Are we done with hardware ray tracing?

Or …

How can we make real-time raytracing more pervasive?

Holger Gruen – *Principal engineer*

XPU Technology and Research group

Intel Corporation

HPG 2020

Ray tracing hardware is great!

- It solves real-time problems rasterization can't solve efficiently
- Recent DXR games have shown a variety of fantastic ray traced effects
- There is also great progress with regards to denoising
- Yet, ray traced rendering is more than just tracing rays
  - BVH management (issues wrt build/refit/streaming)
  - Hit point shading (SIMD utilization issues)

  ⇒ Current AAA games need to limit ray tracing
- This talk is about open problems that need to be overcome
  - Beyond what just making GPUs faster will give us
  - e.g. like we have overcome traversal performance problems

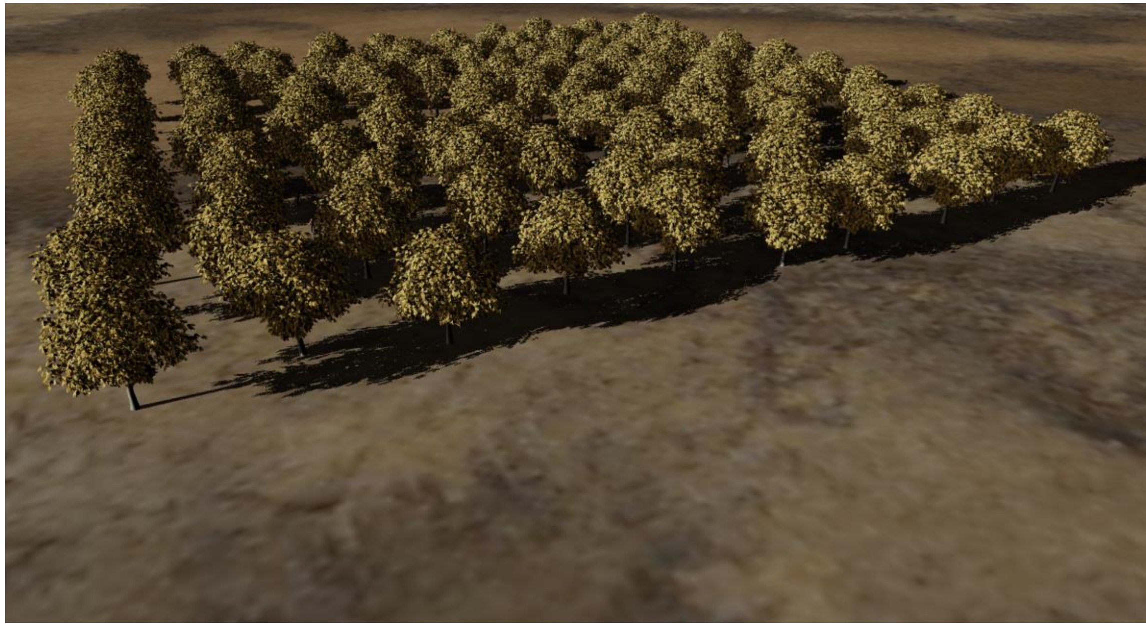
Rasterization can deal with …

- 1000s of uniquely on-the fly skinned characters
- An animated forest with alpha-tested foliage
- Highly programmable on-the-fly per instance deformations (UE Kite demo)
- (DX12) mesh shaders that generate dynamic geometry on-the-fly
- Dynamic geometry that gets tessellated on-the-fly
- Massive amounts of virtual geometry that gets streamed and decompressed on-the-fly (see e.g. UE5 demo video)

But what about raytracing?

