Scalable Real-Time Ray Tracing

Adam Lake
Software Engineer, GPU Rendering

03/2023

Intel® Arc A-series Ray Tracing Architecture: Mapping the DXR API to hardware

PART I

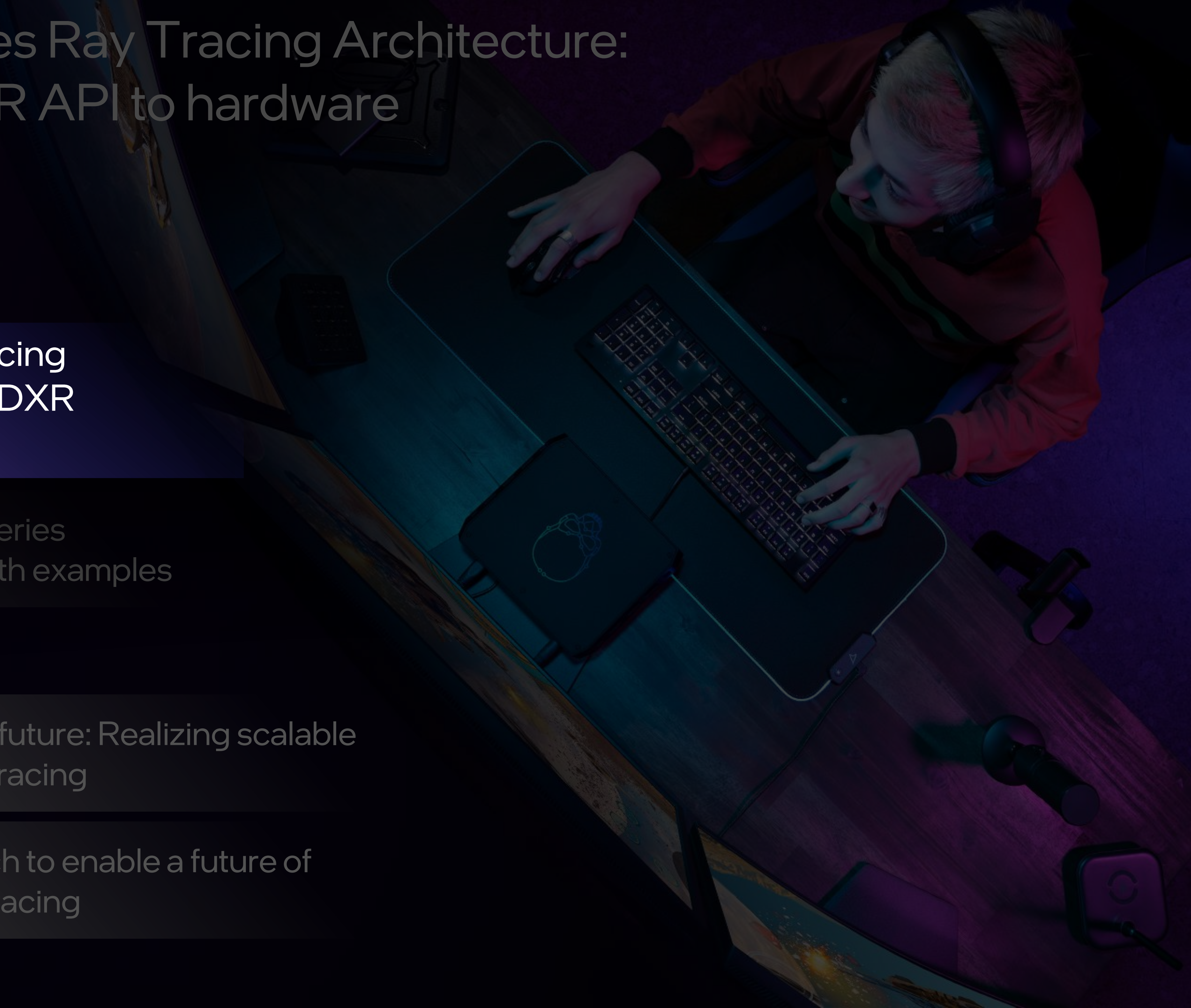
Intel® Arc A-series Ray Tracing Architecture: Mapping the DXR API to hardware

Optimizing for Intel Arc A-series Graphics: Best practices with examples

PART II

Vision for the future: Realizing scalable real-time ray tracing

Intel's research to enable a future of scalable ray tracing



Primer: DXR Programming Model

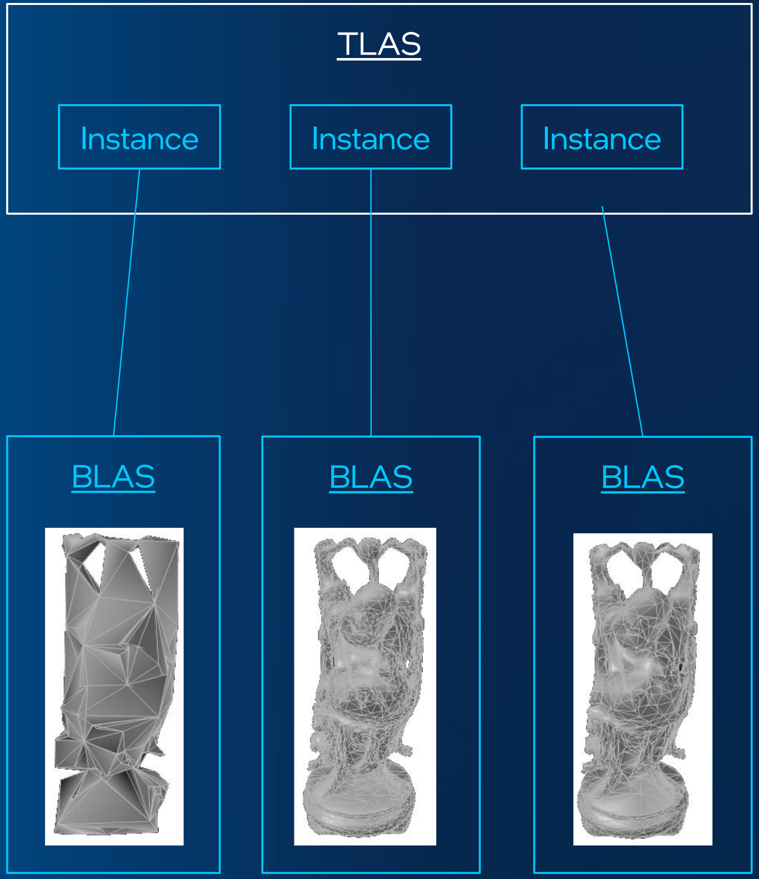
API

- Extend compute shader Dispatch(x,y,z) to include a full set of shader records
  - DispatchRays(x,y,z,shader records)
- Provide geometry database which rays traverse via an acceleration structure
  - top level acceleration structure (TLAS)
  - bottom level acceleration structure (BLAS), geometry represented in a (bounding volume hierarchy) BVH structure
  - State handled via ray tracing pipeline state object (RTPSO)

HLSL

- Shader records to generate rays
  - [shader("raygeneration")]
- Shader records to handle hits and misses of scene geometry
  - [shader("anyhit")], [shader("closesthit")], [shader("miss")]
  - anyhit : every hit (transparent geometry, ...)
  - closesthit: executed once for closest hit
  - miss: ray misses all geometry
  - intersection: user defined primitives
- Indexing of shader records
  - Shader record indexing handled by runtime, see DXR spec and [Usher 2022]

TLAS includes build flags, pointers to BLAS instances



BLAS includes matrix transform, build flags, BVH and geometry

DXR Programming Model: Putting it all together

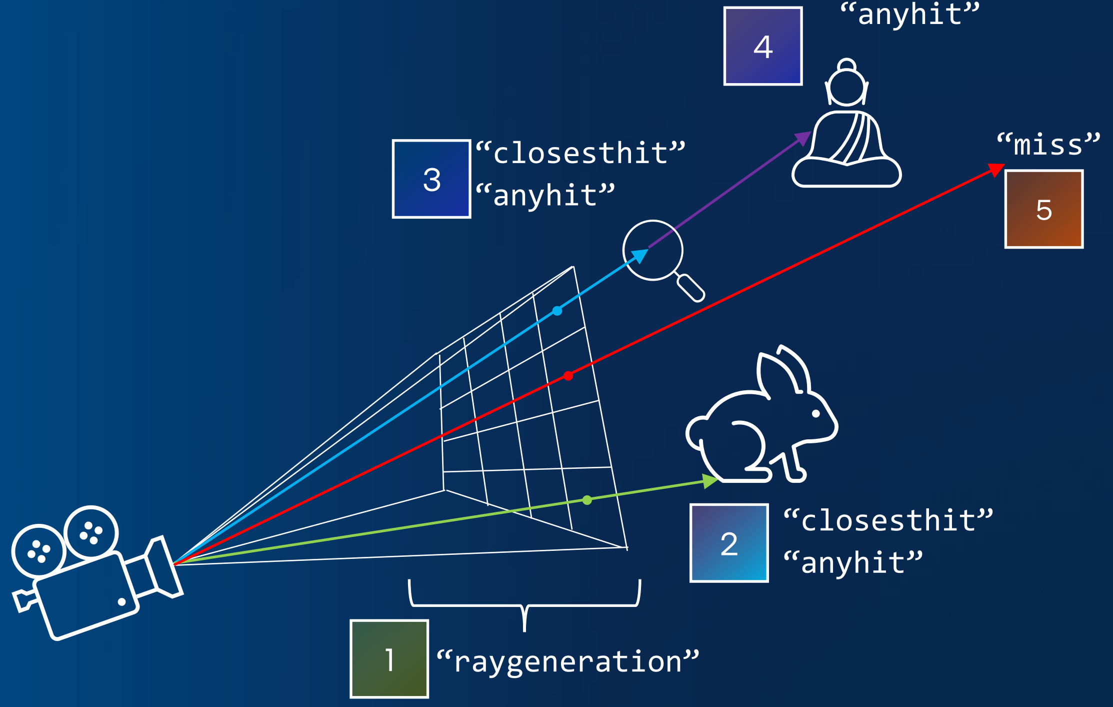
01 Rays generated via *ray generation shader* into the scene geometry

02 Invoke *anyhit* and *closesthit* when acceleration structure traversal determines an intersection

03 Invoke *anyhit* and *closesthit* when acceleration structure traversal determines an intersection, ray continues

04 Since this isn't the closest hit for the ray, only *anyhit* shader invoked

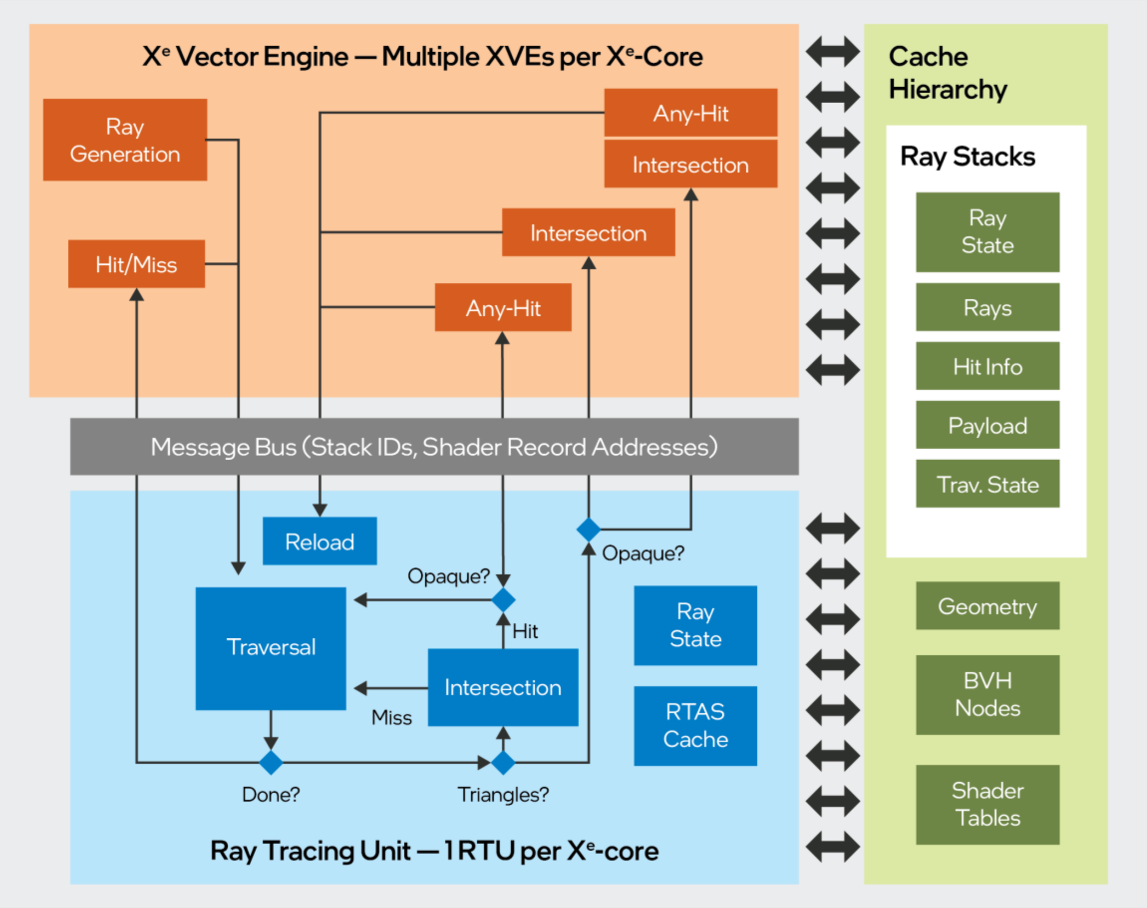
05 No geometry intersected by ray, invoke *miss* shader



Note:
optional to invoke all shader passes, flags to optimize acceleration structure traversal and early exit

Hardware Support for ray tracing

- Orange: shader code authored by programmer, compiled HLSL to the Xe Vector Engine ISA
- Blue: hardware units, programmable by driver but not via API
- Green: state input by application, managed by runtime and hardware units to store state while processing a DispatchRays() call

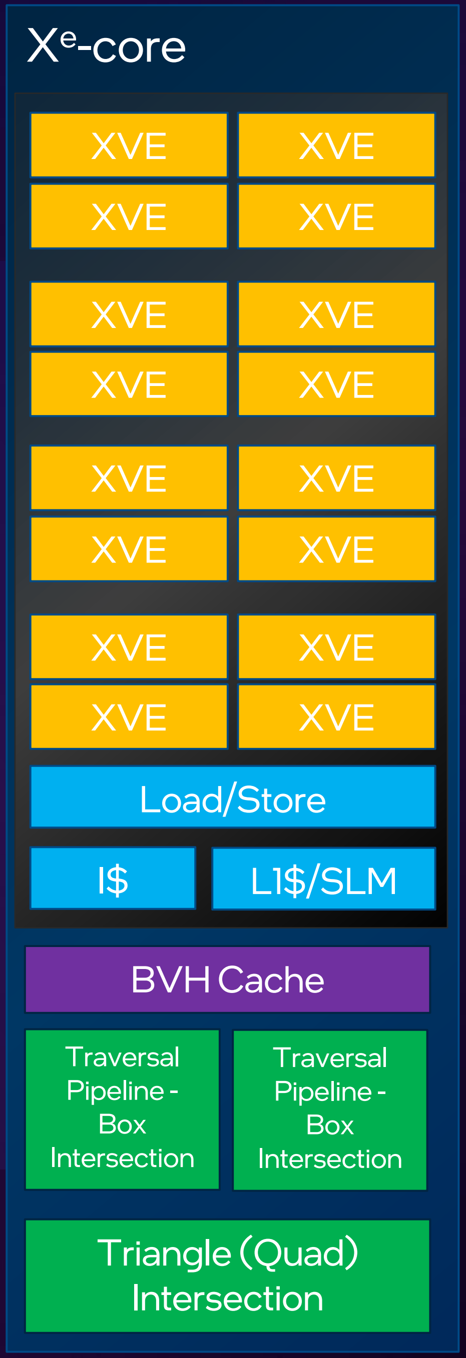


Intel® Arc A-Series support for hardware ray tracing

X^e-core

- 16 Programmable Vector Engines
  - SIMD8 or SIMD16 mode
- Dedicated BVH Cache
  - Fast access to acceleration structure
- 2 Ray Traversal Pipelines combined throughput
  - 1 ray/triangle intersection per clock
  - 12 ray/box intersections per clock
- Accessible via DXR 1.0, DXR 1.1, Vulkan Extensions

Product	Number of X^e-cores/ Ray tracing Units
Arc A310	6
Arc A380	8
Arc A750	28
Arc A770	32



