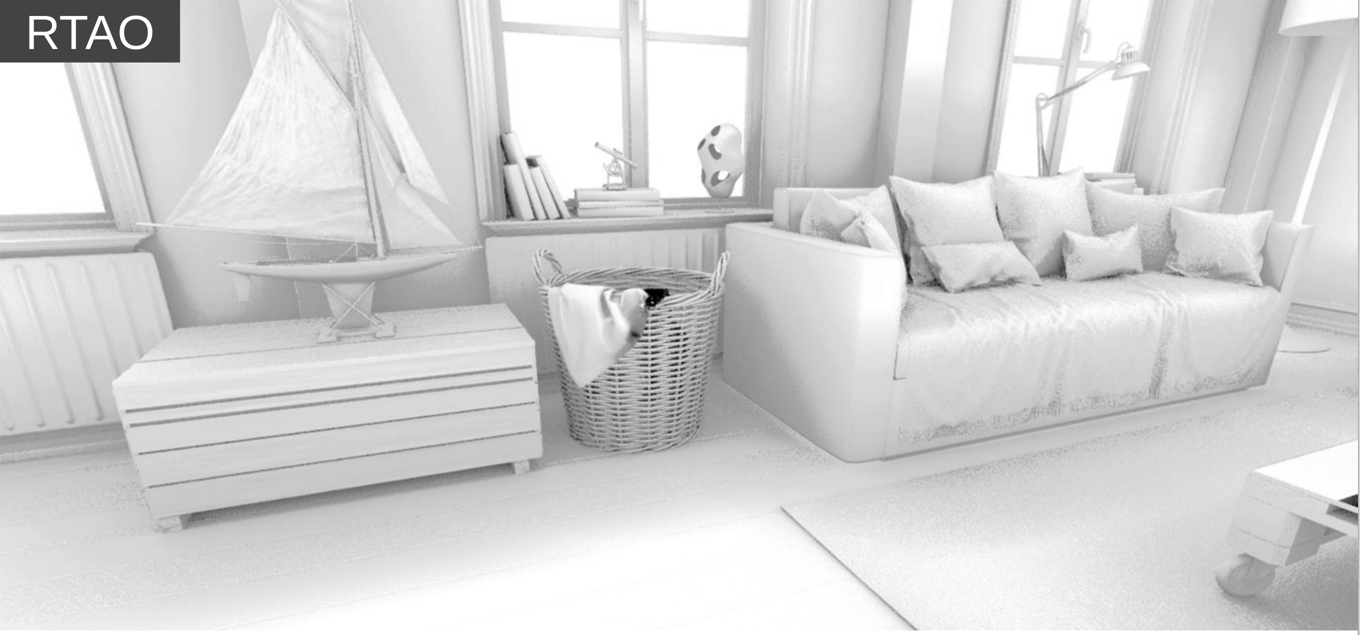
NVIDIA® RTX™ Technology

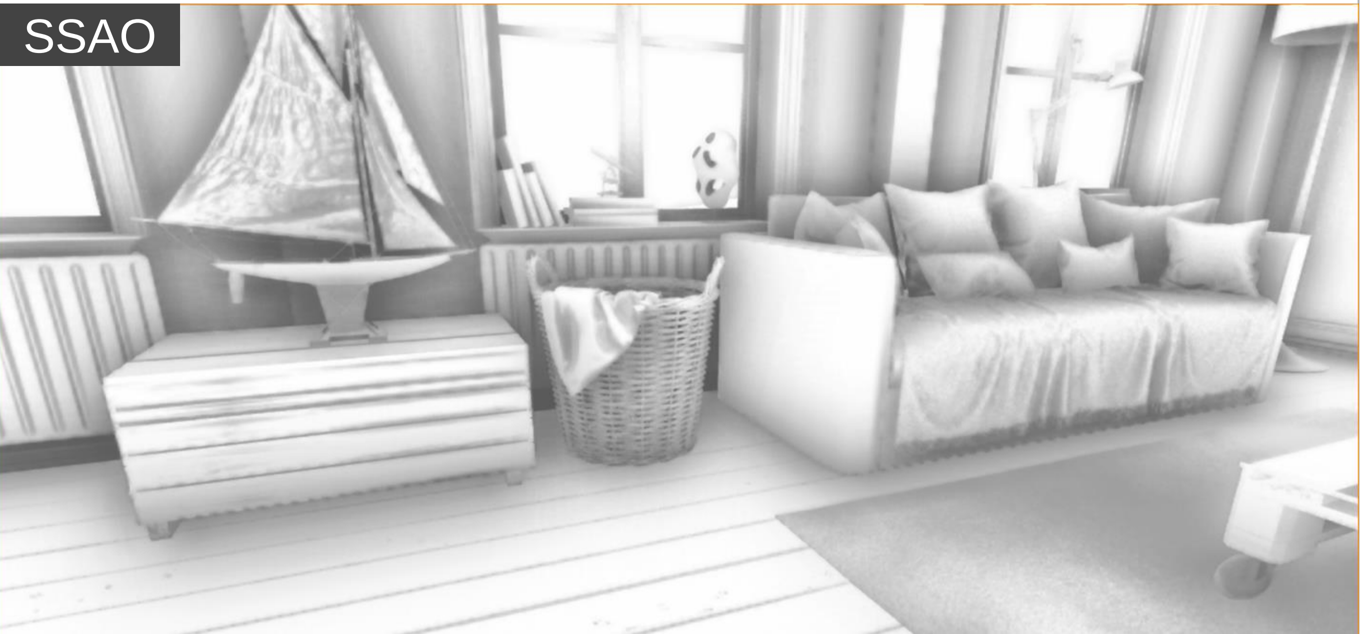
NVIDIA Volta GPU



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

Ray Tracing: Not Just Graphics

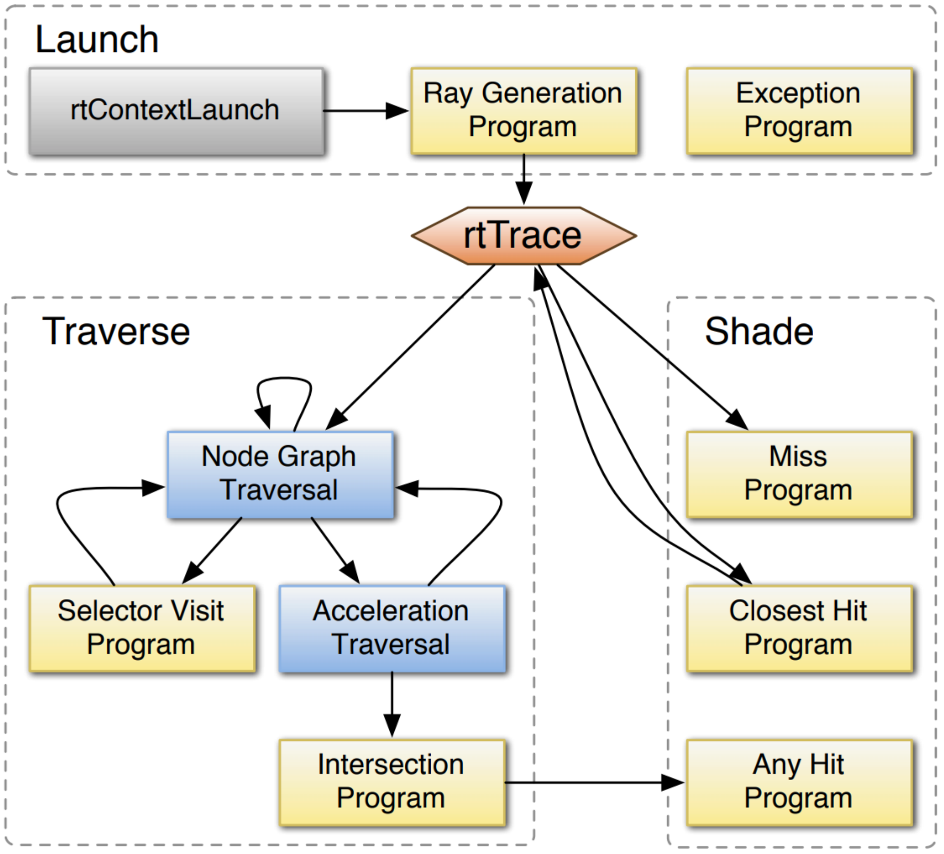
- Physics, particle simulations
- Audio path tracing / simulation
- AI visibility queries

Exposing RTX through Vulkan

VK_NV_raytracing

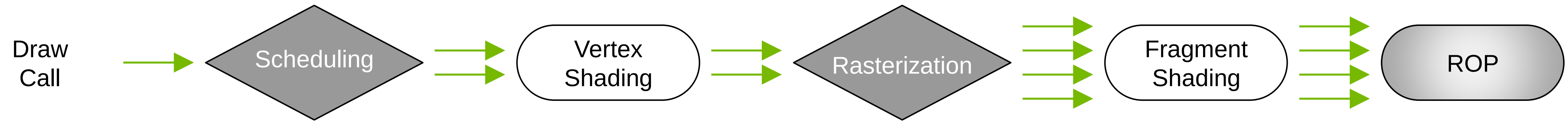
API design principles

- Proven ray tracing API concepts
- Hardware-agnostic
- Good fit with Vulkan API
- Implementable on compute functionality