A simple example (1080p, high end GPU*):

- A test scene with ~19M triangles
  - 100 uniquely skinned characters
  - 100 uniquely animated trees
- Gbuffer rasterization: ~2.7 ms
- Raytracing cost: ~8.6 ms
  - Compute Shader animation: ~2.5 ms
  - BLAS updates: ~3.2 ms
  - Primary rays: ~2.9 ms (1 ray/pixel)

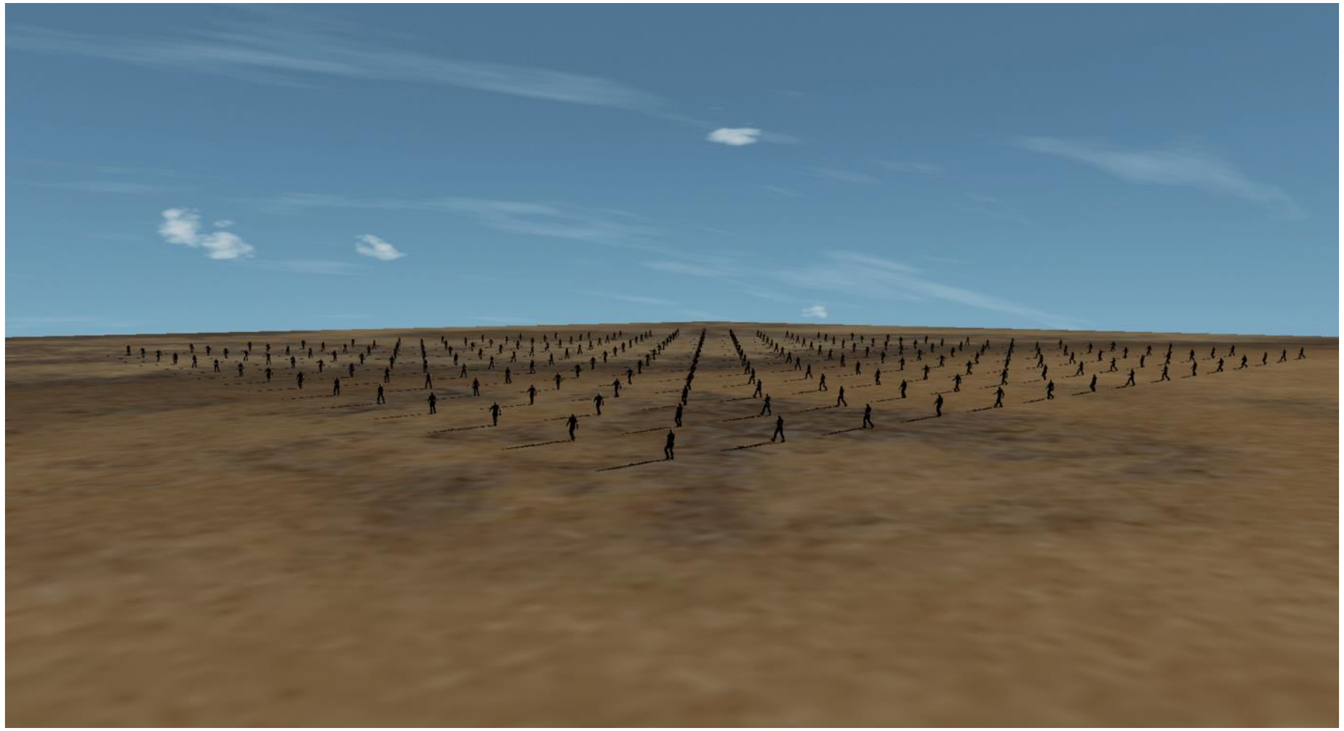


*NVIDIA® GeForce® RTX® 2080Ti

BVH related issues

BVH building/refitting costs

- BVH updates don't yet scale like streamed rasterized geometry
  - 225 skinned characters (20k triangles each)
  - Rasterization: ~1.8 ms
  - Animation + BVH refit: ~6 ms