X^e-core

XVE XVE
XVE XVE
XVE XVE
XVE XVE
XVE XVE
XVE XVE
XVE XVE
XVE XVE

Load/Store

I$ L1$/SLM

BVH Cache

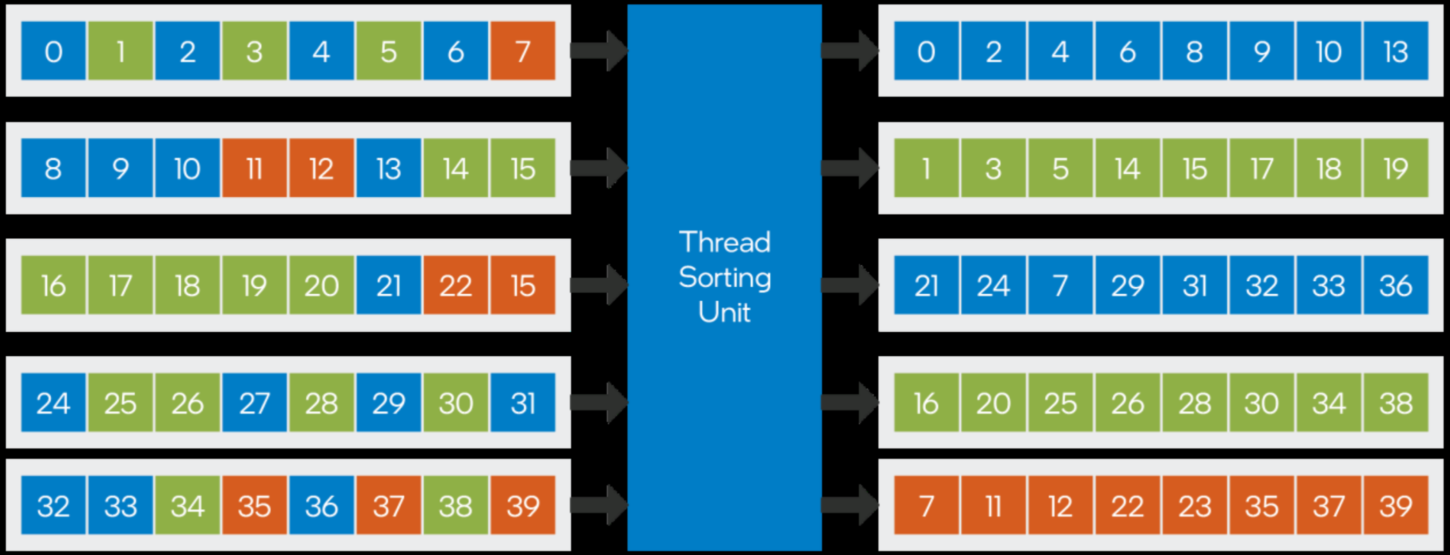
Traversal Pipeline - Box Intersection

Traversal Pipeline - Box Intersection

Triangle (Quad) Intersection

Thread Sorting Unit (TSU)

- Ray traversal in the common case is incoherent: neighboring rays generated during ray generation and subsequent child rays will hit various objects.
- **Thread sorting unit** gathers incoherent rays and repacks rays that share the same shader record index to improve SIMD utilization
- 128 sort keys is upper limit for TSU for no spills



🟦 0	🟩 1	🟦 2	🟩 3	🟦 4	🟩 5	🟦 6	🟧 7
🟦 8	🟦 9	🟦 10	🟧 11	🟧 12	🟦 13	🟩 14	🟩 15
🟩 16	🟩 17	🟩 18	🟩 19	🟩 20	🟦 21	🟧 22	🟧 15
🟦 24	🟩 25	🟩 26	🟦 27	🟩 28	🟦 29	🟩 30	🟦 31
🟦 32	🟦 33	🟩 34	🟧 35	🟦 36	🟧 37	🟩 38	🟧 39

Thread Sorting Unit

🟦 0	🟦 2	🟦 4	🟦 6	🟦 8	🟦 9	🟦 10	🟦 13
🟩 1	🟩 3	🟩 5	🟩 14	🟩 15	🟩 17	🟩 18	🟩 19
🟦 21	🟦 24	🟦 7	🟦 29	🟦 31	🟦 32	🟦 33	🟦 36
🟩 16	🟩 20	🟩 25	🟩 26	🟩 28	🟩 30	🟩 34	🟩 38
🟧 7	🟧 11	🟧 12	🟧 22	🟧 23	🟧 35	🟧 37	🟧 39

Built from ground up with divergent workloads in mind

Journey of a ray through the pipeline: Model vs. Reality

Model	TraceRay()	....

- API exposes the ray traversal as if its one single call and we wait for the return value

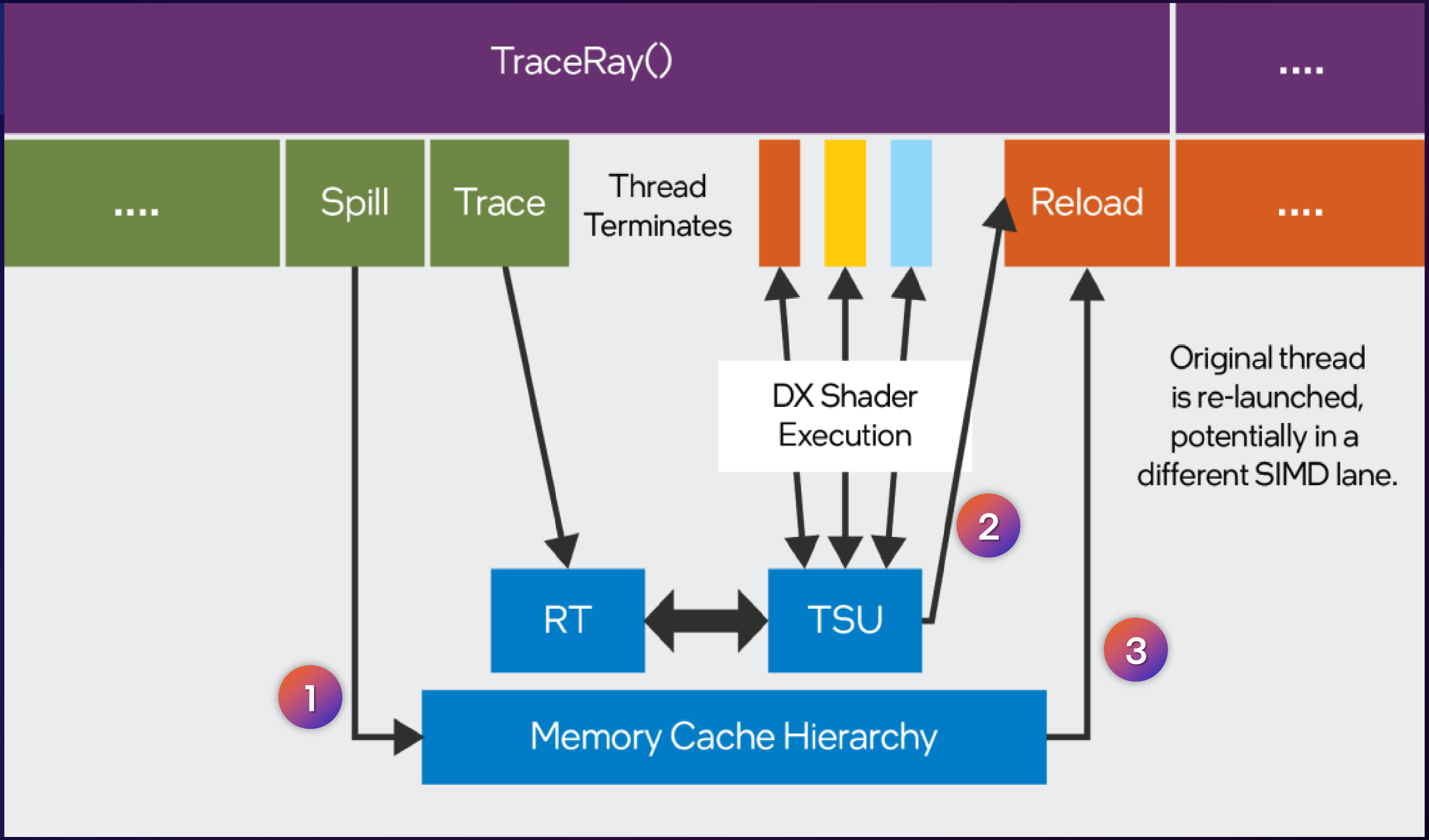
Journey of a ray through the pipeline: Model vs. Reality

Model

- API exposes the ray traversal as if its one single call and we wait for the return value

Reality

- At TraceRay() the thread terminates, state is saved to memory, and traversal through the BVH begins.
- The TSU sorts rays by hit shader record and executes shaders w/ high SIMD utilization.
- After shader execution the state just before TraceRay() is restored and execution continues.

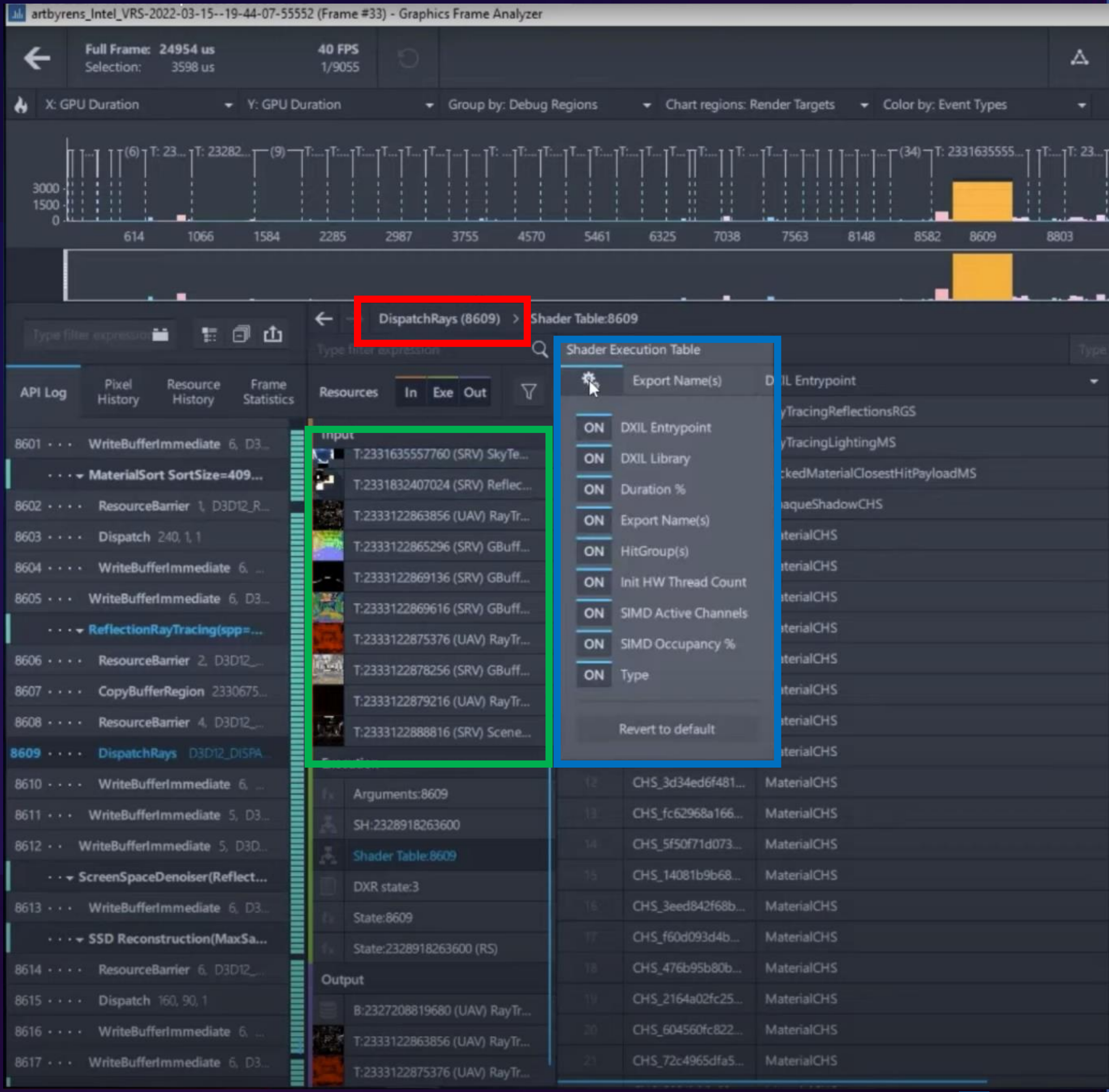


Original thread is re-launched, potentially in a different SIMD lane.

GPA Supports DXR

GPA supports DXR

- Select DispatchRays()
- Visualize:
  - per shader thread occupancy
  - time to execute
  - input and output buffers
  - ...
- More features coming in 2023!



artbyrens_Intel_VRS-2022-03-15--19-44-07-55552 (Frame #33) - Graphics Frame Analyzer

Full Frame: 24954 us
Selection: 3598 us
40 FPS
1/9055

X: GPU Duration · Y: GPU Duration · Group by: Debug Regions · Chart regions: Render Targets · Color by: Event Types

8601 · · · WriteBufferImmediate 6, D3...
· · · MaterialSort SortSize=409...
8602 · · · · ResourceBarrier 1, D3D12_R...
8603 · · · · Dispatch 240, 1, 1
8604 · · · · WriteBufferImmediate 6, ...
8605 · · · WriteBufferImmediate 6, D3...
· · · ReflectionRayTracing(spp=...
8606 · · · · ResourceBarrier 2, D3D12_...
8607 · · · · CopyBufferRegion 2330675...
8608 · · · · ResourceBarrier 4, D3D12_...
8609 · · · · DispatchRays D3D12_DISPA...
8610 · · · · WriteBufferImmediate 6, ...
8611 · · · WriteBufferImmediate 5, D3...
8612 · · · WriteBufferImmediate 5, D3D...
· · · ScreenSpaceDenoiser(Reflect...
8613 · · · WriteBufferImmediate 6, D3...
· · · SSD Reconstruction(MaxSa...
8614 · · · · ResourceBarrier 6, D3D12_...
8615 · · · · Dispatch 160, 90, 1
8616 · · · · WriteBufferImmediate 6, ...
8617 · · · WriteBufferImmediate 6, D3...

Input
T:2331635557760 (SRV) SkyTe...
T:2331832407024 (SRV) Reflec...
T:2333122863856 (UAV) RayTr...
T:2333122865296 (SRV) GBuff...
T:2333122869136 (SRV) GBuff...
T:2333122869616 (SRV) GBuff...
T:2333122875376 (UAV) RayTr...
T:2333122878256 (SRV) GBuff...
T:2333122879216 (UAV) RayTr...
T:2333122888816 (SRV) Scene...
Arguments:8609
SH:2328918263600
Shader Table:8609
DXR state:3
State:8609
State:2328918263600 (RS)

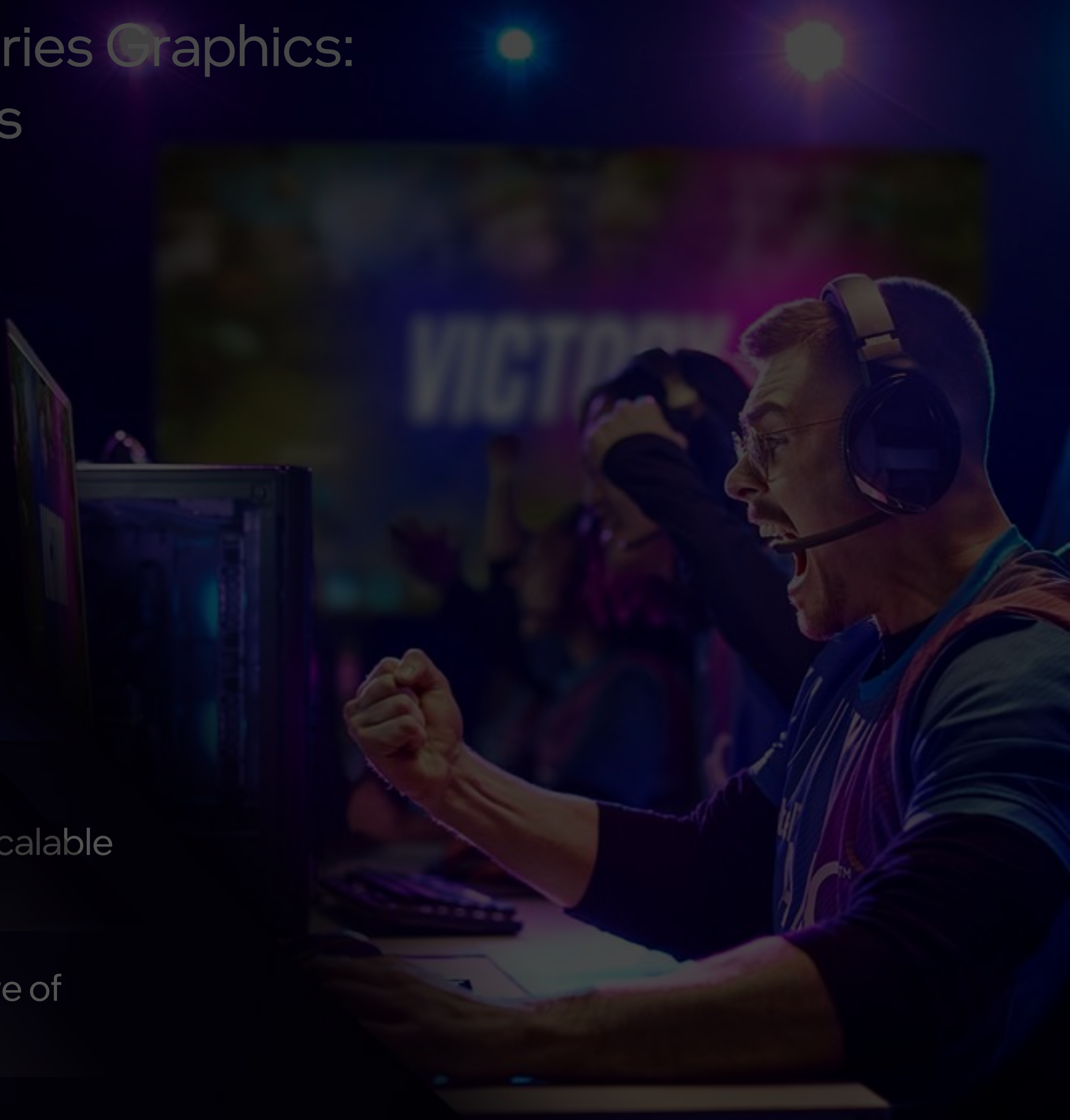
Output
B:2327208819680 (UAV) RayTr...
T:2333122863856 (UAV) RayTr...
T:2333122875376 (UAV) RayTr...