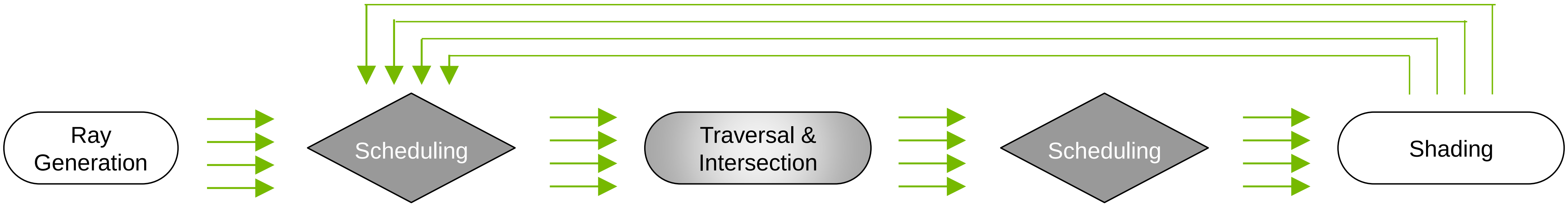


Graphics Pipelines

Rasterization



Ray Tracing



Building the Pipeline

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 object**

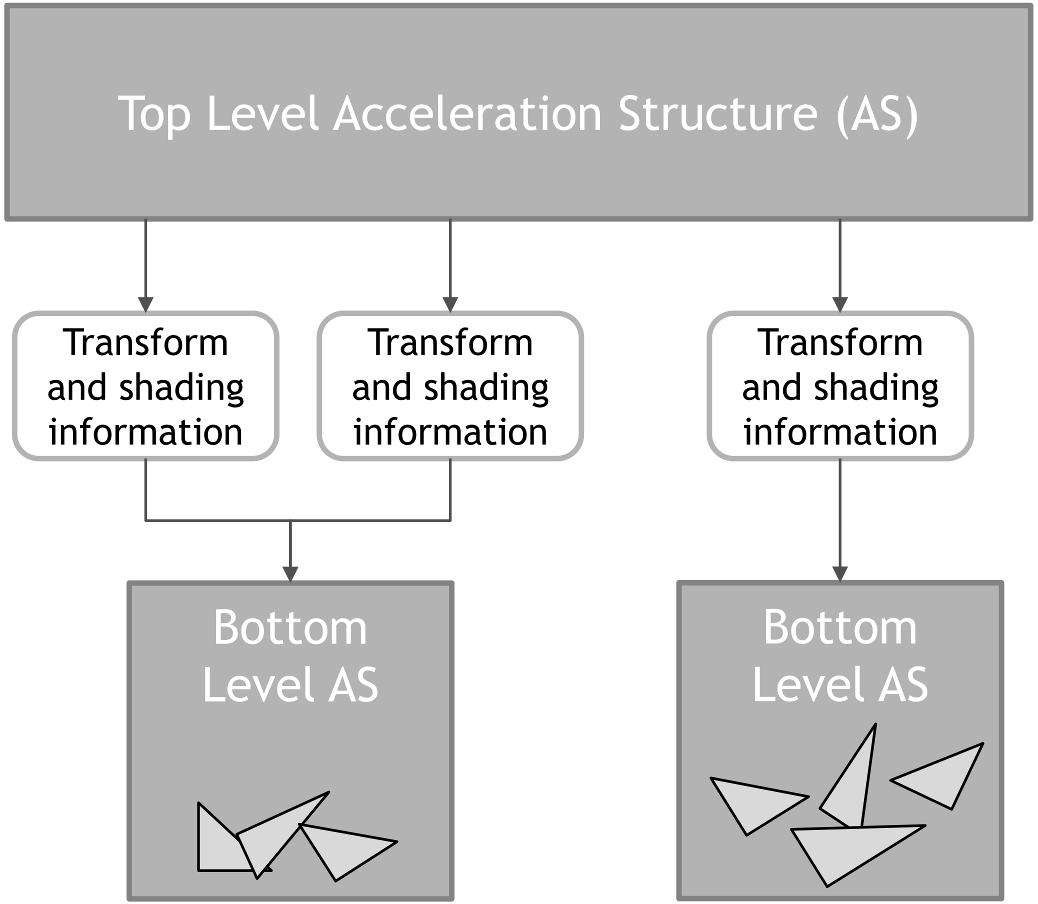
Acceleration Structure Concepts

- Ray tracing: testing **one ray against all scene primitives**

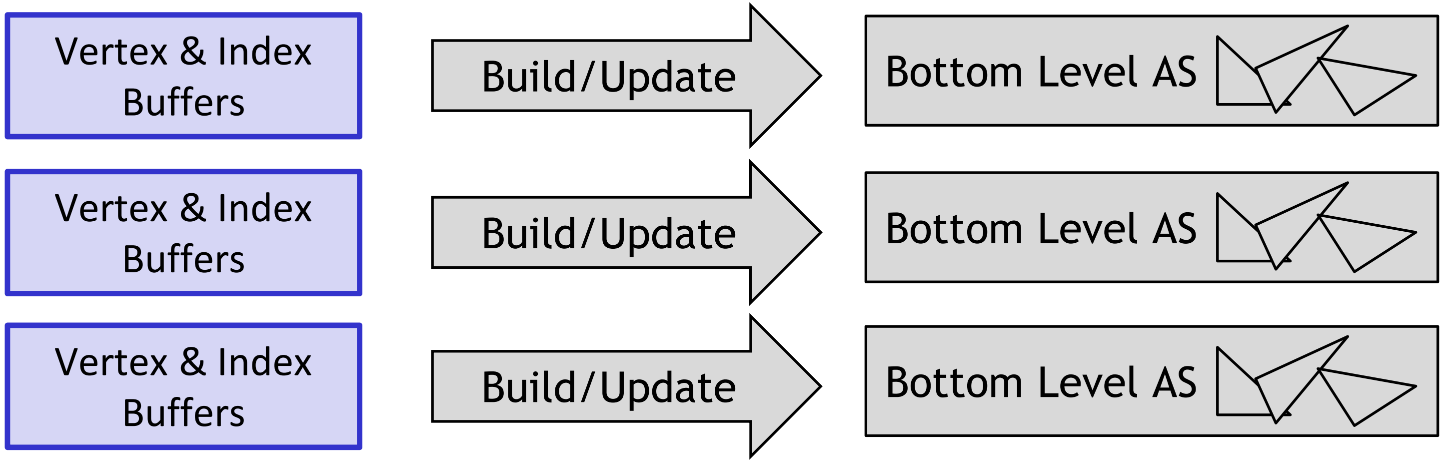
- Acceleration structures **required for efficient operation**

Acceleration Structure Concepts

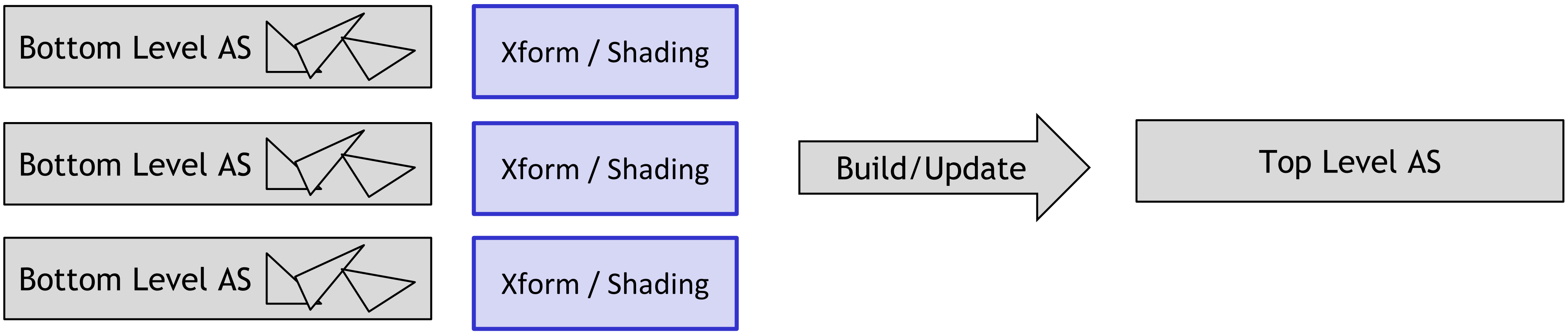
- Dual-level acceleration structure
- Opaque (implementation-defined) data structures
- Efficient build and update