BVH building/refitting costs

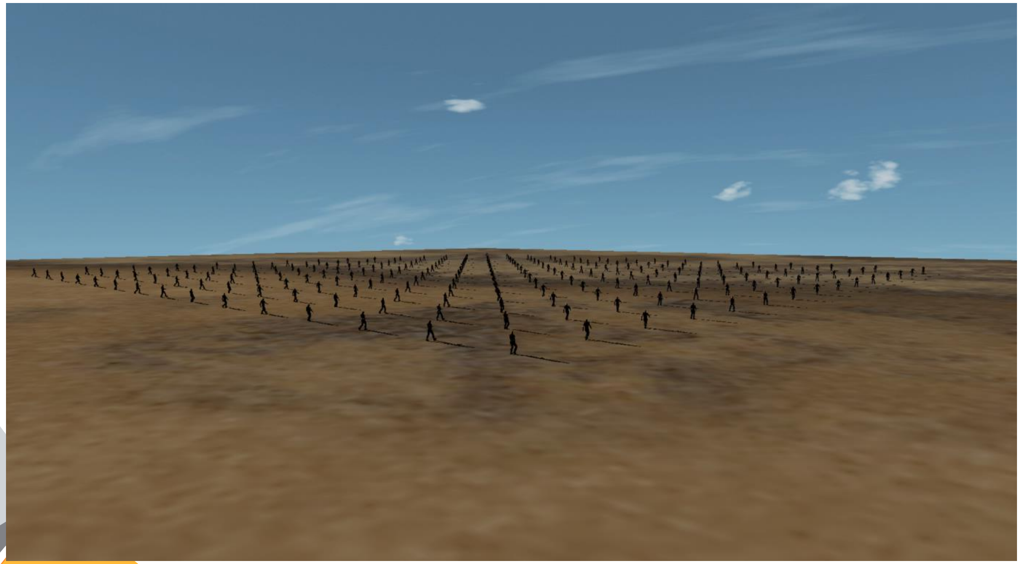
- BVH updates don't yet scale like streamed rasterized geometry
  - 100 animated (non-rigid) trees (160k triangles each)
  - Rasterization: ~2.4 ms
  - animation + BVH refit: ~5.8 ms

- Procedural and dynamically tessellated geometry typically perform worse

High BVH Memory Footprints

- Modern games push a lot of dynamic/procedural geometry!
- A dynamic BVH consumes ~60-80 bytes per triangle

  ⇒225 uniquely skinned characters (20k tris each) consume ~280MB
  ⇒100 uniquely animated trees (160k tris each) consume ~980MB



High BVH Memory Footprints

- Static BVHs still use about 30-50 bytes per prim
  - ⇒Raytraced effects can access most of the scene geometry due to secondary rays
  - ⇒The whole scene may need to be in the BVH (pathtracing)
- Near future: UE5 scenes rumored to have billions of triangles
  - ⇒Needs very aggressive view dependent culling
  - ⇒Is current BVH storage/build/update/streaming technology up to this task?

⇒Current AAA games limit BVH complexity and as a result ray tracing

Potential solutions to the above BVH issues

<u>Reduce memory footprints:</u>

- Hardware support for lossy geometry compression (BVH+Traversal)?

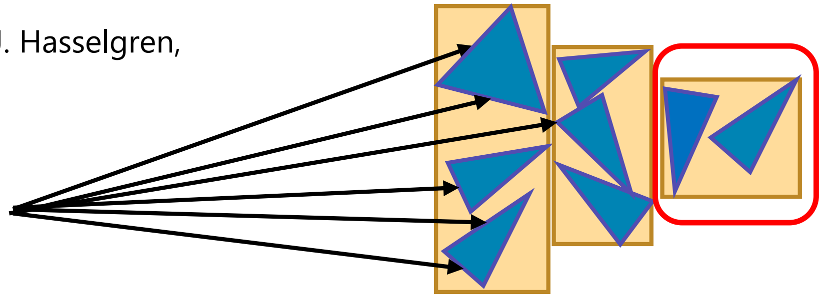
<u>Reduce build/refit costs:</u>

- Hardware accelerated builds?
- Support for lazy builds?
  - Similar to procedural texture problem: 'AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors', P. Clarberg, R. Toth, J. Hasselgren, J. Nilsson, T. Akenine-Möller

<u>Reduce memory footprint **&** build/refit costs:</u>

- Support lazy&caches hardware builds for transient dynamic or procedural pieces of geometry?