ON DXIL Entrypoint
ON DXIL Library
ON Duration %
ON Export Name(s)
ON HitGroup(s)
ON Init HW Thread Count
ON SIMD Active Channels
ON SIMD Occupancy %
ON Type

yTracingReflectionsRGS
yTracingLightingMS
ckedMaterialClosestHitPayloadMS
aqueShadowCHS
terialCHS
terialCHS
terialCHS
terialCHS
terialCHS
terialCHS
terialCHS
terialCHS

Export Name(s)	DXIL Entrypoint
CHS_3d34ed6f481...	MaterialCHS
CHS_fc62968a166...	MaterialCHS
CHS_5f50f71d073...	MaterialCHS
CHS_14081b9b68...	MaterialCHS
CHS_3eed842f68b...	MaterialCHS
CHS_f60d093d4b...	MaterialCHS
CHS_476b95b80b...	MaterialCHS
CHS_2164a02fc25...	MaterialCHS
CHS_604560fc822...	MaterialCHS
CHS_72c4965dfa5...	MaterialCHS

Optimizing for Intel Arc A-series Graphics: Best practices with examples



PART I

Intel® Arc A-series Ray Tracing Architecture: Mapping the DXR API to hardware

Optimizing for Intel Arc A-series Graphics: Best practices with examples

PART II

Vision for the future: Realizing scalable real-time ray tracing

Intel's research to enable a future of scalable ray tracing

Ray Tracing Optimizations

Optimization of *acceleration structure traversal*

- Improve performance of traversal of the bounding volume hierarchy
- Tradeoffs between efficient updates and traversal

Optimization of *shader execution*

- Tail recursive shaders, single TraceRay() at end of shader vs. multiple TraceRay() calls
- Leverage DXR ray tracing specific flags as input to TraceRay()
- Avoid use of inline RayQueries

Specific examples and more details today

Application Developer's Guide for more details [Barczak 2022]

Acceleration Structure Best Practices: Flag usage

DO use Ray Tracing Accleration Structure (RTAS) build flags

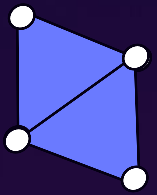
- PREFER_FAST_TRACE for static assets and for top level acceleration structures
- PREFER_FAST _BUILD only if PREFER_FAST_TRACE is too slow

DO NOT USE

- NO_DUPLICATE_ANYHIT_INVOCATION geometry flag
  - Disables some performance optimizations: Requires implementation to guarantee only once per ray anyhit shader invocation
  - Can potentially cause double digit perf loss!
- BUILD_FLAG_ALLOW_UPDATE build flag if it isn't needed
  - Expensive in both memory footprint and performance

Heuristic: Budget for a full TLAS rebuild every frame for stable performance at cost of higher per frame cost

Acceleration Structure Best Practices: Helping triangles form quads



- Use indexed geometry, quad formation does not occur for non-indexed geometry
  - DO NOT use stream output, produces a disconnected mesh
  - Avoid disconnected tris, a mesh with no index reuse will result in no quad formation
- DO perform vertex cache optimization on index buffers, these produce triangle orderings which maximize local vertex re-use. Vertex cache optimization is ideal for quad formation. If possible, optimize for a 16-triangle window.

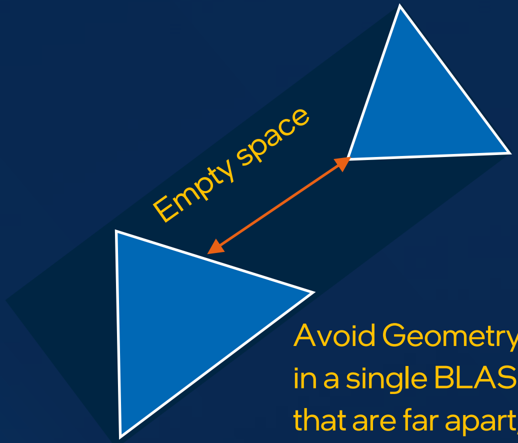
For good vertex cache optimization: *meshoptimizer* by Arseny Kapoulkine: https://github.com/zeux/meshoptimizer

- If GPU generated geometry, do geometry generation in one pass and RTAS construction in second pass, don't interleave because it will serialize the BVH build.

Acceleration Structure Best Practices: Geometry Representation

- DO NOT
  - Duplicate geometry in same location
    - Hide multiple geometry instances out of view in same location
    - Large # of primitives or instances in same position can cause a TDR if even a single ray finds them!
  - Combine geometries in a single BLAS if they are far apart
    - Empty space hurts performance vs. multiple BVHs with less unoccupied space
  - Have multiple instances of smaller BVHs (ie, per material)
    - Instead, these should be part of a single BVH
      - Gives RTAS builder more options for partitioning geometry and eliminating void areas, gains up to 2x
- DO: Avoid long skinny triangles: large bounding box

Avoid Overlapping geometry and long skinny triangles

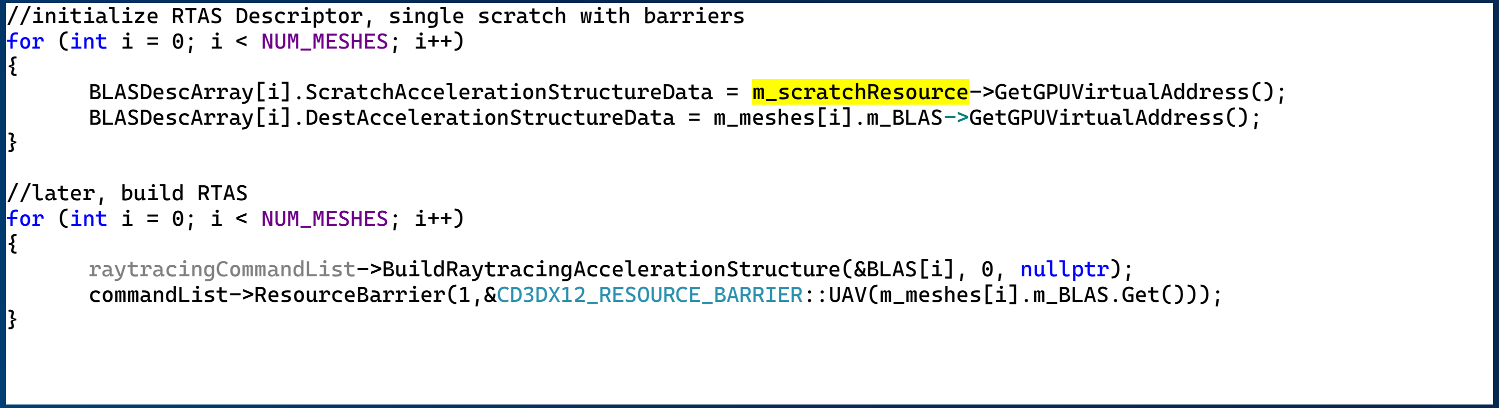


Avoid Geometry in a single BLAS that are far apart, careful with animated objects that separate over time

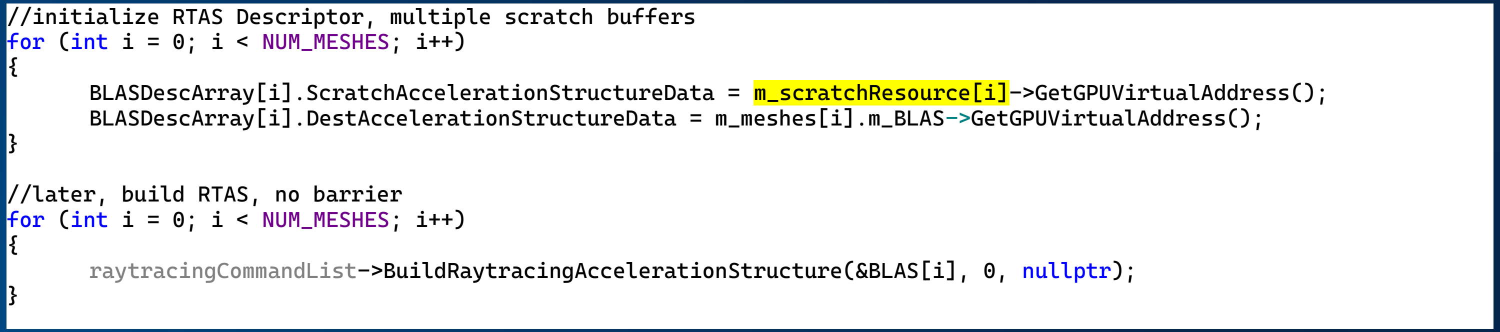
Acceleration Structure Best Practices: Barriers

DO avoid barriers between builds:

- Issue back-to-back builds on non-overlapping scratch memory
- BVH build can then be concurrent vs. serialized with overlapping scratch



```
//initialize RTAS Descriptor, single scratch with barriers
for (int i = 0; i < NUM_MESHES; i++)
{
	BLASDescArray[i].ScratchAccelerationStructureData = m_scratchResource->GetGPUVirtualAddress();
	BLASDescArray[i].DestAccelerationStructureData = m_meshes[i].m_BLAS->GetGPUVirtualAddress();
}

//later, build RTAS
for (int i = 0; i < NUM_MESHES; i++)
{
	raytracingCommandList->BuildRaytracingAccelerationStructure(&BLAS[i], 0, nullptr);
	commandList->ResourceBarrier(1,&CD3DX12_RESOURCE_BARRIER::UAV(m_meshes[i].m_BLAS.Get()));
}
```

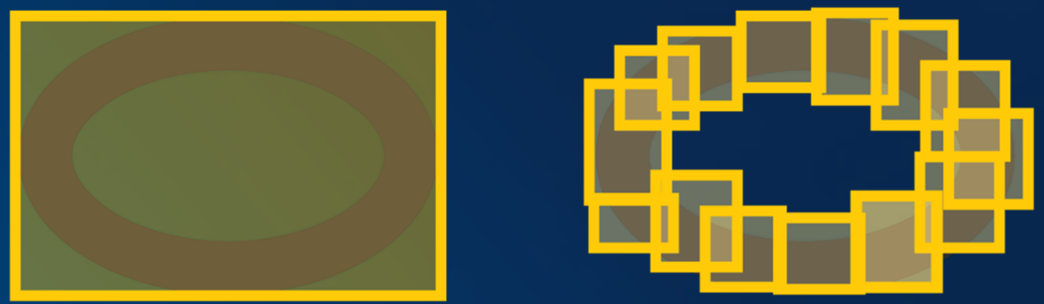


```
//initialize RTAS Descriptor, multiple scratch buffers
for (int i = 0; i < NUM_MESHES; i++)
{
	BLASDescArray[i].ScratchAccelerationStructureData = m_scratchResource[i]->GetGPUVirtualAddress();
	BLASDescArray[i].DestAccelerationStructureData = m_meshes[i].m_BLAS->GetGPUVirtualAddress();
}

//later, build RTAS, no barrier
for (int i = 0; i < NUM_MESHES; i++)
{
	raytracingCommandList->BuildRaytracingAccelerationStructure(&BLAS[i], 0, nullptr);
}
```

Use array of scratch resources and remove barrier vs. a single shared scratch space

Acceleration Structure Best Practices: Alpha tested geometry and hit shaders

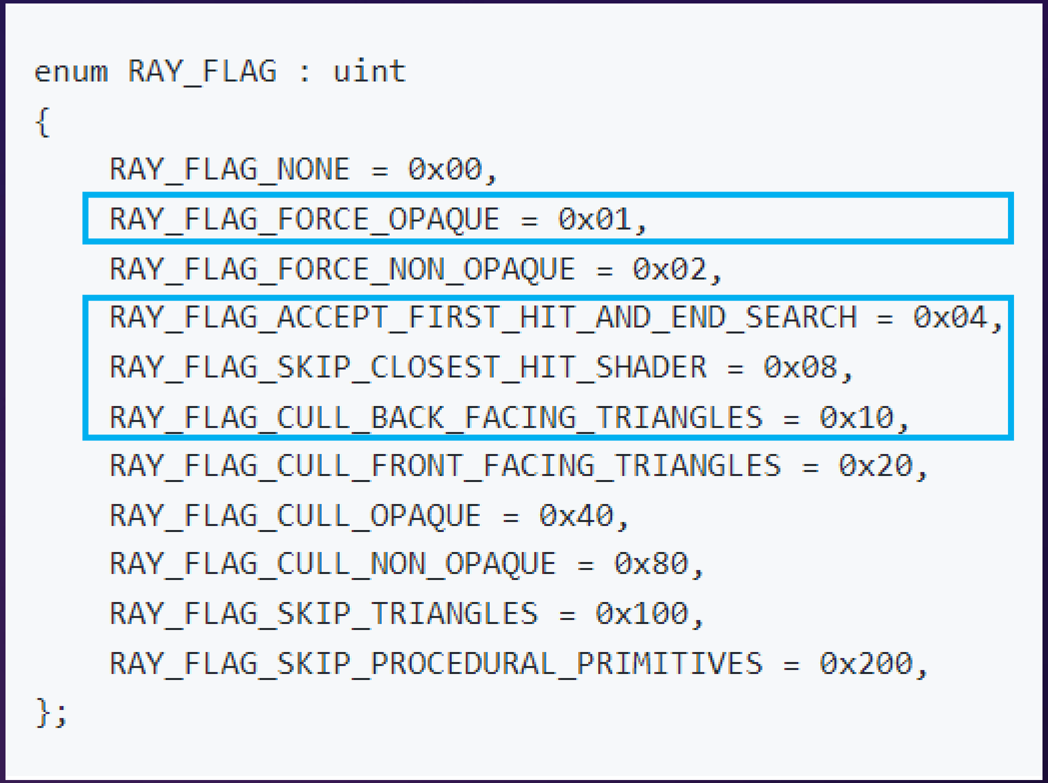
- Good practice for rasterization, even more important for Ray Tracing due to cost of invocation of any hit shaders
- When using alpha tested geometry implemented as textured tris and any hit shaders and the geometry contains large, transparent regions, subdivide the geometry so it tightly bounds the non-transparent region.



Shader Optimizations: Ray Flags

- DO use as many ray flags as possible!
- Pass them in as compile time constants so compiler can see them vs. passing in as constant buffers
- Even if you know your instances are opaque, still use FORCE_OPAQUE
  - We can optimize knowing only 1 anyhit shader invoked
- Don't use FORCE_NON_OPAQUE if you can avoid it. Ray will assume you can have an any hit shader for each triangle and has to check. Only needed for transparent geometry.