Building Acceleration Structures



Building Acceleration Structures



Creating Acceleration Structures

- New object type `VkAccelerationStructureNV`
- New descriptor type for binding AS to shaders

```
struct VkAccelerationStructureCreateInfoNV {
  VkStructureType sType;
  void *pNext;
  // top-level or bottom-level
  VkAccelerationStructureTypeNV type;
  VkAccelerationStructureCreateFlagsNV flags;

  // used when copying an existing object
  VkAccelerationStructureNV *pOriginal;

  uint32_t numInstances;
  uint32_t numGeometries;
  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

```
vkGetAccelerationStructureMemoryRequirementsNV(
  VkDevice device,
  const
VkAccelerationStructureMemoryRequirementsInfoNV
*pInfo,
  VkMemoryReqiurements2 *pMemoryRequirements
);

vkBindAccelerationStructureMemoryNV(…);
```

Acceleration Structure Build/Update

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

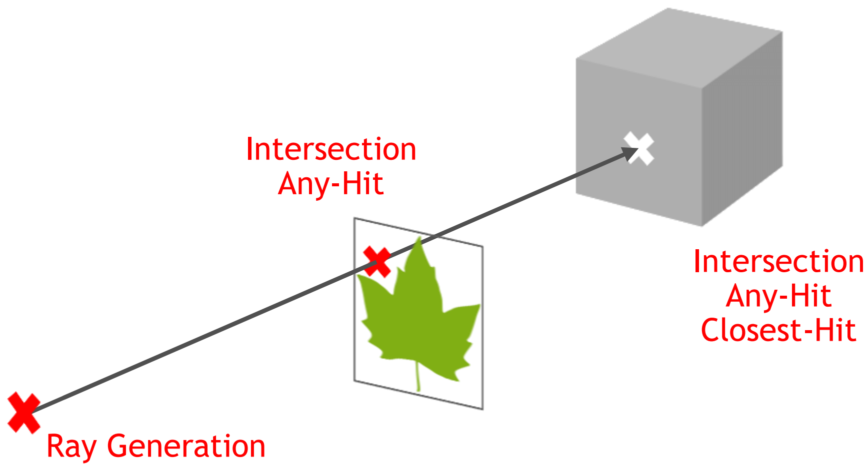
```
vkCmdBuildAccelerationStructureNV(
  VkCommandBuffer cmdBuf,
  VkAccelerationStructureTypeNV type,
  uint32_t numInstances,
  const VkBuffer instanceData,
  const VkDeviceSize instanceOffset,
  uint32_t numGeometries,
  const VkGeometryNV *pGeometries,
  VkBuildAccelerationStructureFlagsNV
     flags,
  VkBool32 update,
  VkAccelerationStructureNV dst,
  VkAccelerationStructureNVsrc,
  VkBuffer scratch
);
```

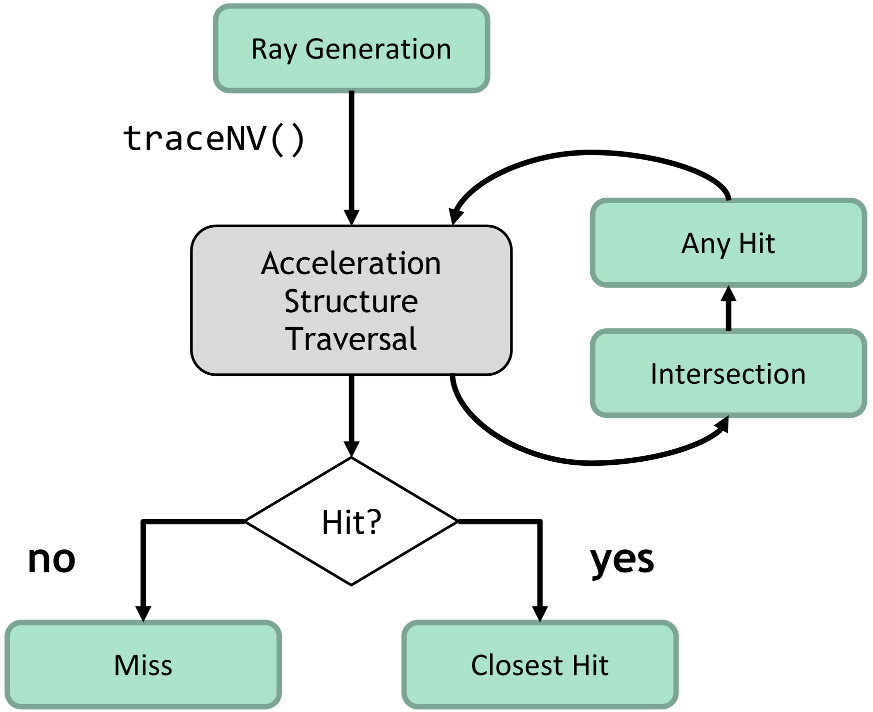
Acceleration Structure Copy

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