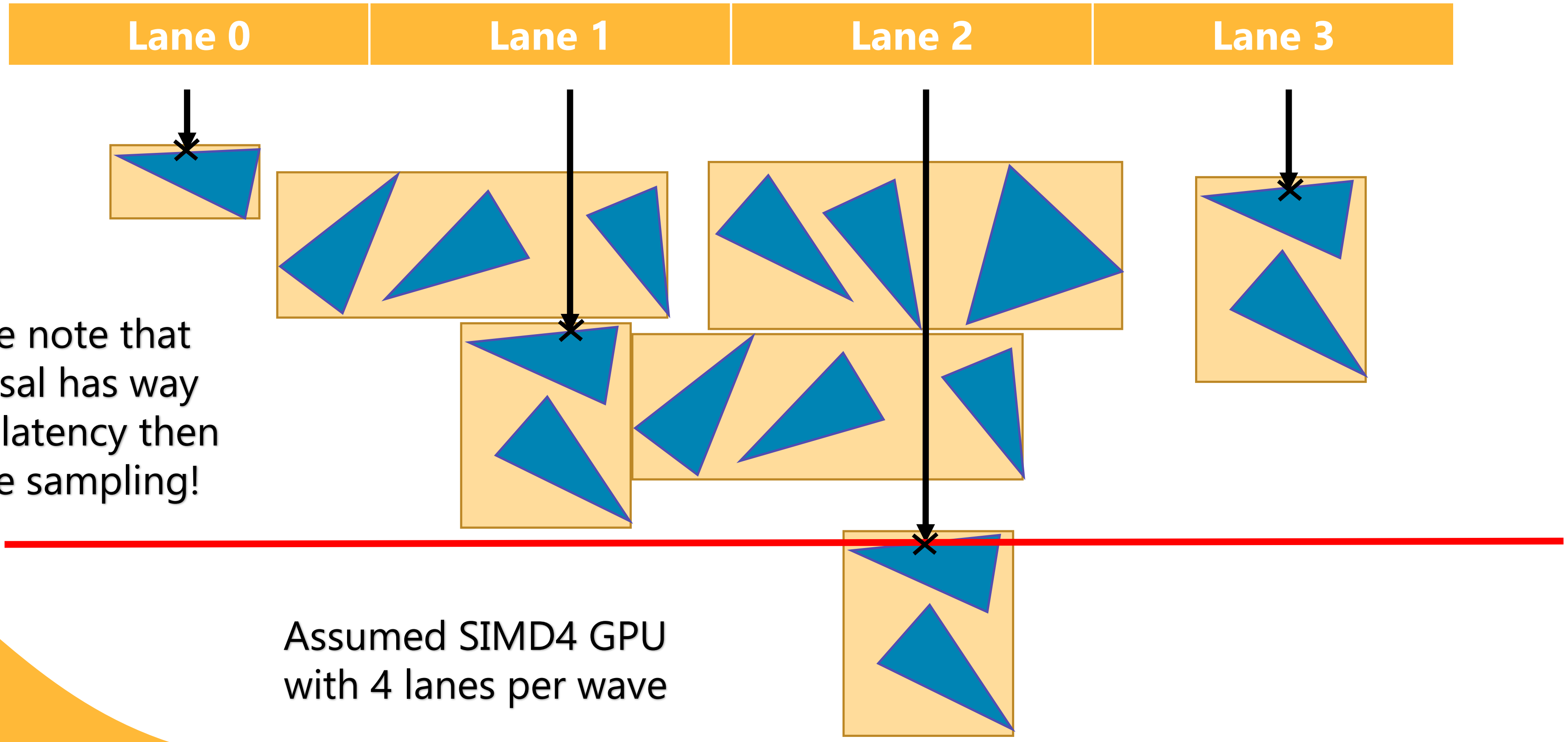
<u>Lazy builds:</u>
Only update nodes when they are visited by rays



Coherency & SIMD utilization issues

Most common coherency & SIMD utilization issues

- Divergent ray traversal duration/steps for rays
  ⇒All SIMD lanes blocked until the ray with the highest # of traversal steps returns