Flags of highest value



```
enum RAY_FLAG : uint
{
    RAY_FLAG_NONE = 0x00,
    RAY_FLAG_FORCE_OPAQUE = 0x01,
    RAY_FLAG_FORCE_NON_OPAQUE = 0x02,
    RAY_FLAG_ACCEPT_FIRST_HIT_AND_END_SEARCH = 0x04,
    RAY_FLAG_SKIP_CLOSEST_HIT_SHADER = 0x08,
    RAY_FLAG_CULL_BACK_FACING_TRIANGLES = 0x10,
    RAY_FLAG_CULL_FRONT_FACING_TRIANGLES = 0x20,
    RAY_FLAG_CULL_OPAQUE = 0x40,
    RAY_FLAG_CULL_NON_OPAQUE = 0x80,
    RAY_FLAG_SKIP_TRIANGLES = 0x100,
    RAY_FLAG_SKIP_PROCEDURAL_PRIMITIVES = 0x200,
};
```

Shader optimizations in practice

```
struct MyRayPayload { float3 hit_position, normal, diffuse, hit;};

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  pl.hit_position = ComputeHitPosition();
  pl.normal = ComputeNormal();
  pl.diffuse = ComputeDiffuseColor();
  pl.hit = true;
}

[shader("miss")]
void miss ( inout MyRayPayload pl )
{
  pl.hit = false;
}

RWTexture2D<float3> render_target;

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  float3 color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
  if( pl.hit )
    color += DoLighting( pl.hit_position, pl.normal, pl.diffuse);
  else
    color += ComputeMissColor(ray.Origin,ray.Direction);
  render_target.Store( DispatchRaysIndex().xy, pl.color );
}
```

1

```
struct MyRayPayload {  float16_t3 color; /*lower precision*/ };

RWTexture2D<float3> render_target;

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  float3 hit_position = ComputeHitPosition();
  float3 normal = ComputeNormal();
  float3 diffuse = ComputeDiffuseColor();
  float3 lighting = DoLighting(hit_position, normal, diffuse);
  render_target.Store( DispatchRaysIndex().xy, pl.color + lighting );
}

[shader("miss")]
void miss( inout MyRayPayload pl )
{
  float3 miss_color = ComputeMissColor(WorldRayOrigin(),WorldRayDirection());
  render_target.Store( DispatchRaysIndex().xy, pl.color + miss_color );
}

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  pl.color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
}
```

Shader optimizations in practice

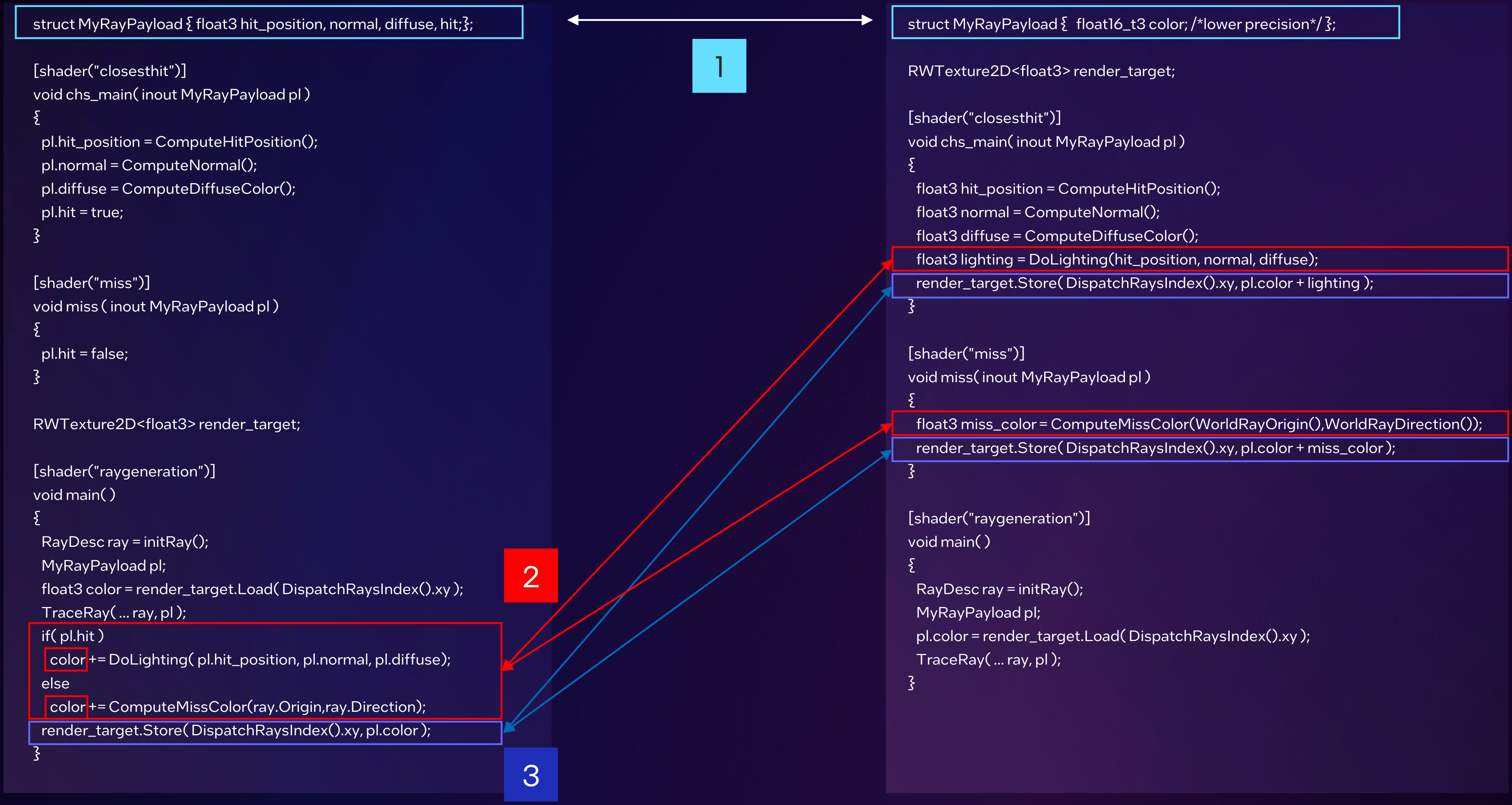
```
struct MyRayPayload { float3 hit_position, normal, diffuse, hit;};

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  pl.hit_position = ComputeHitPosition();
  pl.normal = ComputeNormal();
  pl.diffuse = ComputeDiffuseColor();
  pl.hit = true;
}

[shader("miss")]
void miss ( inout MyRayPayload pl )
{
  pl.hit = false;
}

RWTexture2D<float3> render_target;

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  float3 color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
  if( pl.hit )
   color += DoLighting( pl.hit_position, pl.normal, pl.diffuse);
  else
   color += ComputeMissColor(ray.Origin,ray.Direction);
  render_target.Store( DispatchRaysIndex().xy, pl.color );
}
```

1

2

```
struct MyRayPayload {  float16_t3 color; /*lower precision*/ };

RWTexture2D<float3> render_target;

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  float3 hit_position = ComputeHitPosition();
  float3 normal = ComputeNormal();
  float3 diffuse = ComputeDiffuseColor();
  float3 lighting = DoLighting(hit_position, normal, diffuse);
  render_target.Store( DispatchRaysIndex().xy, pl.color + lighting );
}

[shader("miss")]
void miss( inout MyRayPayload pl )
{
  float3 miss_color = ComputeMissColor(WorldRayOrigin(),WorldRayDirection());
  render_target.Store( DispatchRaysIndex().xy, pl.color + miss_color );
}

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  pl.color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
}
```

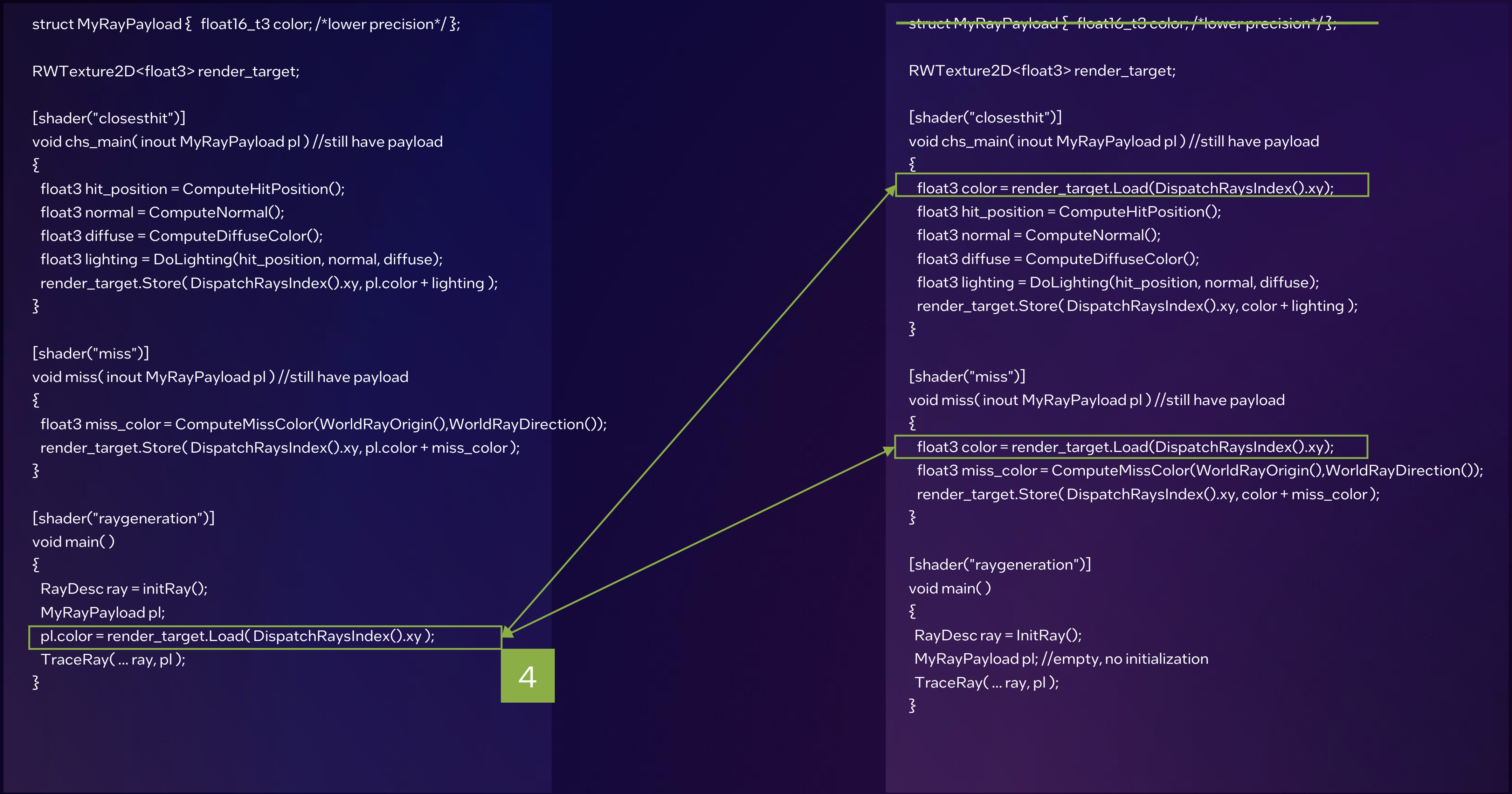
Shader optimizations in practice



```
struct MyRayPayload { float3 hit_position, normal, diffuse, hit;};

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  pl.hit_position = ComputeHitPosition();
  pl.normal = ComputeNormal();
  pl.diffuse = ComputeDiffuseColor();
  pl.hit = true;
}

[shader("miss")]
void miss ( inout MyRayPayload pl )
{
  pl.hit = false;
}

RWTexture2D<float3> render_target;

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  float3 color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
  if( pl.hit )
   color += DoLighting( pl.hit_position, pl.normal, pl.diffuse);
  else
   color += ComputeMissColor(ray.Origin,ray.Direction);
  render_target.Store( DispatchRaysIndex().xy, pl.color );
}
```

```
struct MyRayPayload {  float16_t3 color; /*lower precision*/ };

RWTexture2D<float3> render_target;

[shader("closesthit")]
void chs_main( inout MyRayPayload pl )
{
  float3 hit_position = ComputeHitPosition();
  float3 normal = ComputeNormal();
  float3 diffuse = ComputeDiffuseColor();
  float3 lighting = DoLighting(hit_position, normal, diffuse);
  render_target.Store( DispatchRaysIndex().xy, pl.color + lighting );
}

[shader("miss")]
void miss( inout MyRayPayload pl )
{
  float3 miss_color = ComputeMissColor(WorldRayOrigin(),WorldRayDirection());
  render_target.Store( DispatchRaysIndex().xy, pl.color + miss_color );
}

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  pl.color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
}
```

Shader optimizations in practice



```
struct MyRayPayload {  float16_t3 color; /*lower precision*/ };

RWTexture2D<float3> render_target;

[shader("closesthit")]
void chs_main( inout MyRayPayload pl ) //still have payload
{
  float3 hit_position = ComputeHitPosition();
  float3 normal = ComputeNormal();
  float3 diffuse = ComputeDiffuseColor();
  float3 lighting = DoLighting(hit_position, normal, diffuse);
  render_target.Store( DispatchRaysIndex().xy, pl.color + lighting );
}

[shader("miss")]
void miss( inout MyRayPayload pl ) //still have payload
{
  float3 miss_color = ComputeMissColor(WorldRayOrigin(),WorldRayDirection());
  render_target.Store( DispatchRaysIndex().xy, pl.color + miss_color );
}

[shader("raygeneration")]
void main( )
{
  RayDesc ray = initRay();
  MyRayPayload pl;
  pl.color = render_target.Load( DispatchRaysIndex().xy );
  TraceRay( ... ray, pl );
}
```

```
~~struct MyRayPayload {  float16_t3 color; /*lower precision*/ };~~

RWTexture2D<float3> render_target;

[shader("closesthit")]
void chs_main( inout MyRayPayload pl ) //still have payload
{
  float3 color = render_target.Load(DispatchRaysIndex().xy);
  float3 hit_position = ComputeHitPosition();
  float3 normal = ComputeNormal();
  float3 diffuse = ComputeDiffuseColor();
  float3 lighting = DoLighting(hit_position, normal, diffuse);
  render_target.Store( DispatchRaysIndex().xy, color + lighting );
}

[shader("miss")]
void miss( inout MyRayPayload pl ) //still have payload
{
  float3 color = render_target.Load(DispatchRaysIndex().xy);
  float3 miss_color = ComputeMissColor(WorldRayOrigin(),WorldRayDirection());
  render_target.Store( DispatchRaysIndex().xy, color + miss_color );
}

[shader("raygeneration")]
void main( )
{
 RayDesc ray = InitRay();
 MyRayPayload pl; //empty, no initialization
 TraceRay( ... ray, pl );
}
```

Real Workloads

1. Tail Recursion: ~9.5% increase* in performance by making the global illumination (GI) specular shader tail recursive
2. Minimizing spill data: ~16.7% increase* in performance in GI renderer

- Bonus!

3. Replacing loop iterators as constants visible to compiler:
   - ~17% increase* in performance in shadow pass
   - ~28% increase* in global illumination renderer

[Intel Sponza Scene 2022]

Instead of this:

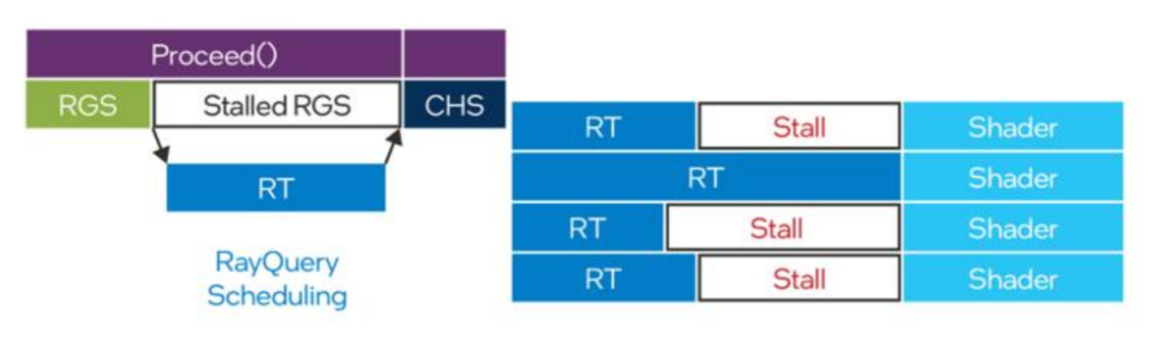
- `for(int i=0;i<constant_buffer.loop_count;i++)`

Do this:

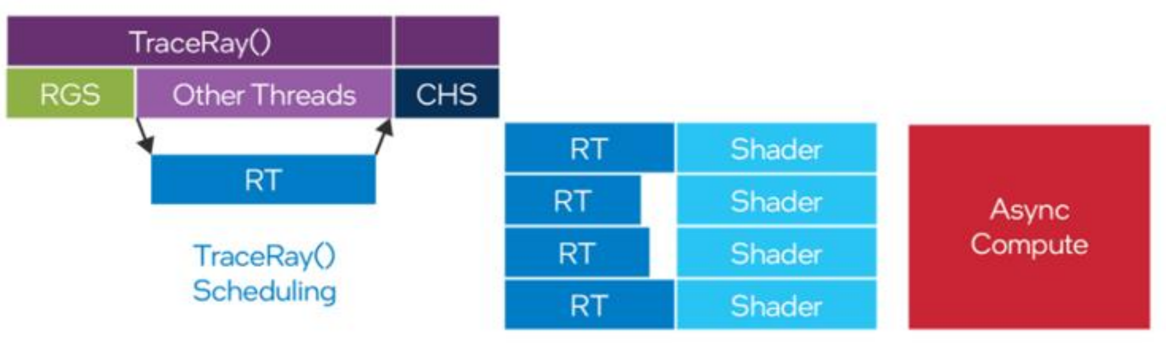
- On CPU side, add a define to the dxc shader compilation for DXR:
  - `dxc_shader.add_define(“LOOP_COUNT”, “1”)`
- In HLSL: modify for loop to look like this:
  - `for(int i=0;i<LOOP_COUNT;i++)`

DXR 1.0 TraceRay() vs. DXR 1.1 RayQuery()

- DXR 1.1 introduced a synchronous form of ray tracing by using ray queries from any shader.
- These incoherent ray queries mean we cannot use the TSU
- Longest-running ray query requires the thread executing a SIMD set of rays to also occupy a thread slot while waiting for the longest running thread.