```
vkCmdCopyAccelerationStructure(
  VkCommandBuffer cmdBuf,
  VkAccelerationStructure dst,
  VkAccelerationStructure src,
  VkCopyAccelerationStructureFlags flags
);
```

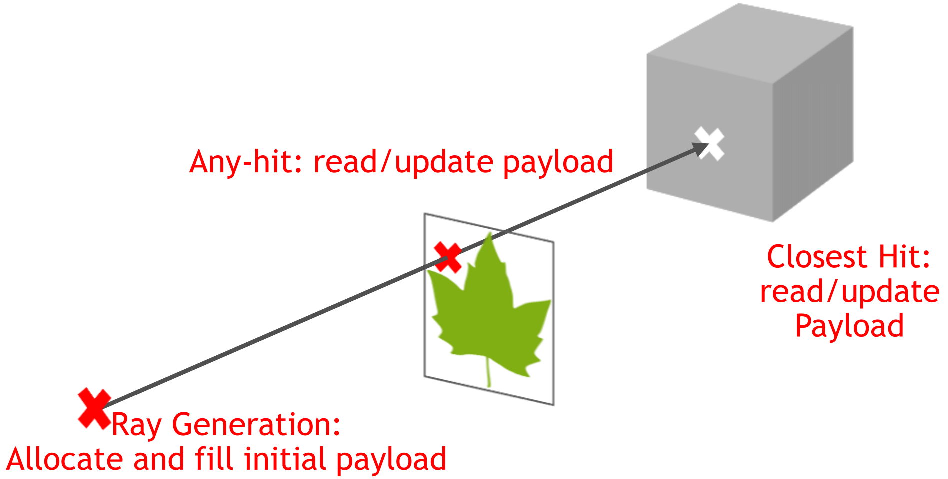
Ray tracing shader domains

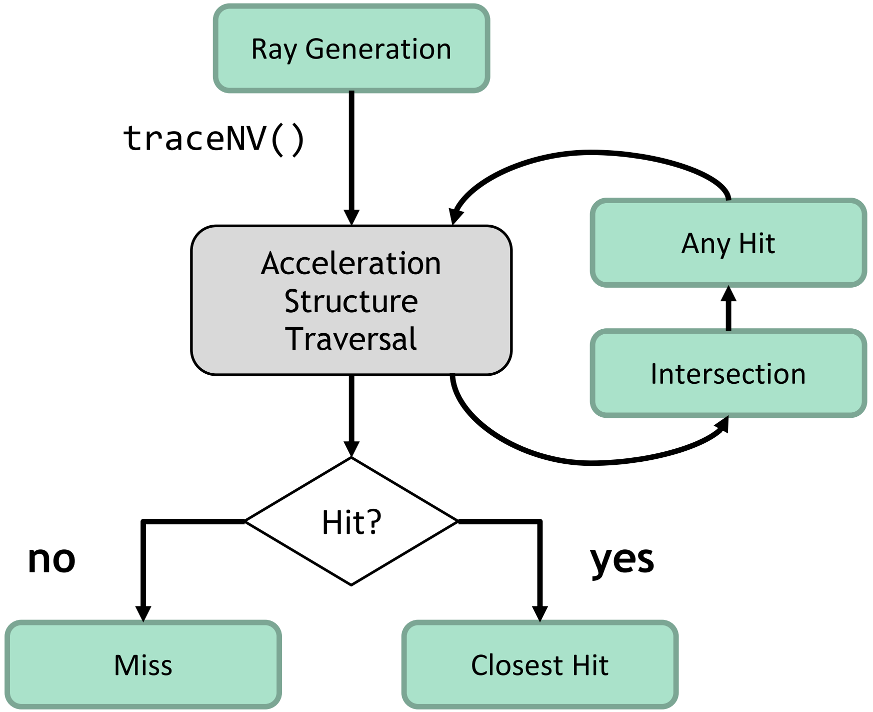




Inter-shader communication

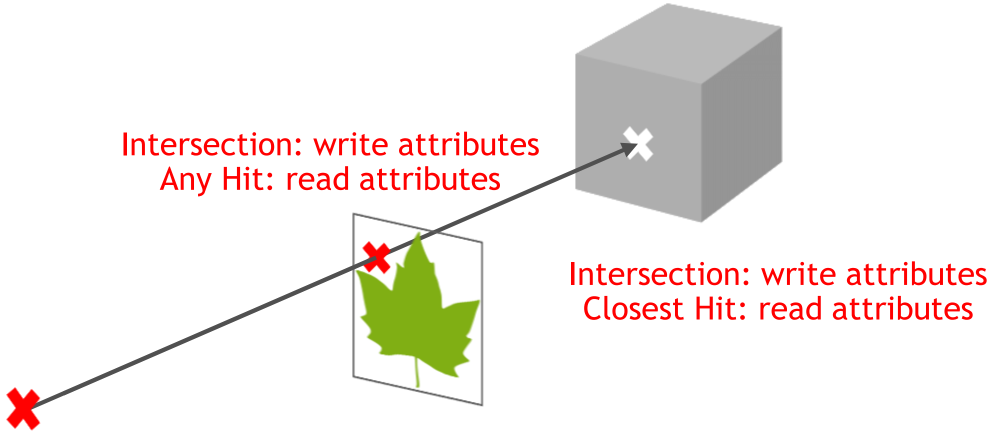
- Ray payload
  - Application-defined structure to pass data between hit stages and shader stage that spawned a ray
  - Used to return final intersection information to ray generation shader