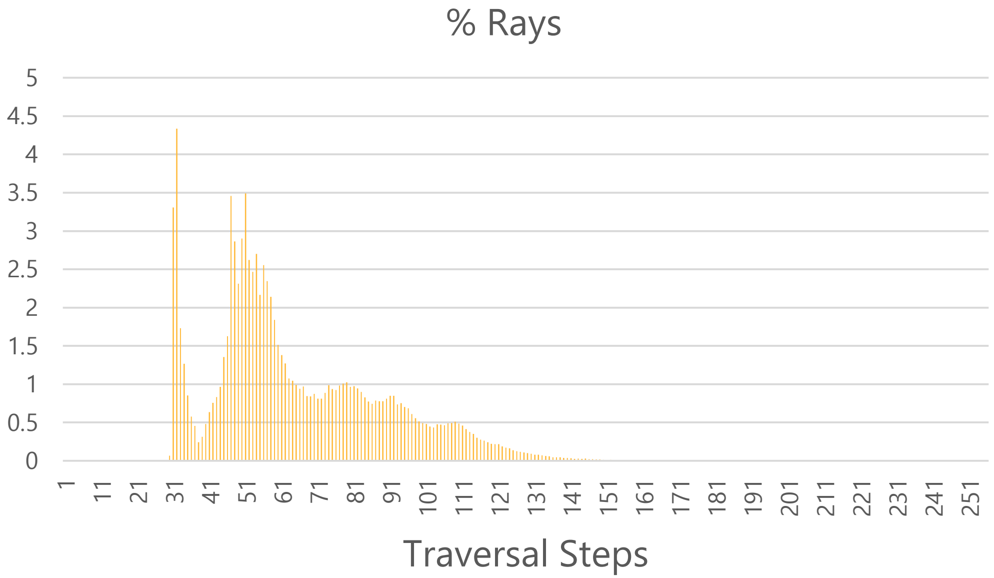


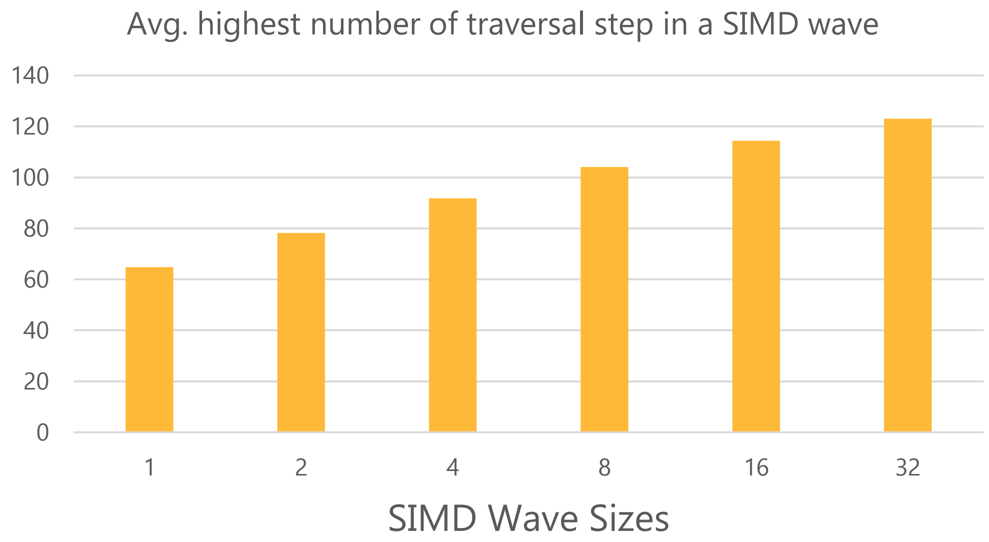
Please note that traversal has way higher latency then Texture sampling!

Assumed SIMD4 GPU with 4 lanes per wave

Most common coherency & SIMD utilization

- Divergent ray traversal duration/steps for rays
  - ⇒All SIMD lanes blocked until the ray with the highest # of traversal steps returns