Problem: Longest-running ray determines performance of all rays in the thread



Solution: TraceRay() exits thread, allowing other threads to make progress

DXR 1.1: Best practices for RayQuery()

- Don't assume a SIMD32 wave!
  - Write shader code that does not assume a SIMD32 wide wave
    - Intel compiles to an optimal SIMD size (SIMD8, SIMD16, or SIMD32) based on register pressure, **`groupshared`** memory utilization, etc.
  - One way to determine if SIMD size less than 32 is supported:
    - **`D3D12_FEATURE_DATA_D3D12_OPTIONS1.WaveLaneCountMin`**
  - If wave size of 32 is *required*
    - **`WaveSize<numLanes>`** attribute in Shader Model 6.6 to compile 32 wide wave

- Use wave intrinsics instead of barriers
  - Barriers can force *all* waves to wait for longest running wave

Asynchronous Compute and Ray Tracing

- `groupshared` (i.e., local memory) and ray tracing hardware acceleration leverages the same L1 cache
- Therefore, to ensure better performance, avoid `groupshared` memory
  - DXR 1.0 `TraceRay()`
    - In compute shaders that are scheduled concurrently
  - DXR 1.1 `RayQuery()`
    - In compute shaders that are scheduled concurrently
    - In the compute shader issuing the `RayQuery()`

Bonus! More Shader Optimizations…

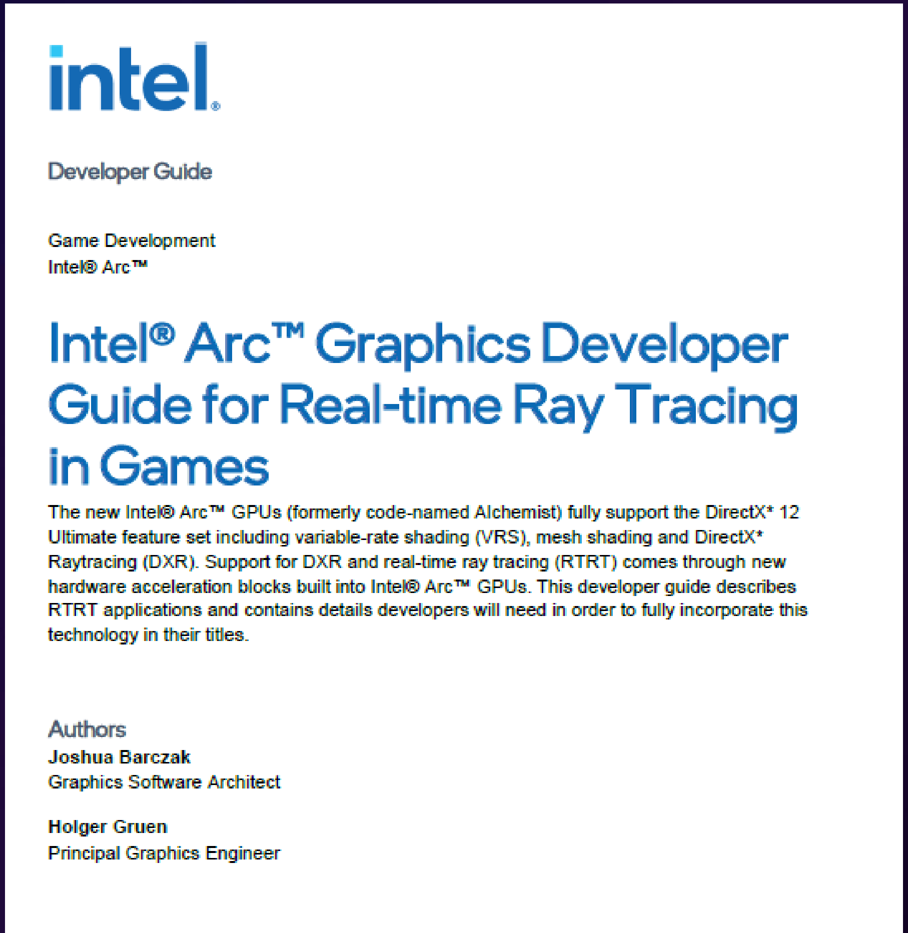
- **TraceRay optimizations**
  - Use only one TraceRay() call per shader
  - Use only one ray generation shader per RTPSO, ideally with only one TraceRay() call
- **Ray Payload Optimizations**
  - Set maximum payload size and hit shader attribute counts as low as possible
  - Pass ray payloads by reference instead of copying them Payloads are stored in memory, reads and writes compile to loads and stores.
  - Keep payloads as small as possible, store data as unorm or snorm when possible and use ALU instructions for conversions.
  - In general, prefer recompute to load/store!
  - Don't put things in ray payloads you can get from other places
    - Ray data, global constants, …
  - Do not initialize or rely on initialized payloads before tracing
- **Compilation and shader management**
  - Avoid including shaders that will not be used in a DispatchRays()
  - Create separate RTPSOs() for each pass (shadow, AO, reflections, GI, …), otherwise, compiler assumes all shaders might be used at once

- **Shader Optimizations**
  - Set recursion depth to 1 if you do not use recursion
  - If your shaders don't use local arguments, use same table record by setting shader table stride to 0.
  - In ray generation shaders, try to have rays emitted by direction/origin, better cache coherency, BVH node hits in BVH cache can increase performance, could be as much as ~2-3x in some cases!
  - If possible, avoid any hit shaders
  - Do not use as an uber shader with a switch by ray type
    - Use instance masking and ray flags instead
    - Use only when there are no alternatives
  - DO use miss shaders vs. putting the skydome into the acceleration structure, Allows a miss shader to avoid traversing to leaf nodes of the BVH

I can’t remember all of these!

See our Optimization Guide:

Intel® Arc™ Graphics Developer Guide for Real-time Ray Tracing in Games



intel.

Developer Guide

Game Development
Intel® Arc™

Intel® Arc™ Graphics Developer Guide for Real-time Ray Tracing in Games

The new Intel® Arc™ GPUs (formerly code-named Alchemist) fully support the DirectX* 12 Ultimate feature set including variable-rate shading (VRS), mesh shading and DirectX* Raytracing (DXR). Support for DXR and real-time ray tracing (RTRT) comes through new hardware acceleration blocks built into Intel® Arc™ GPUs. This developer guide describes RTRT applications and contains details developers will need in order to fully incorporate this technology in their titles.

Authors

Joshua Barczak
Graphics Software Architect

Holger Gruen
Principal Graphics Engineer

Vision for the future:
Realizing scalable real-time ray tracing

PART I

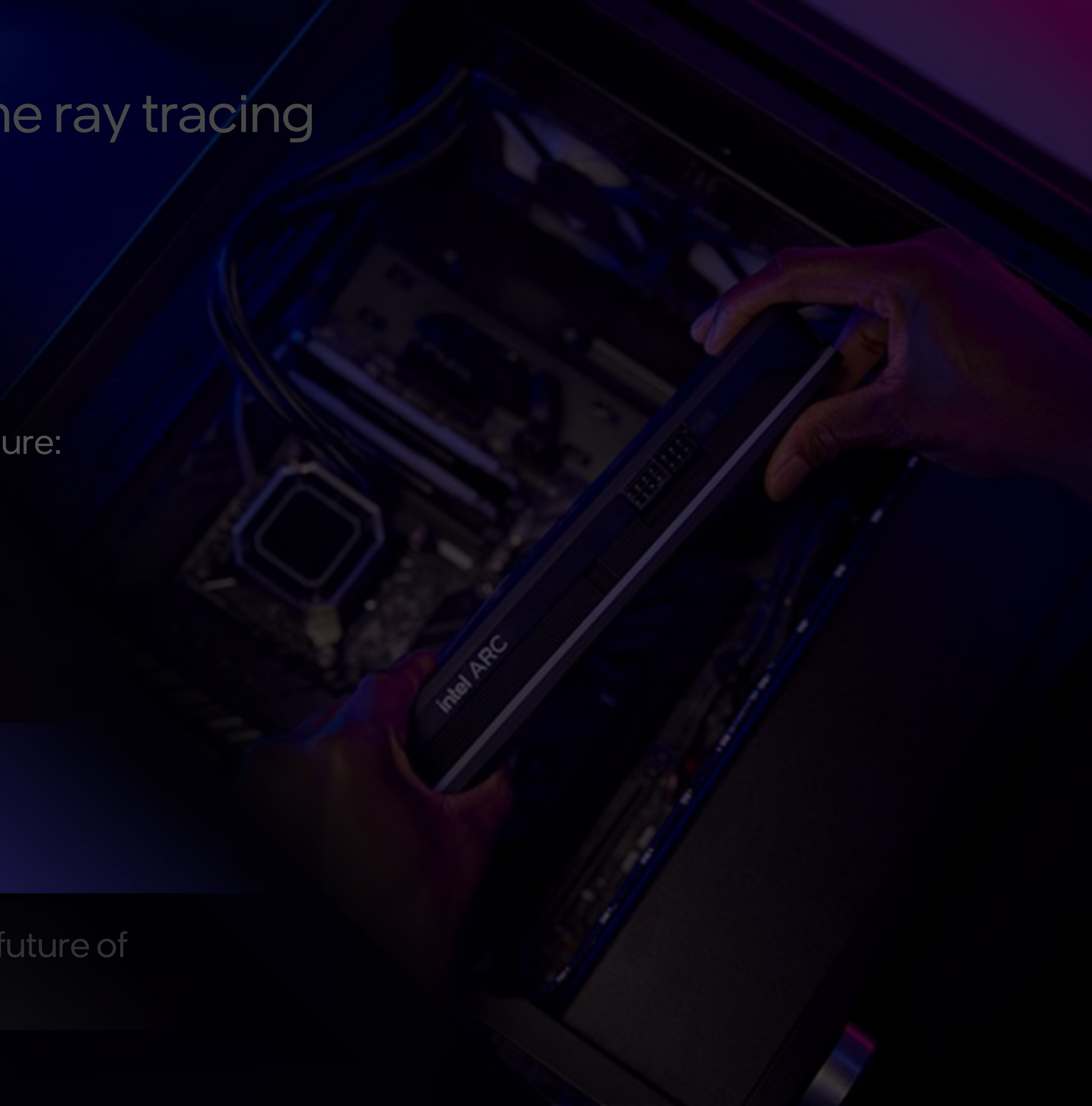
Intel® Arc A-series Ray Tracing Architecture: Mapping the DXR API to hardware

Optimizing for Intel Arc A-series Graphics: Best practices with examples

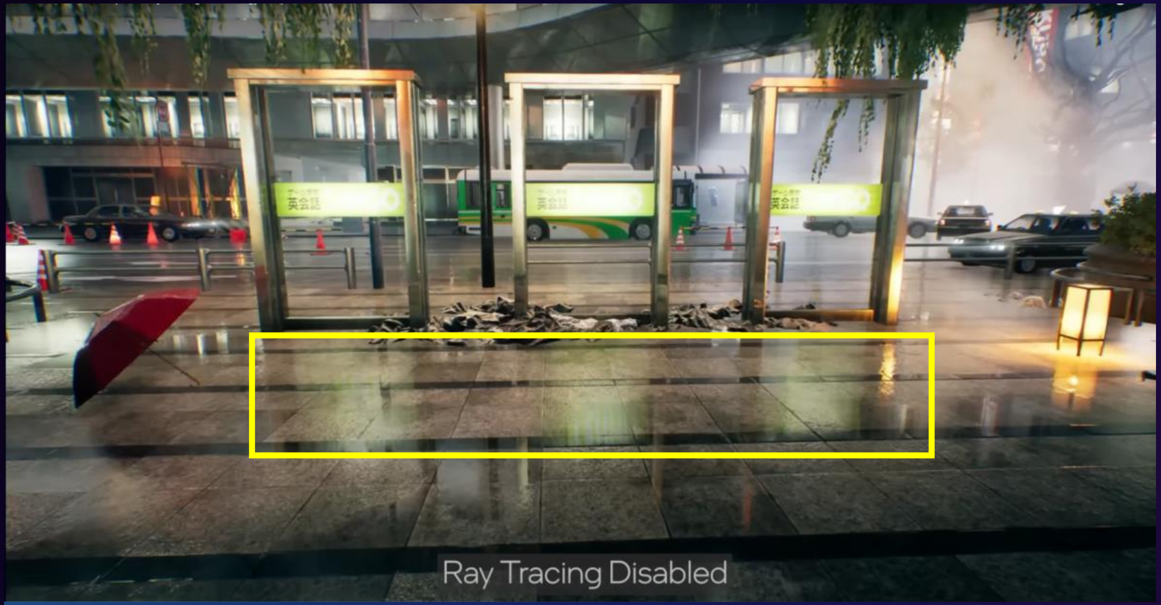
PART II

Vision for the future: Realizing scalable real-time ray tracing

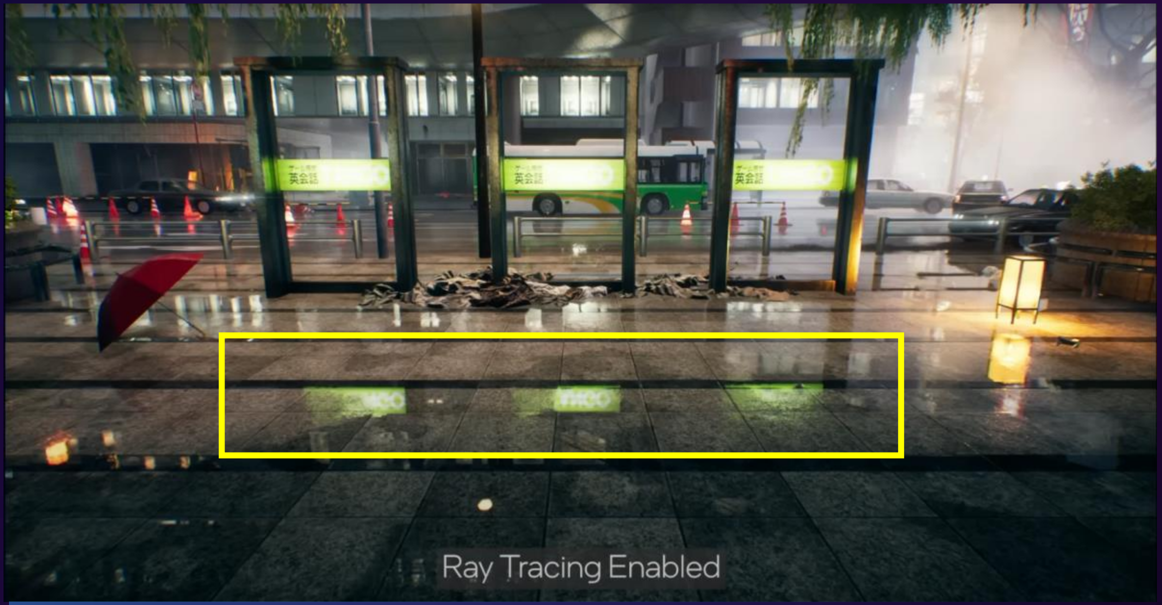
Intel's research to enable a future of scalable ray tracing



Lower quality visuals with screen space approximations vs Ray Tracing



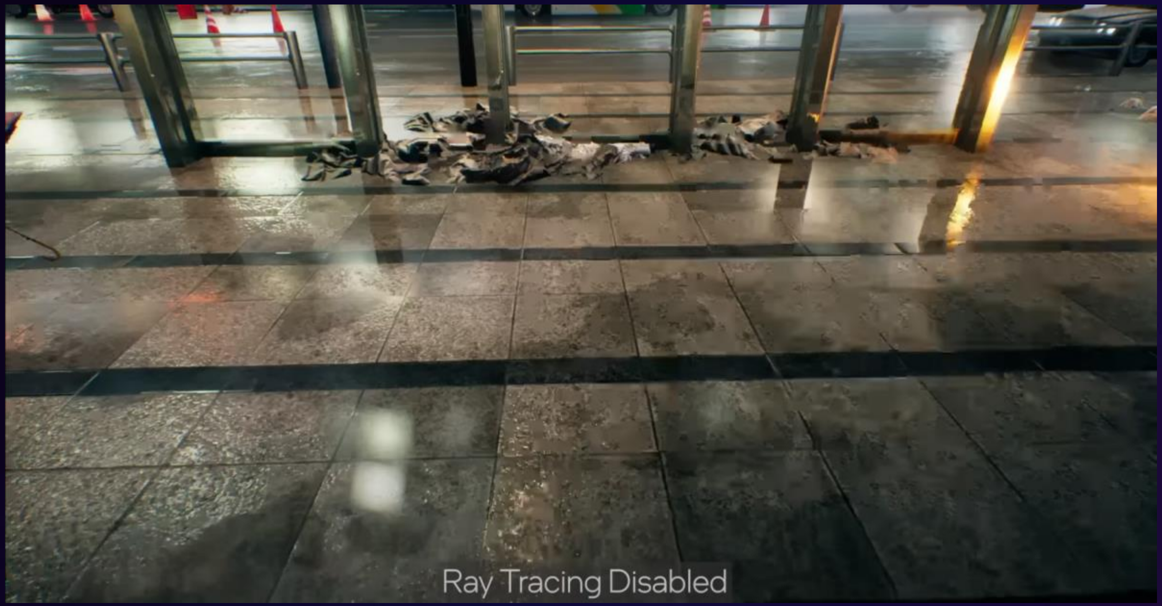
Screen Space Reflections

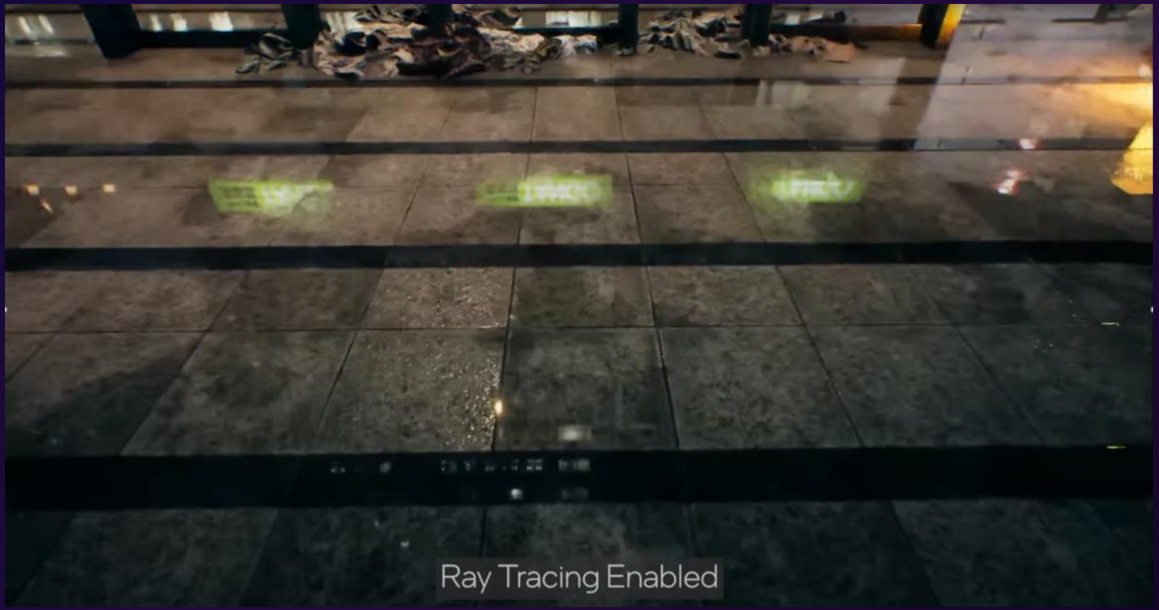


Ray-Traced Reflections

Images courtesy of Ghostwire: Tokyo [Intel 2022]