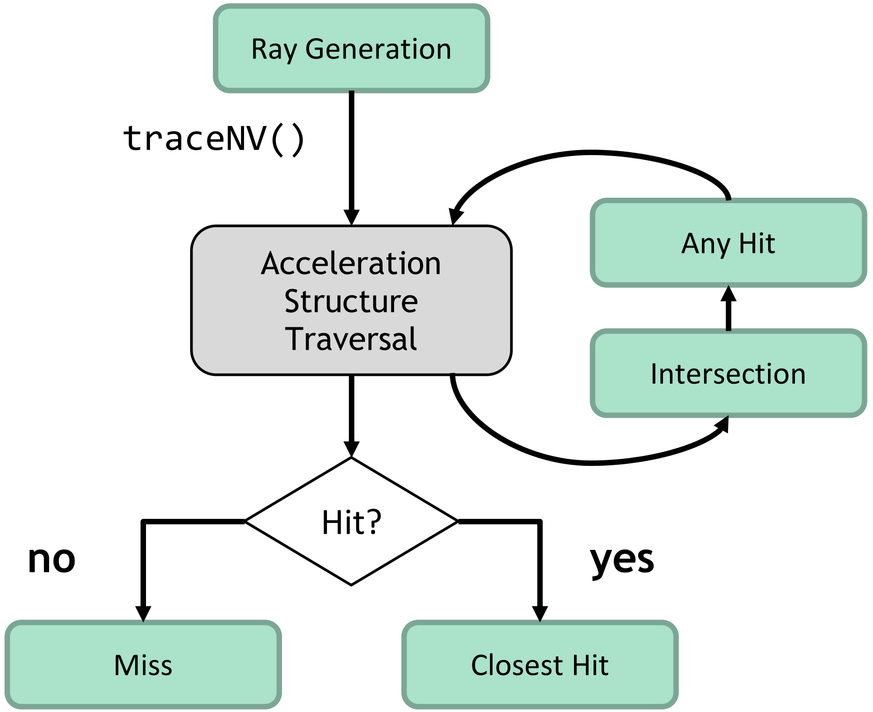




Inter-shader communication

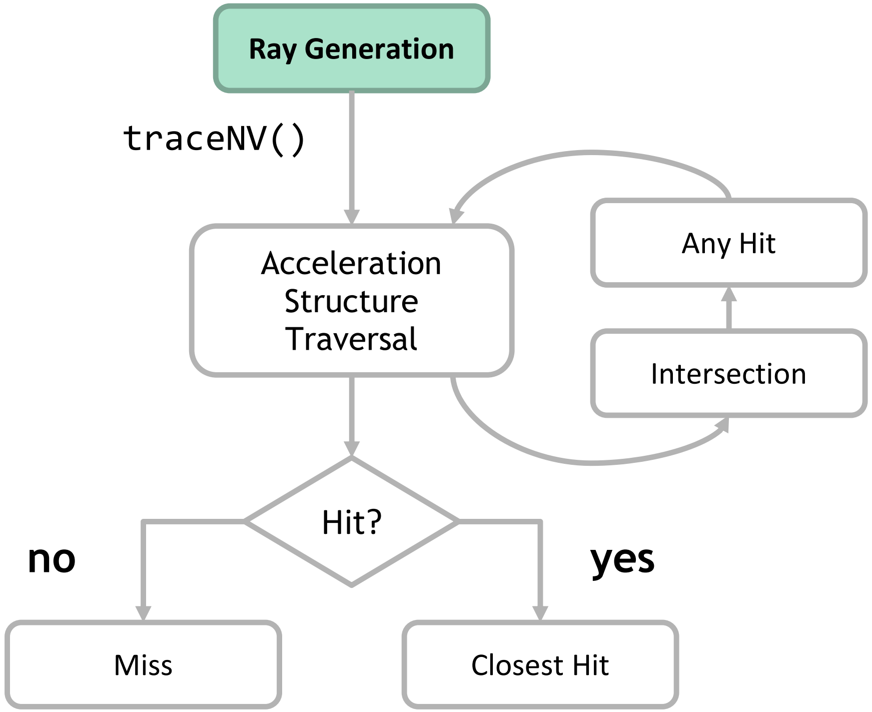
- **Ray Attributes**
  - Application-defined structure to pass intersection information from intersection shader to hit shaders





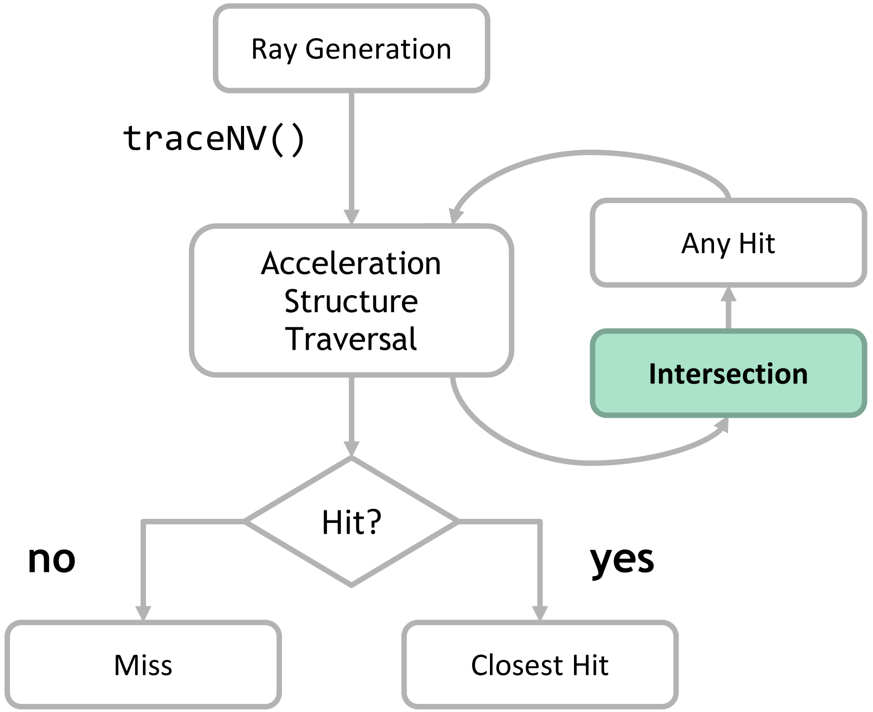
Ray Generation Shaders

- Starting point for all ray tracing work
- Simple 2D grid of threads launched from host
- Traces rays, writes final output



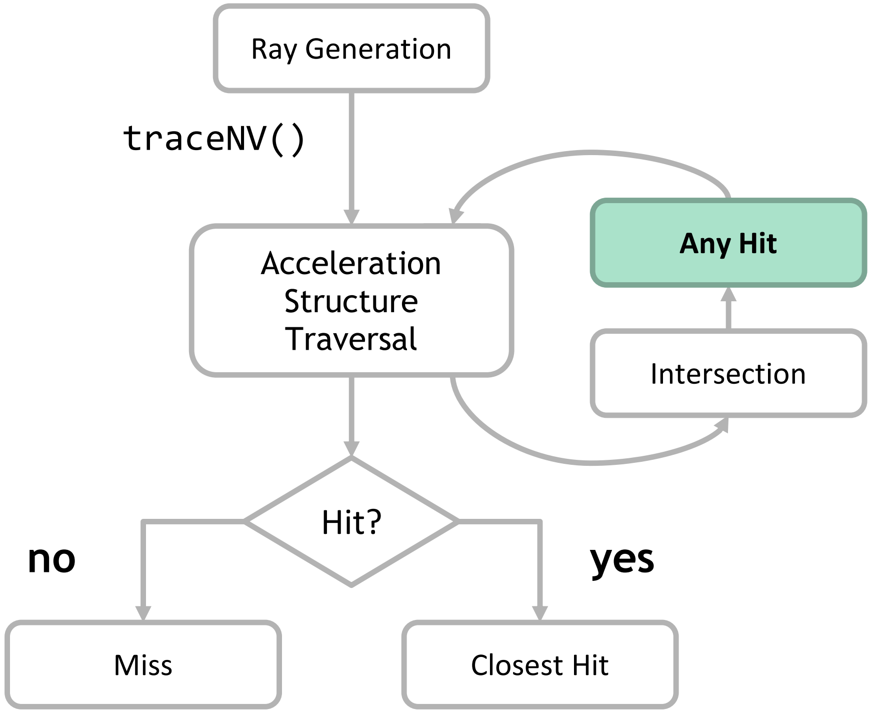
Intersection Shaders

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