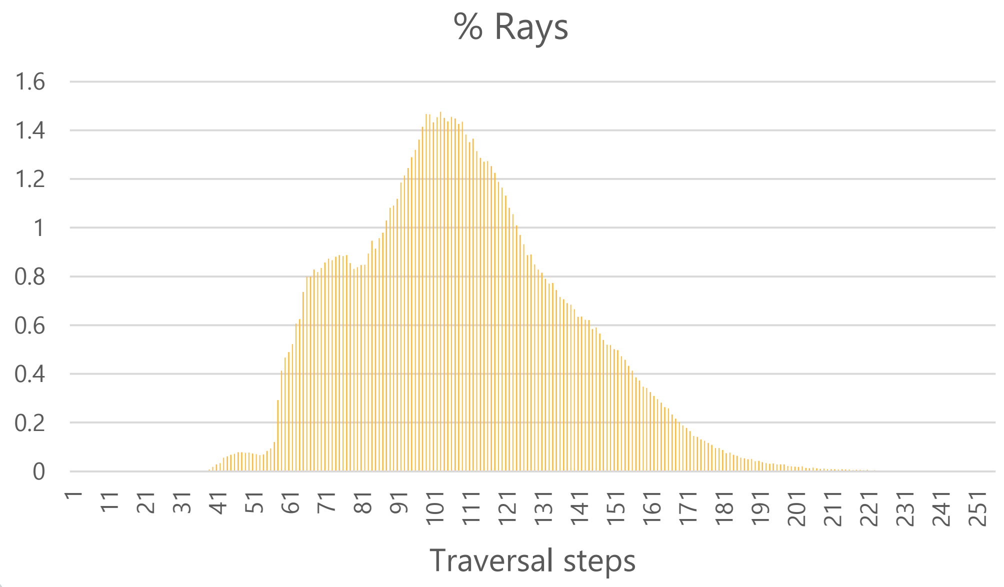


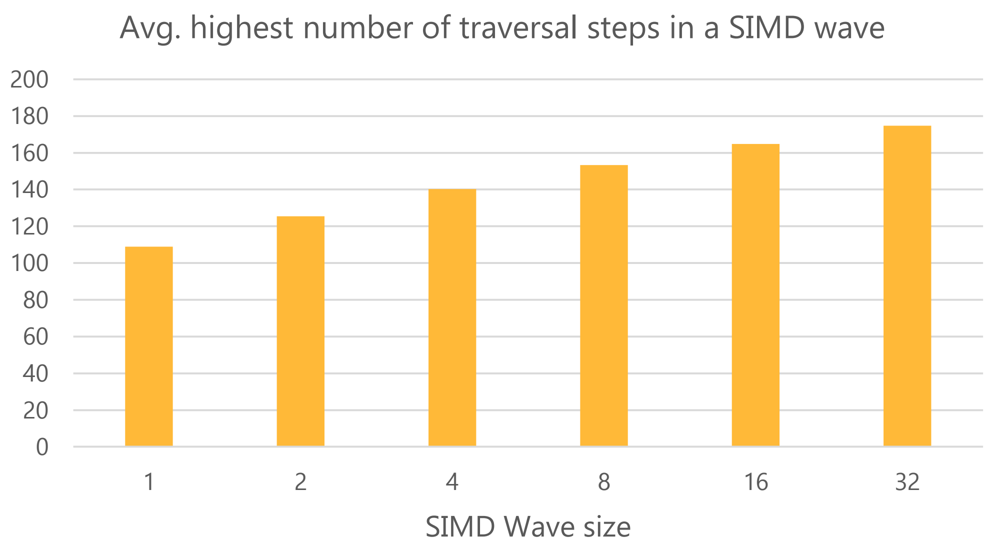


Camera rays from a DXR games

Most common coherency & SIMD utilization

- Divergent ray traversal duration/steps for rays
  - ⇒All SIMD lanes blocked until the ray with the highest # of traversal steps return