Screen Space Reflections are lost when reflected light source is not on screen

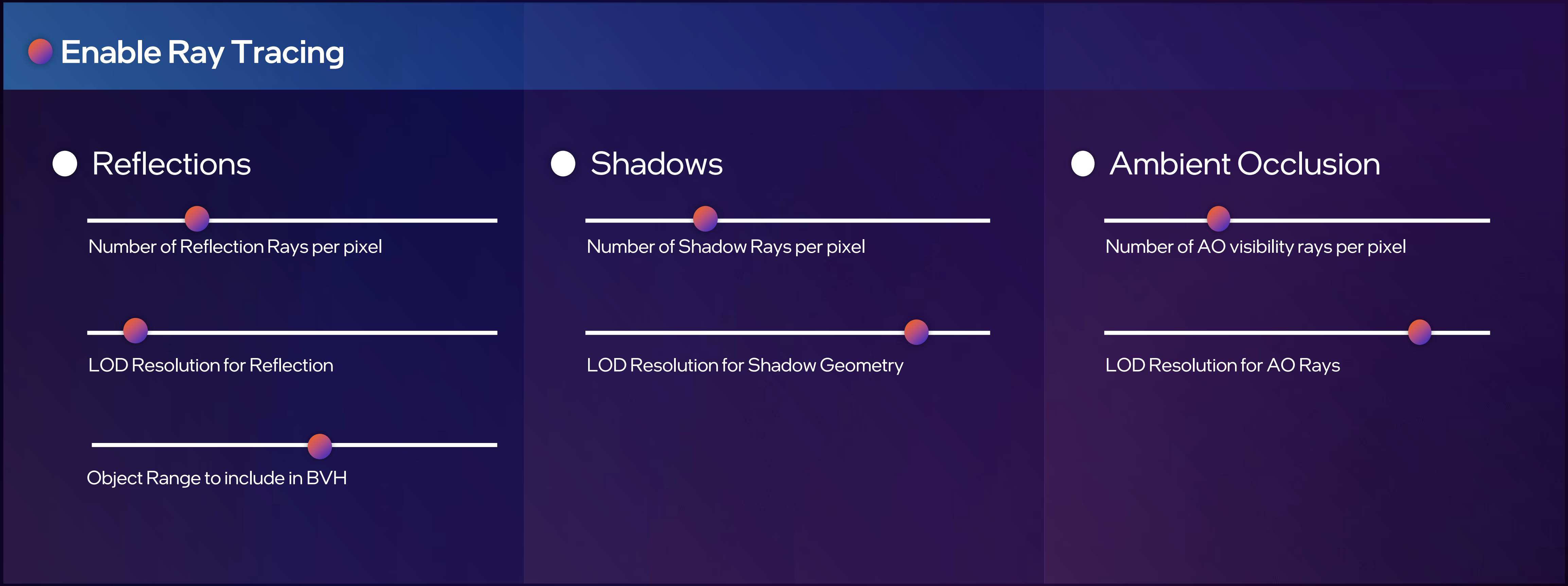




As you tilt you head down, the SSR reflections disappear, this is not how the real-world works!

Images courtesy of Ghostwire: Tokyo [Intel 2022]

Scalable Real-Time Ray Tracing Effects

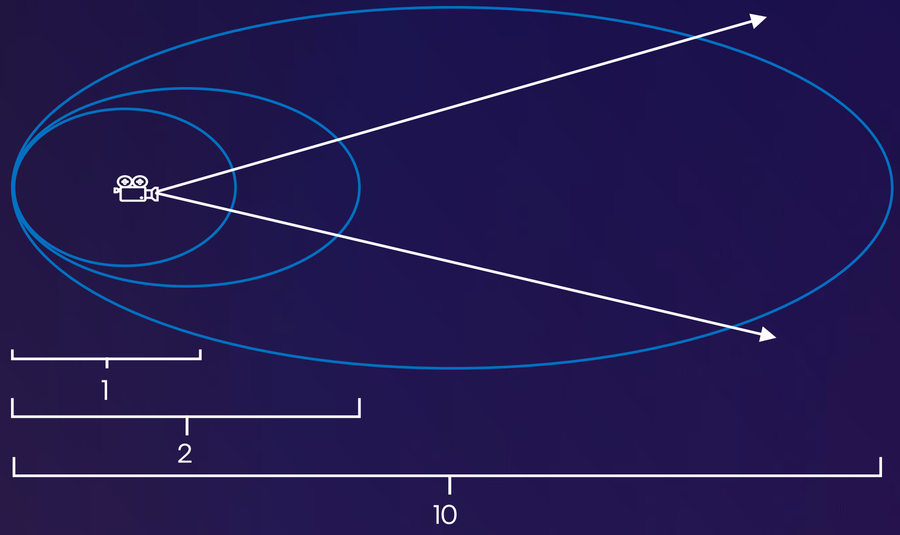


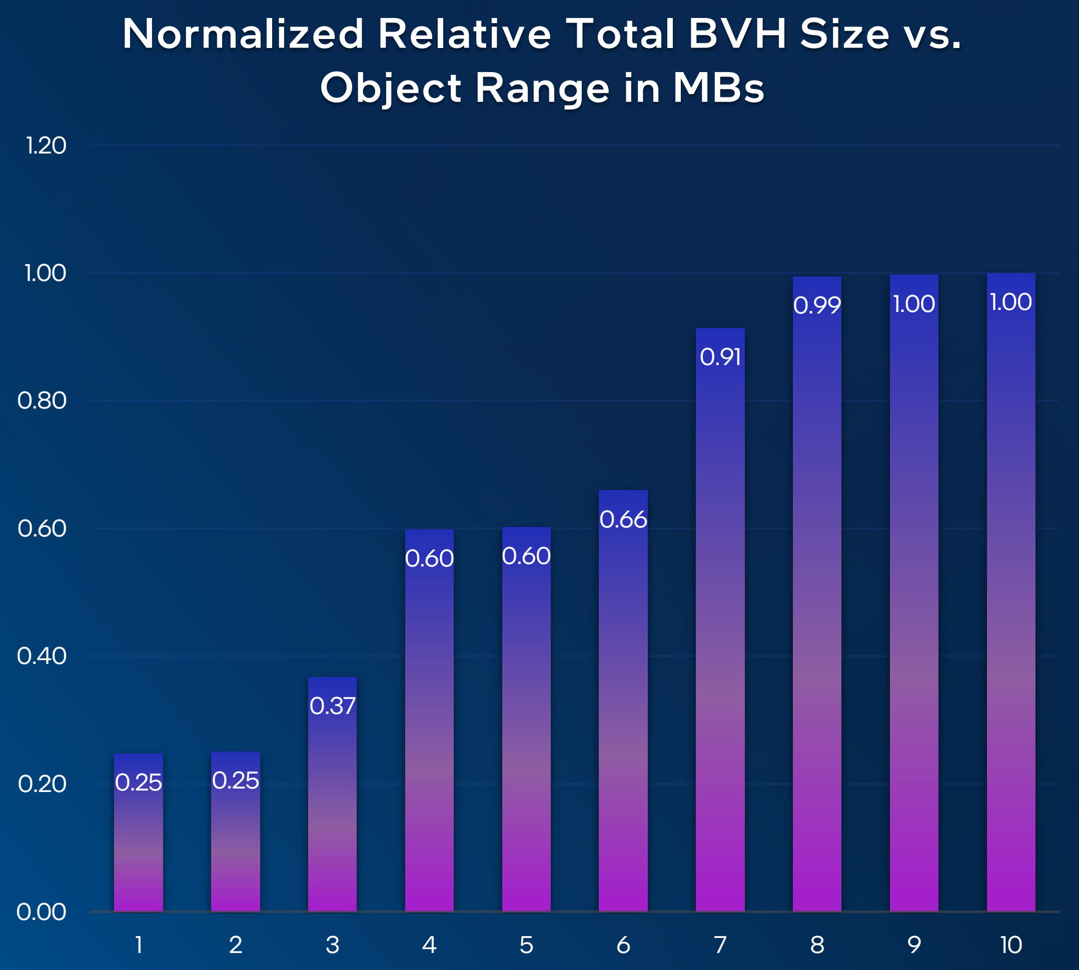
● Enable Ray Tracing

- Reflections
  - Number of Reflection Rays per pixel
  - LOD Resolution for Reflection
  - Object Range to include in BVH
- Shadows
  - Number of Shadow Rays per pixel
  - LOD Resolution for Shadow Geometry
- Ambient Occlusion
  - Number of AO visibility rays per pixel
  - LOD Resolution for AO Rays

Scaling number of models included in BVH by distance from player

- Allow to control the range of objects around the player that are used to build the BVH for reflections
- As Object Range in BVH increases, size of BVH grows
- Give user control over visual quality vs. performance



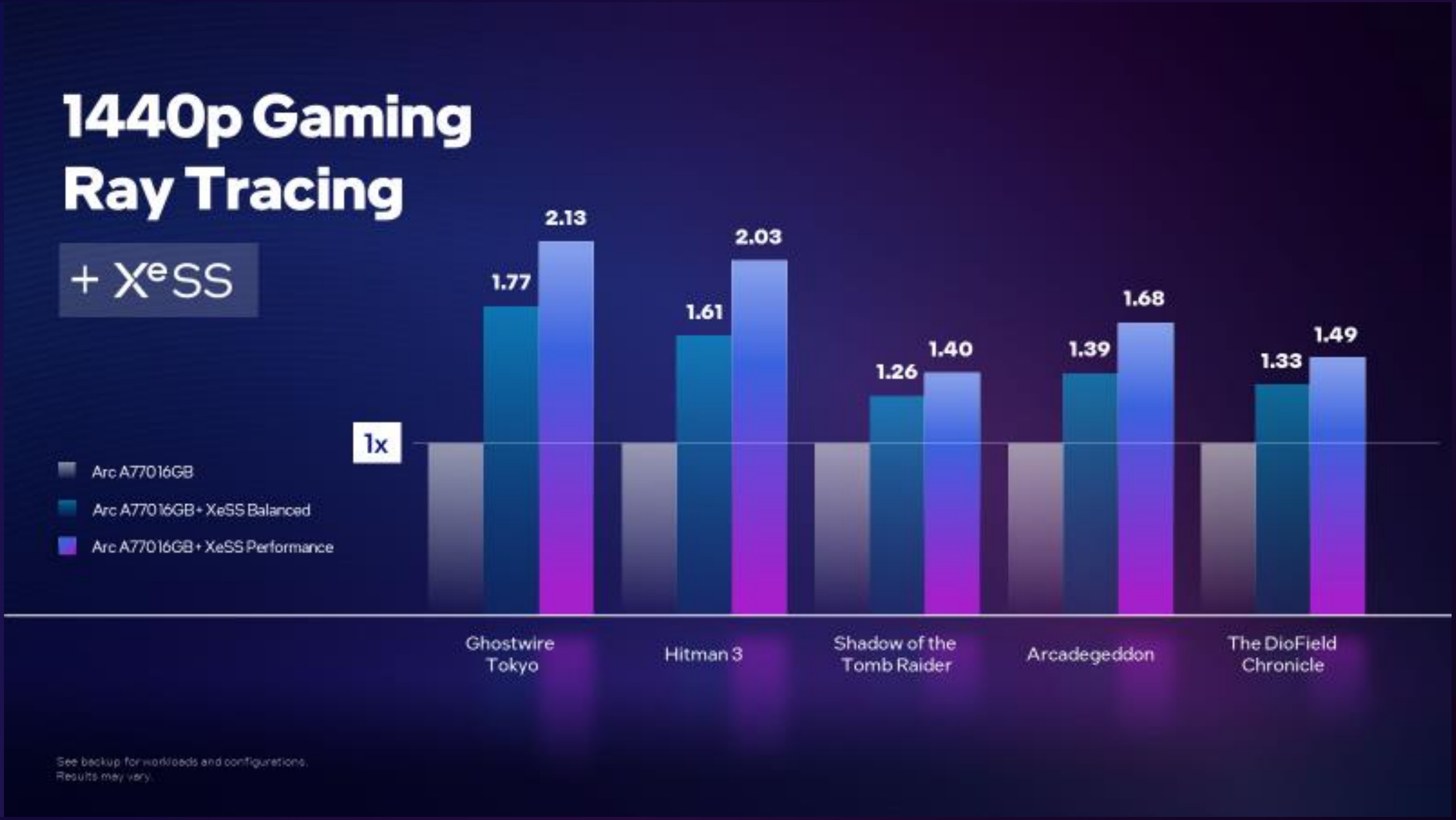


intel. ARC

XeSS + Ray Tracing

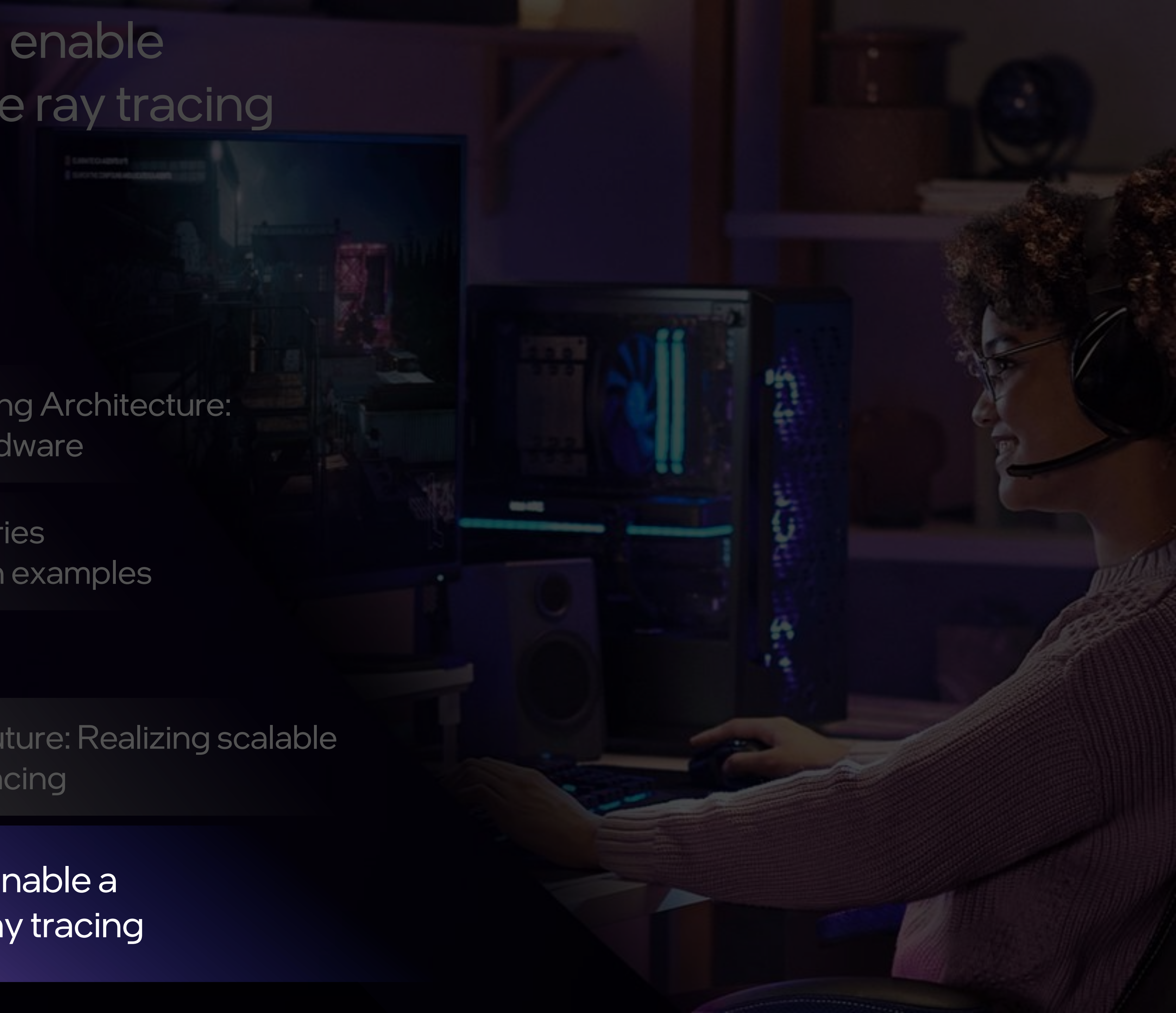
- No discussion of scalability is complete without upscaling solutions
- Game developers seeing *amazing* results incorporating XeSS with and without ray tracing.