Any-hit shader

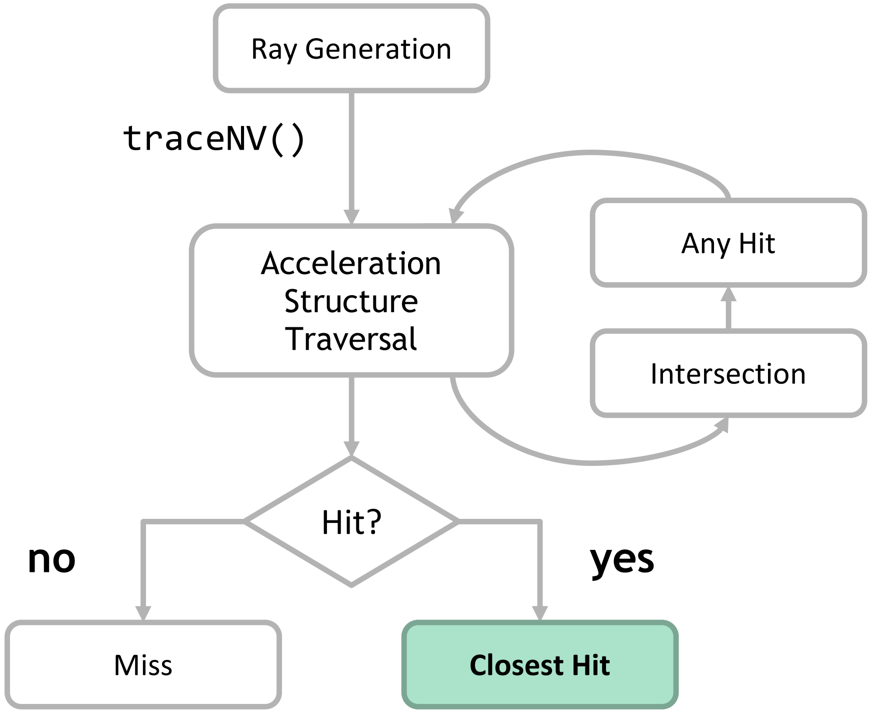
- Invoked after an intersection is found
- Multiple intersections invoked in arbitrary order



Closest-hit shader

- Invoked on the closest intersection of the ray

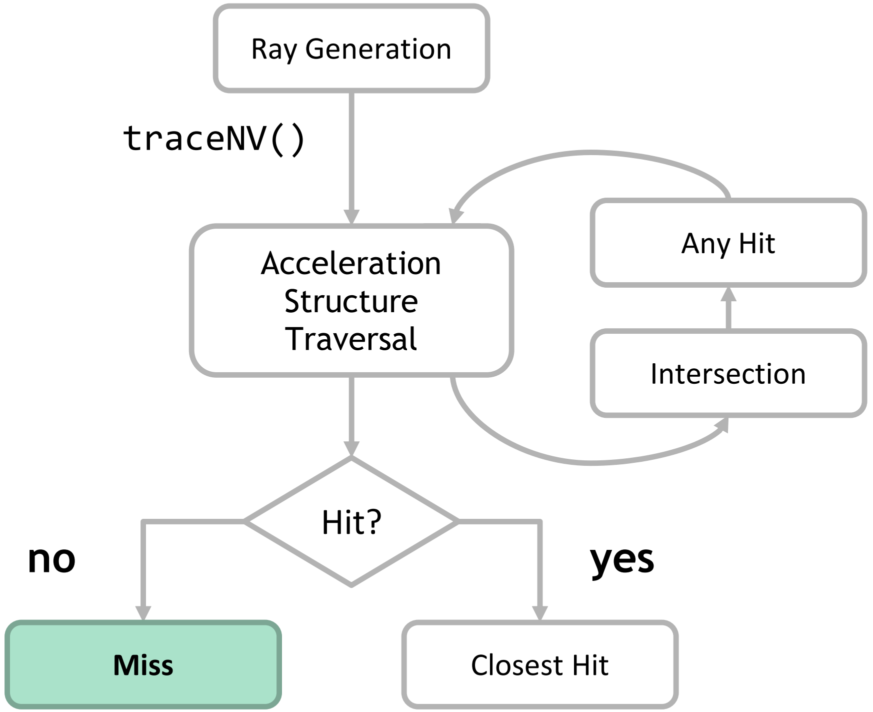
- Can read attributes and trace rays to modify the payload



Miss shader

- Invoked if no hit is found and accepted

- Can trace rays and modify ray payload



Inter-shader Interface Mapping to GLSL

- Ray payload and intersection attributes: new block decorations