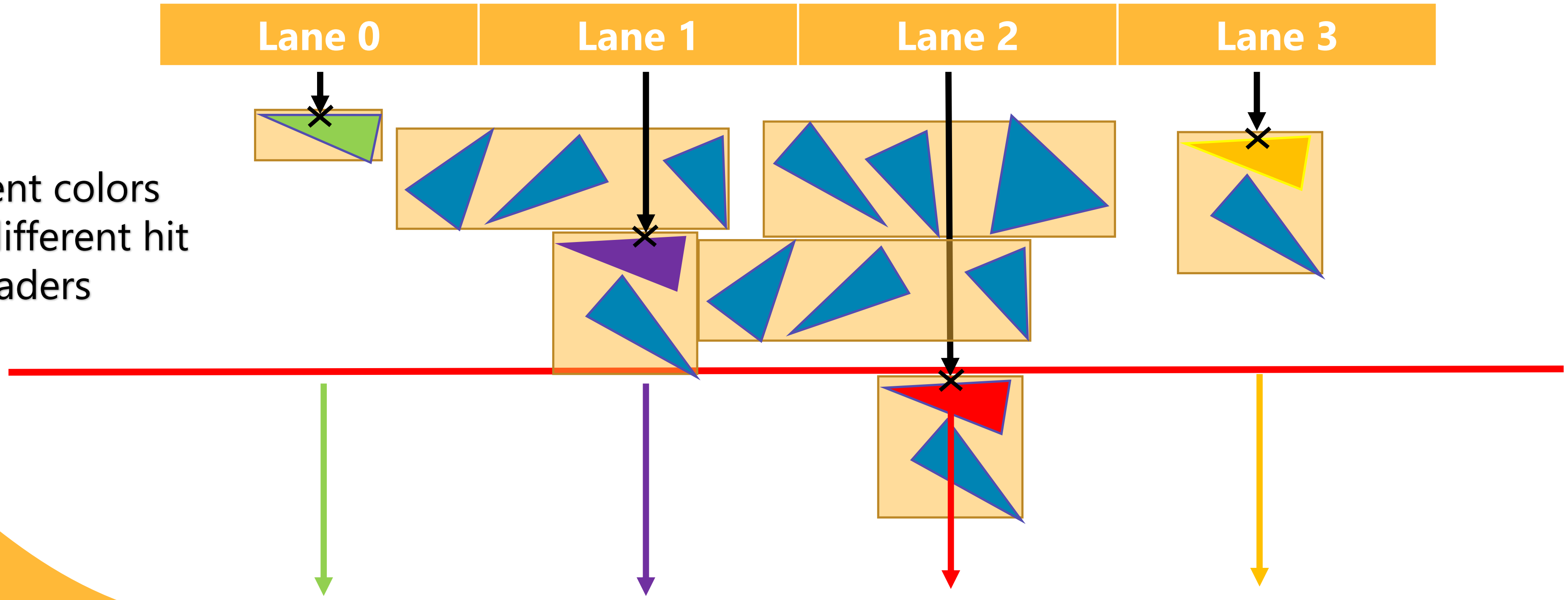




AO rays from a DXR games

Most common coherency & SIMD utilization issues

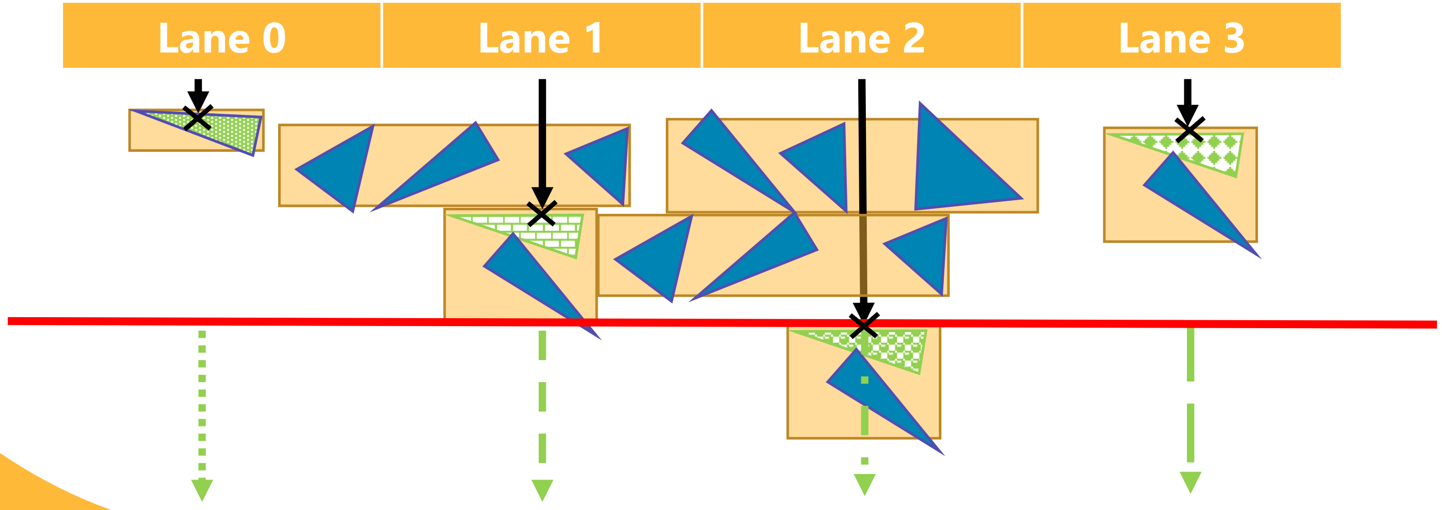
- Divergent ray traversal duration/steps for rays
  ⇒All SIMD lanes blocked until the longest ray traversal path is done
- Shader path divergence
  ⇒SIMD lanes may need to execute different hit/material shaders