- *More details in another talk that is part of this series*



See backup for workloads and configurations.
Results may vary.

Intel’s research to enable a future of scalable ray tracing



PART I

Intel® Arc A-series Ray Tracing Architecture: Mapping the DXR API to hardware

Optimizing for Intel Arc A-series Graphics: Best practices with examples

PART II

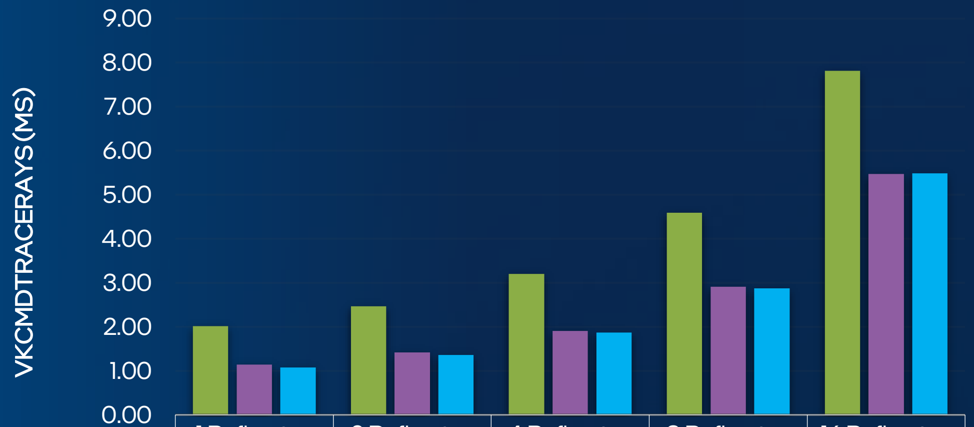
Vision for the future: Realizing scalable real-time ray tracing

Intel’s research to enable a future of scalable ray tracing

Ray Tracing Level of Detail

- Cost of BVH traversal is a key contributor to performance
  - Reducing the # of triangles in BVH can improve performance
- A novel stochastic level of detail (LOD) algorithm for ray tracing from [Lloyd et al 2020]
- What is stochastic LOD?
  - On a per ray basis, trace LOD[i] or LOD[i+1]
  - Cross-dissolve stochastically from one to the other based on distance, ray path length, etc.
  - Compared to discrete LOD, results in smoother transition
  - Downsides:
    - Only 8 levels of transition [Gruen 2021]
    - The InstanceMask can no longer be used for intended use cases. For example, the TLAS includes a full set of visible objects but then uses the InstanceMask for objects that can cast shadows.

Intel Arc A-Series Performance:
No LOD vs. Discrete vs. Stochastic with Instance Mask, millseconds in VkCmdTraceRays()
Lower is better



	1 Reflection Ray	2 Reflection Rays	4 Reflection Rays	8 Reflection Rays	16 Reflection Rays
🟩 No LOD	2.02	2.47	3.20	4.59	7.82
🟪 Discrete (Instance Mask)	1.14	1.42	1.91	2.91	5.47
🟦 Stochastic (Instance Mask)	1.08	1.36	1.87	2.87	5.49

🟩 No LOD 🟪 Discrete (Instance Mask) 🟦 Stochastic (Instance Mask)

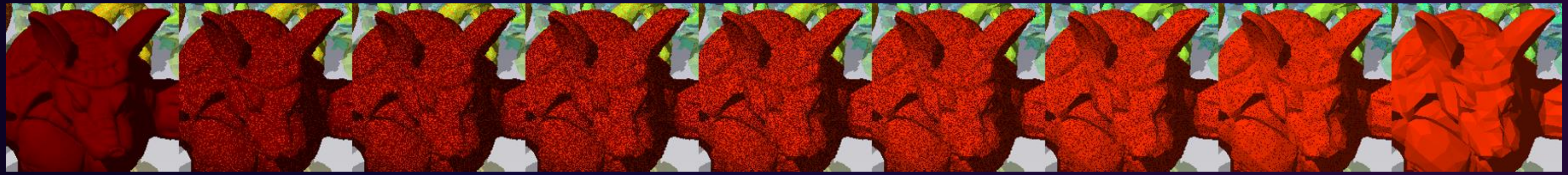
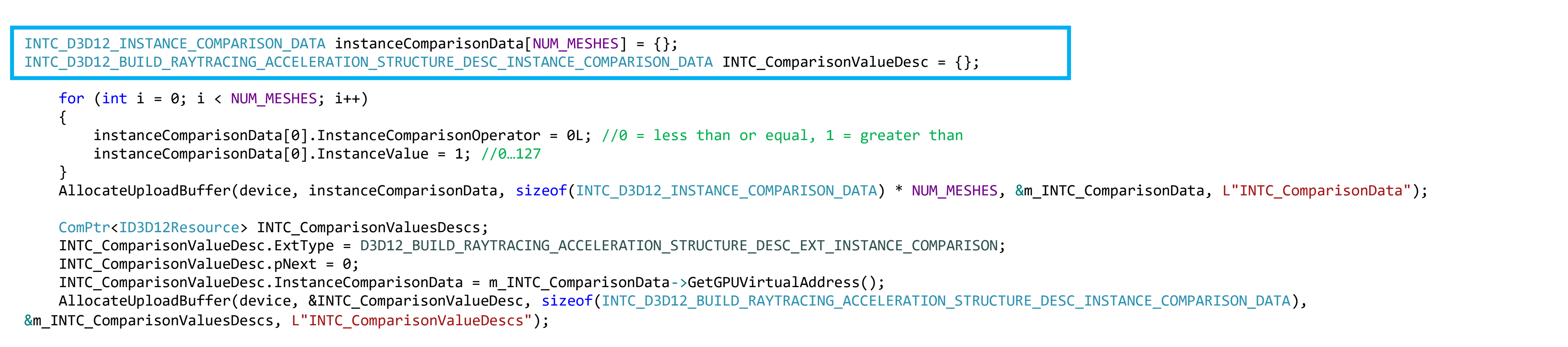


Image from Lloyd 2020

Can we improve on LOD via instance mask?

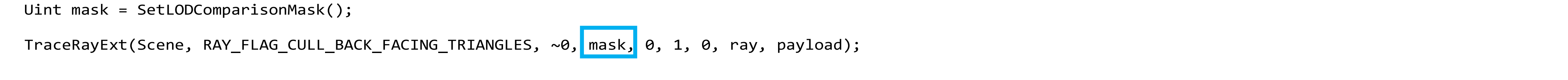
CPU Side:

New Instance Comparison Structure:



```
INTC_D3D12_INSTANCE_COMPARISON_DATA instanceComparisonData[NUM_MESHES] = {};
INTC_D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC_INSTANCE_COMPARISON_DATA INTC_ComparisonValueDesc = {};

    for (int i = 0; i < NUM_MESHES; i++)
    {
        instanceComparisonData[0].InstanceComparisonOperator = 0L; //0 = less than or equal, 1 = greater than
        instanceComparisonData[0].InstanceValue = 1; //0…127
    }
    AllocateUploadBuffer(device, instanceComparisonData, sizeof(INTC_D3D12_INSTANCE_COMPARISON_DATA) * NUM_MESHES, &m_INTC_ComparisonData, L"INTC_ComparisonData");

    ComPtr<ID3D12Resource> INTC_ComparisonValuesDescs;
    INTC_ComparisonValueDesc.ExtType = D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC_EXT_INSTANCE_COMPARISON;
    INTC_ComparisonValueDesc.pNext = 0;
    INTC_ComparisonValueDesc.InstanceComparisonData = m_INTC_ComparisonData->GetGPUVirtualAddress();
    AllocateUploadBuffer(device, &INTC_ComparisonValueDesc, sizeof(INTC_D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC_INSTANCE_COMPARISON_DATA),
&m_INTC_ComparisonValuesDescs, L"INTC_ComparisonValueDescs");
```

HLSL:



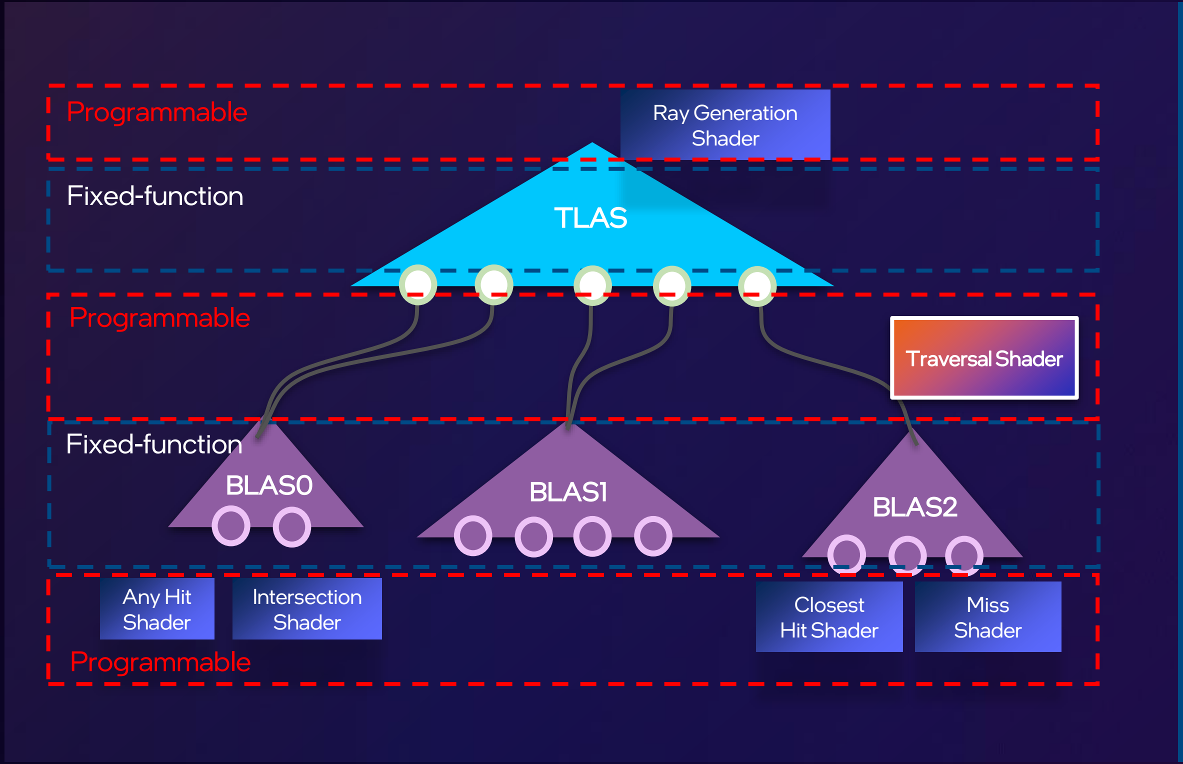
```
Uint mask = SetLODComparisonMask();

TraceRayExt(Scene, RAY_FLAG_CULL_BACK_FACING_TRIANGLES, ~0, mask, 0, 1, 0, ray, payload);
```

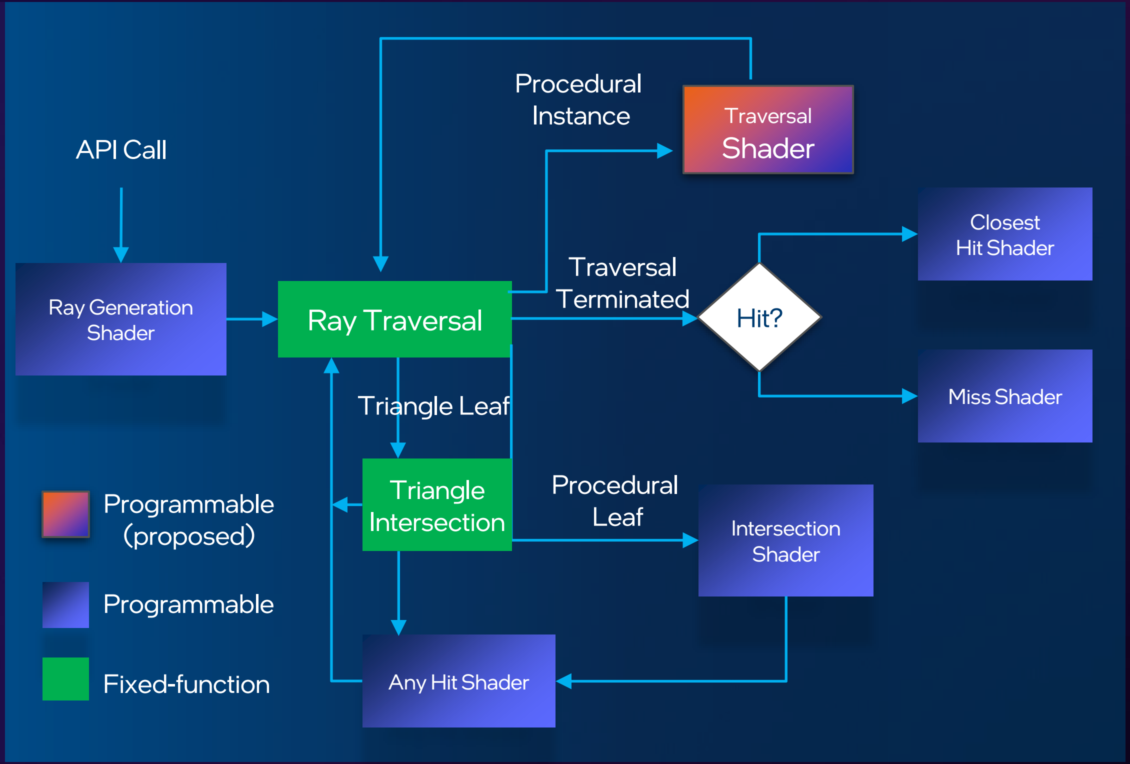
- On GPU, Instance Comparison result OR'd with InstanceMask result to determine if ray traces the BVH. If either fails ray will not be tested against object.

128 comparison values [0..127] vs. InstanceMask limitation of 8 can improve smoothness of LOD transition

Introducing Traversal Shaders

- What are traversal shaders? [Lee 2019 et al.], [Lee 2020 et al.]
  - New programmable stage: procedural selection of BVH during traversal
  - Can be mixed with existing HW instancing (procedural instance node type in TLAS, ~SW instancing)
  - Brings functionality from synchronous Ray Query API with benefits of TSU-based asynchronous sorting

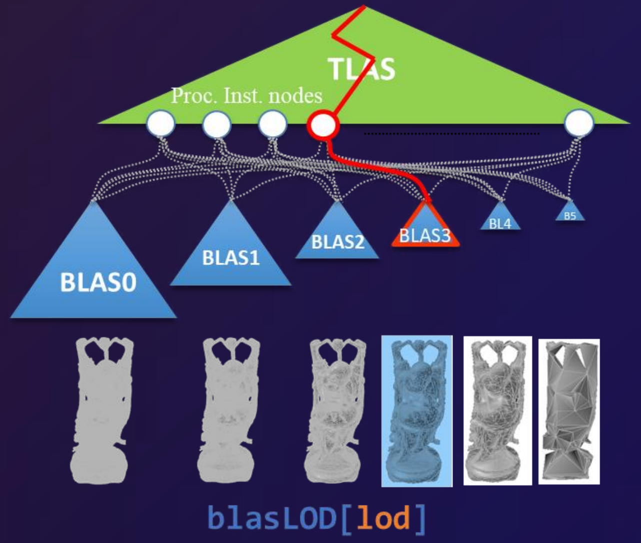




Traversal Shaders for Programmable LoD

Motivation

- Instance mask may already be reserved for purposes other than LoD (artist control / perf reasons)
- LoD selection may depend on hit distance which is unknown at time of TraceRay() call.
- Without traversal shaders, multiple TraceRays are needed, e.g. traversing multiple TLAS-es.



```
{
    […]
    rayDesc.Origin = ObjectRayOrigin();
    rayDesc.Direction = ObjectRayDirection();

    uint lod = ComputeLOD();
    uint shaderTableOffset = 2u + lod;

    ForwardRay(
        blasLOD[lod],
        shaderTableOffset,
        rayDesc,
        rayFlags,
        rayPayload);
}
```