```
rayPayloadNV { … }
intersectionAttributesNV { … }
```

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

```
layout (location= 1)
rayPayloadNV firstInterface {…}

layout (location= 2)
rayPayloadNV secondInterface {…}

traceNV(…, 1) // firstInterface
traceNV(…, 2) // secondInterface
```

GLSL Extensions

- 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

```
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();
```

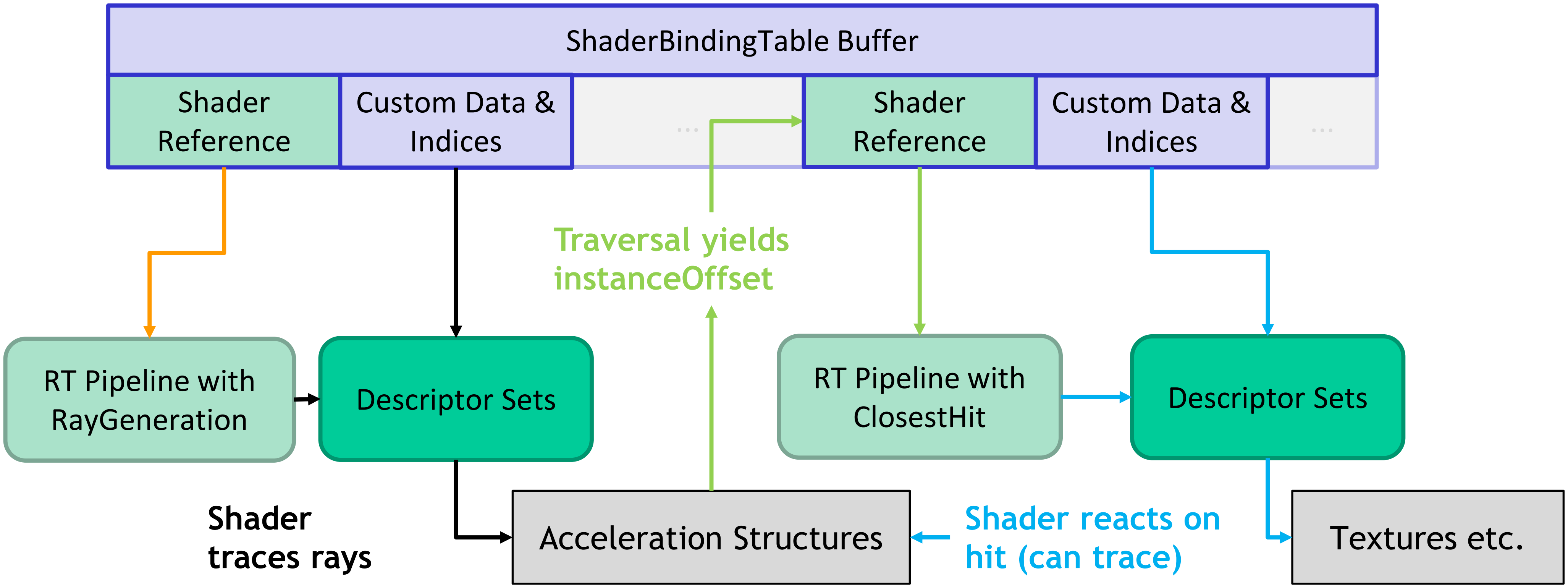
Shader Binding Table

- 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



vkCmdTraceRaysNV

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

```
vkCmdTraceRaysNV(
  VkCommandBuffer cmdBuf,
  VkBuffer shaderBindingTable,
  uint32_t raygenShaderBindingTableIndex,
  uint32_t missShaderBindingTableIndex,
  uint32_t hitShaderBindingTableIndex,
  uint32_t width, uint32_t height
);
```

Ray Tracing Pipeline Creation

- Ray tracing pipeline contains all shaders used together
- Can have multiple shaders of the same stage

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

  VkPipelineCreateFlags flags;
  uint32_t stageCount;
  const
  VkPipelineShaderStageCreateInfo *stages;
  VkPipelineLayout layout;
  size_t shaderBindingTableEntrySize;
  VkPipeline basePipelineHandle;
  int32_t basePipelineIndex;
};

vkCreateRaytracingPipelinesNV(…);
```

Ray Tracing Shader Handles

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

```
vkGetRaytracingShaderHandleNV(
  VkDevice device,
  VkRaytracingPipeline pipeline,
  uint32_t firstShader,
  uint32_t shaderCount,
  size_t dataSize,
  void *pData
);
```

Example Ray Generation Shader

```
// 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
// Same layout as raygen, but just in the default location
// Closest hit shaders only have one payload -
// for the incoming ray
layout rayPayloadNV 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 Ray Generation 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, …);
}
```

Conclusion

- RTX is coming to Vulkan!
- Ray tracing API design proposal offered to Khronos
  - NVIDIA fully committed to working with Khronos on multi-vendor standardization
  - Similar structure to Microsoft’s DXR, helps industry convergence
- Current design is still a work-in-progress