TLAS traversal programmatically selecting LOD via **ForwardRay()**

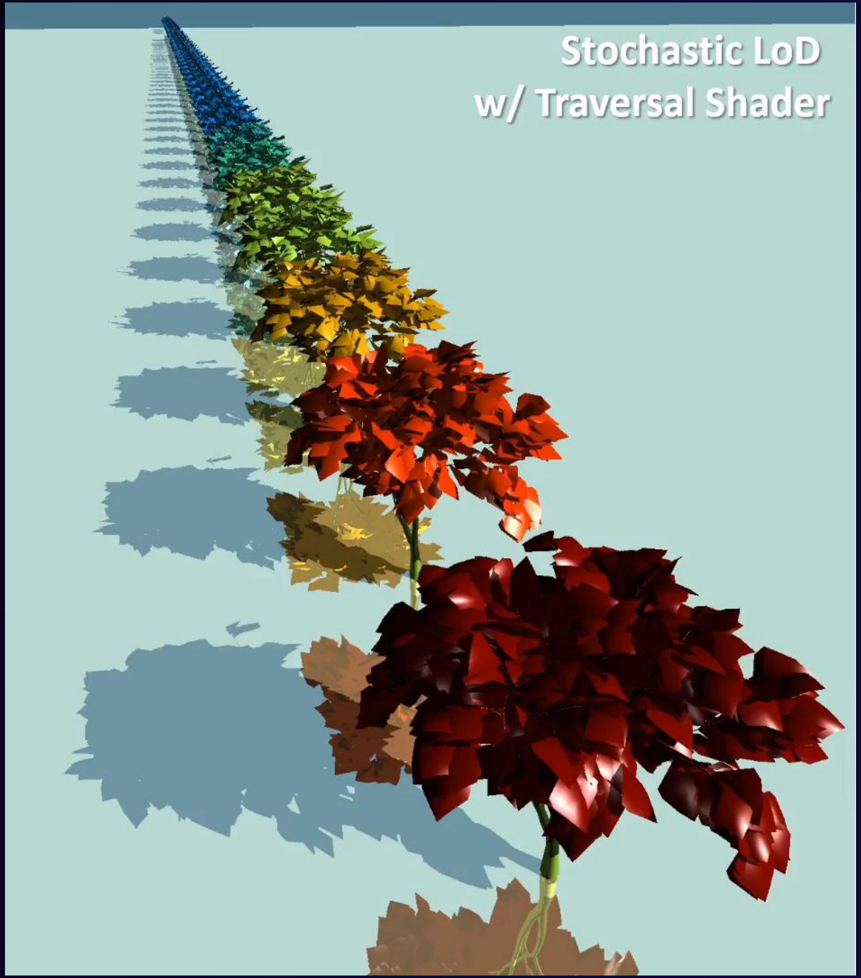
Use cases

- **Stochastic LoD transitions** with indirect rays
- **Adaptive LoD bias for GI:** choose coarser LoD to improve performance based on ray differentials and hit distance
- **Proxy fallback for missing LoD:** if the BLAS is not present (streaming, limited build budget…), forward ray to best available LoD

ForwardRay()is like Traceray()except we specify a low-level BLAS to forward the ray instead of an entire TLAS

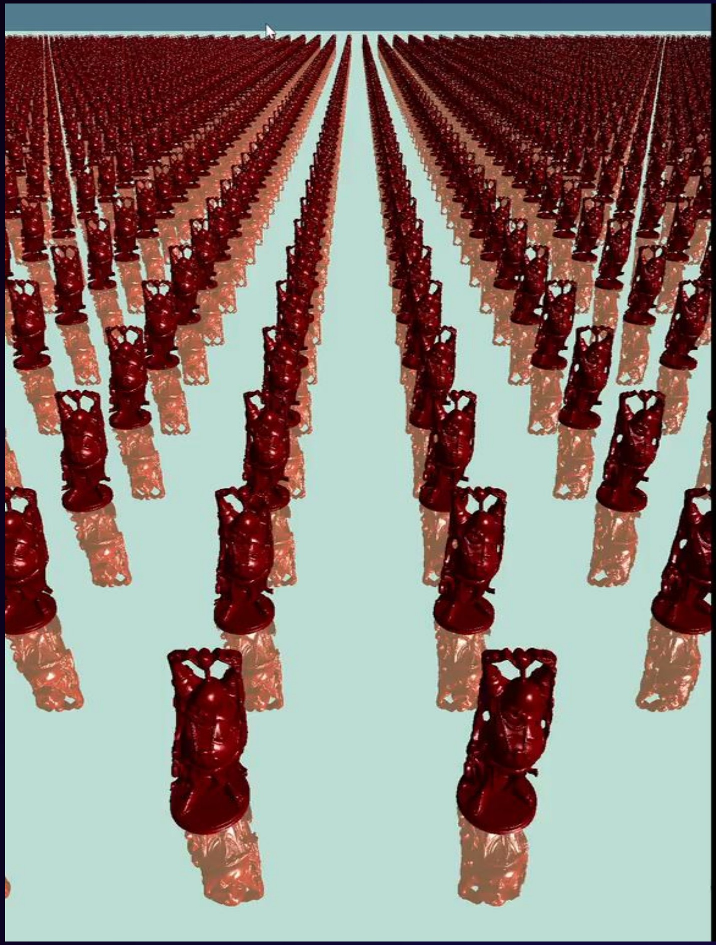
Programmable LoD Use Cases

Stochastic LOD (incl indirect rays)

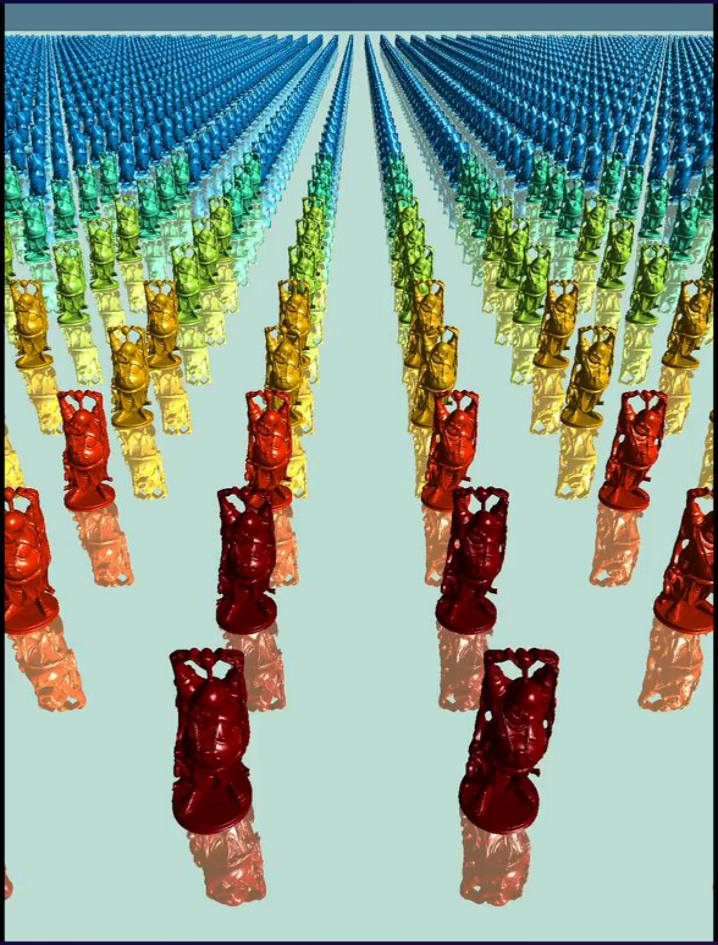


- 6 LODs
- Each LOD 4x reduction in triangles
- 4 Reflection rays
- 1 shadow ray
- Stochastically selects per ray which BVH LOD to traverse
- Running on Intel® Arc A770

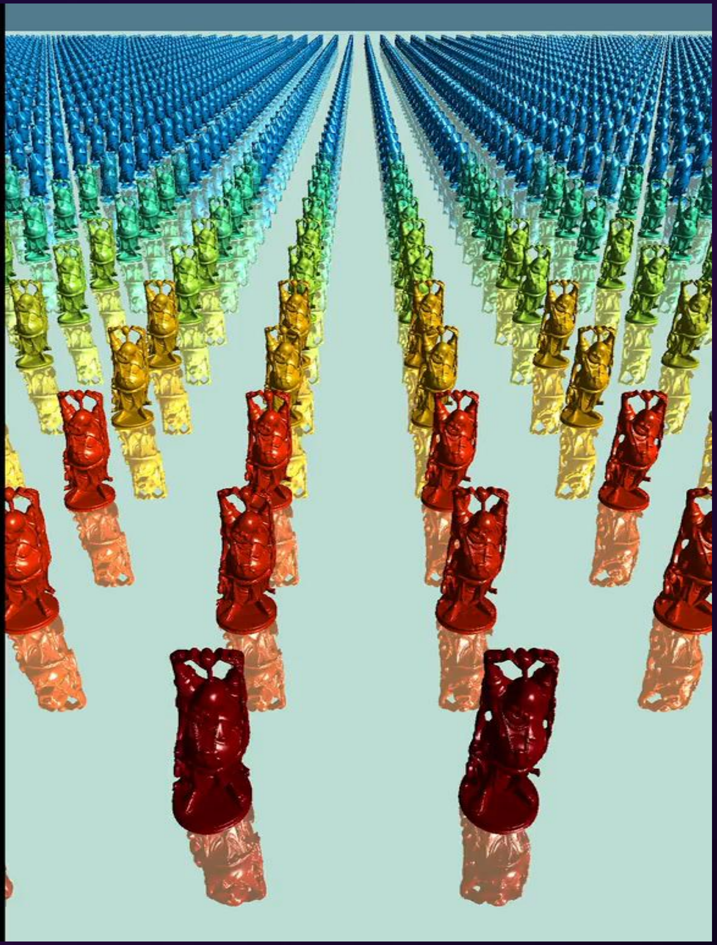
No LOD



Discrete LOD w/ Traversal Shaders



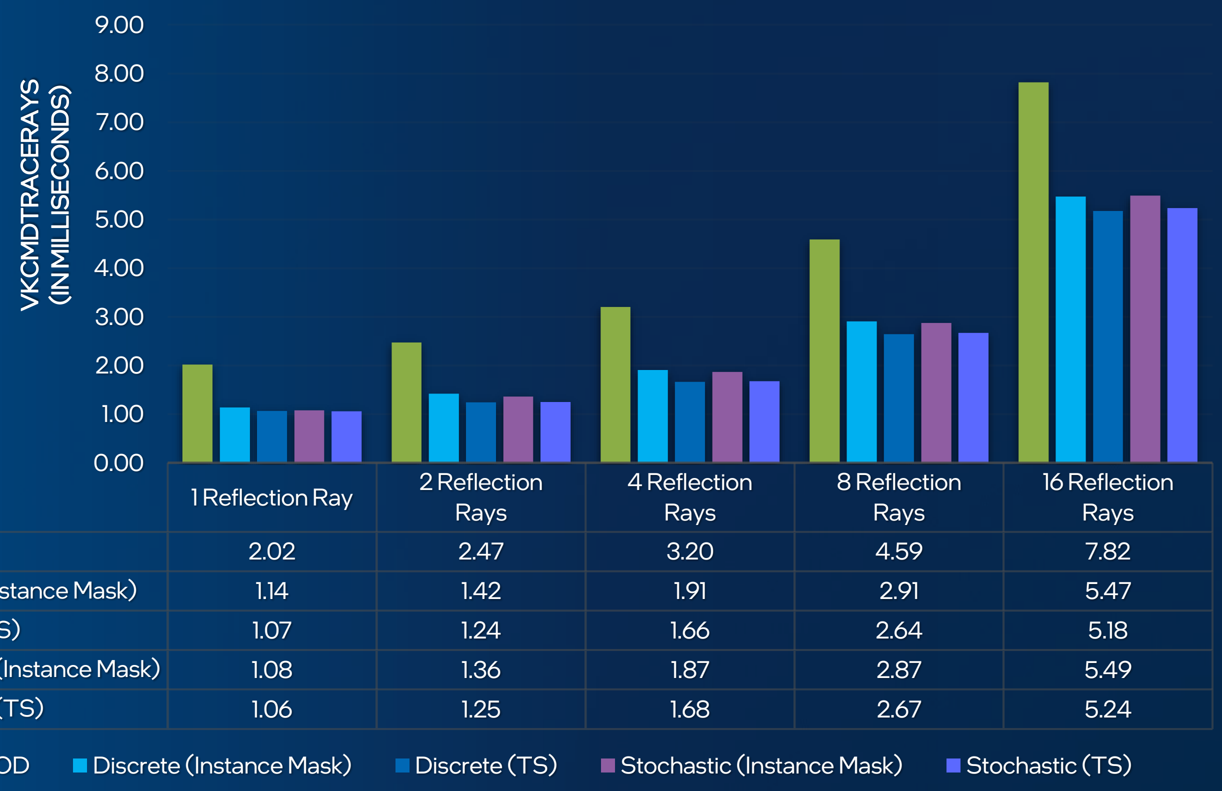
Stochastic LOD w/ Traversal Shaders



Traversal Shader Performance Results

- Fully programmable
- No additional cost over the InstanceMask based solution
- Retain Instance mask functionality

Intel Arc A-Series Performance:
Ray Tracing LOD with and without traversal shaders
milliseconds in vkCmdTraceRays()
lower is better



	1 Reflection Ray	2 Reflection Rays	4 Reflection Rays	8 Reflection Rays	16 Reflection Rays
🟩 No LOD	2.02	2.47	3.20	4.59	7.82
🟦 Discrete (Instance Mask)	1.14	1.42	1.91	2.91	5.47
🟦 Discrete (TS)	1.07	1.24	1.66	2.64	5.18
🟪 Stochastic (Instance Mask)	1.08	1.36	1.87	2.87	5.49
🟪 Stochastic (TS)	1.06	1.25	1.68	2.67	5.24

Rendering inflection point ....

- The future: Ray Tracing Hardware is ubiquitous, from mainstream to high end GPUs
- Long term art and rendering pipeline simplification w/ fewer effects requiring raster and RT implementations
- Empower users with control over scalable ray tracing parameters
- Interested in traversal shaders? Email adam.t.lake@intel.com or gabor.liktor@intel.com for more details

Acknowledgements

- VCG Graphics SW Dev: Przemyslaw Szymanski
- VCG GPU Performance: John Gierach
- VCG GPU Rendering Team: Alexander Kharlamov, Dave Astle, Daniele Pieroni
- AXG GRO Collaborators: Gabor Liktor, Anton Sochenov, Holger Gruen
- AXG Architecture team: Josh Barczak
- AXG Staff and Marketing: Tom Peterson, Ryan Shrout, Damien Triolet
- SATG SSE WPE: Jiawei Shao

References

- [Barczak 2022] Josh Barczak and Holger Gruen, Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games. https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html. Last accessed 1/4/2023.
- [Gruen 2022] Holger Gruen and Josh Barczak. Video: A Quick Guide to Intel's Ray Tracing Architecture, GDC 2022. Last accessed 1/18/2023.
- [Gruen 2021] Holger Gruen. Ray Traced Level of Detail Cross-Fades Made Easy. Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan and Optix. https://www.realtimerendering.com/raytracinggems/rtg2/. Edited by Adam Marrs, Peter Shirley, and Ingo Wald. Last accessed 2/26/2023.
- [Hector 2020] Tobias Hector, Joshua Barczak, Eric Werness. Ray Tracing in Vulkan. https://www.khronos.org/blog/ray-tracing-in-vulkan. Last accessed 1/4/2023.
- [Intel 2022] Inter Arc A-Series Graphics Ray Tracing Technology Deep Dive. https://www.youtube.com/watch?v=J5elOv-CrB8&ab_channel=IntelGraphics. Last accessed 1/29/2023.
- [Intel 2022] Inter Arc Graphics | Intel XeSS Technology Deep Dive. https://www.youtube.com/watch?v=frlXry38tJo&ab_channel=IntelGraphics. Last accessed 1/29/2023.
- [Lyoyd 2020] Branch Lloyd, Oliver Klehm, Martin Stitch. Implementing Stochastic Levels of Detail with Microsoft DirectX Raytracing. https://developer.nvidia.com/blog/implementing-stochastic-lod-with-microsoft-dxr/. Last accessed 2/16/2022.
- [Lee et al 2019] Won-Jong Lee, Gabor Liktor, and Karthik Vaidyanathan. Flexible Ray Traversal with an Extended Programming Model. ACM SIGGRAPH Asia 2019 Technical Brief, pp. 17-20.
- [Lee et al 2020] Won-Jong Lee and Gabor Liktor ACM SIGGRAPH Asia 2020. Lazy Build of Acceleration Structures with Traversal Shaders. ACM SIGGRAPH Asia 2020. Technical Communication, Article No. 11, pp 1-4.
- [Tabellion 2010] Eric Tabellion. 2010. Ray Tracing vs. Point-based GI for Animated Films. In ACM SIGGRAPH 2010 Course: Global Illumination Across Industries.
- [Usher 2022] The Shader Binding Table Demystified. Ray Tracing Gems II, pp 193-211. https://link.springer.com/chapter/10.1007/978-1-4842-7185-8_15. Last Accessed 1/28/2023.

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at **www.Intel.com/PerformanceIndex** (graphics and accelerators).

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. No product or component can be absolutely secure.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

Intel technologies may require enabled hardware, software or service activation.

All product plans and roadmaps are subject to change without notice.

Statements that refer to future plans or expectations are forward-looking statements. These statements are based on current expectations and involve many risks and uncertainties that could cause actual results to differ materially from those expressed or implied in such statements. For more information on the factors that could cause actual results to differ materially, see our most recent earnings release and SEC filings at www.intc.com.

Workloads and Configurations

Claim	System configuration	Measurement	Measurement period
Intel® Arc™ A770 with XeSS delivers increased ray tracing performance at 1440p as measured by FPS when compared to gameplay without XeSS	Graphics: Intel® Arc™ A770 Graphics, Graphics Driver: Engineering Driver 3262, Processor: Intel® Core™ i9-12900K, Asus ROG MAXIMUS Z690 Hero, BIOS: 1601, Memory: 32GB (2x16GB) DDR5 @ 4800MHz, Storage: Corsair MP600 Pro XT 4TB NVMe, OS: Windows 11 Version 22000.795	All games tested at 1440p and 1080p using highest possible settings, except turned off motion blur and screen effects for Shadow of the Tomb Raider. Chose highest preset, then manually increased individual settings to maximum. Ray tracing options set to maximum on all games. XeSS Performance and Balanced Mode tested on all titles. Game workloads that support this claim are Arcadegeddon, The DioField Chronicle, Ghostwire Tokyo, Hitman 3, and Shadow of the Tomb Raider	August 5-8, 2022

Thank you

intel®

intel ARC™