Different colors depict different hit shaders

Most common coherency & SIMD utilization issues

- Divergent ray traversal duration/steps for rays
  ⇒All SIMD lanes blocked until the longest ray traversal path is done
- Shader path divergence
  ⇒SIMD lanes may need to execute different hit shaders
- Divergent textures
  ⇒Same shaders, but SIMD lanes may need to fetch from diverging resources



How to increase coherency & SIMD utilization?

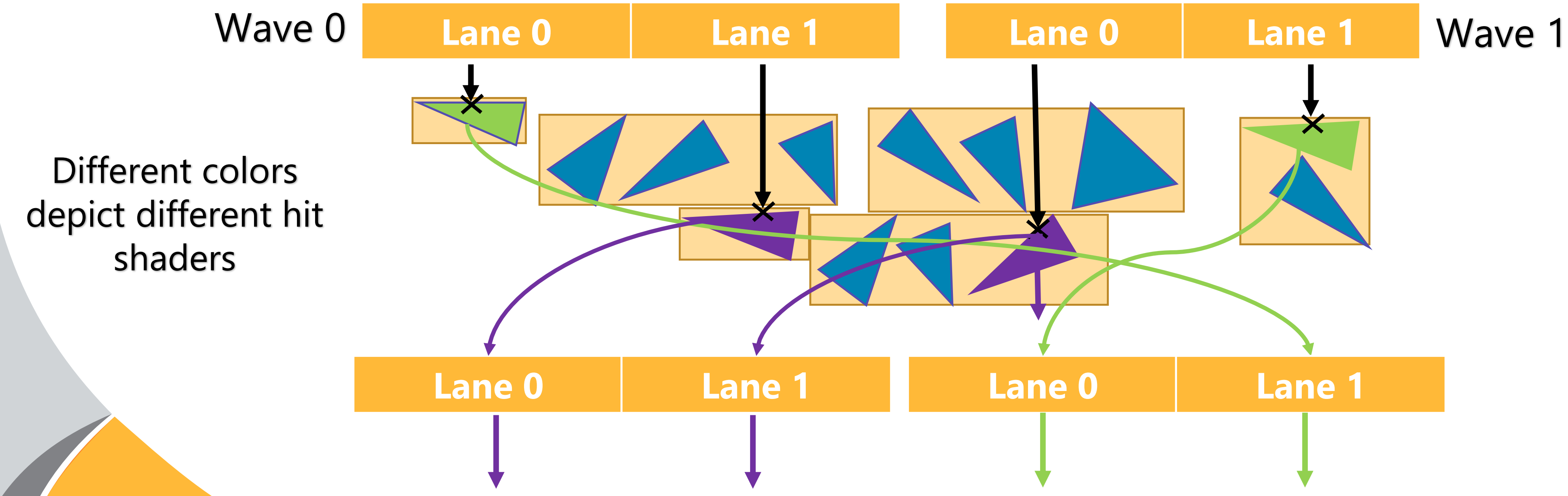
<u>Current solutions:</u>

- Sort rays for more traversal coherency/ less SIMD latency
  - e.g. by origin + direction, Morton code, ...
- Sort hit points for coherent shading
  - e.g. by material ID, shading model, etc.
- Expensive for multiple bounces
  - Hit point streaming can consume cosiderable bandwidth
  - High local shared memory footprints may limit your occupancy

How to increase coherency & SIMD utilization?

Potential future solutions:

- **Could we do asynchronous raytracing?**
- **Could hardware bundle coherent shading requests?**



LOD management issues

Current raytracing hardware allows only limited LOD management

- Crossfading is possible using instance and ray masks (see recent NVIDIA blog)
- Anyhit() shaders allow more programmable fades but are slower
- Geomorphing through refits seems possible but is expensive

<u>**Potential future solutions:**</u>

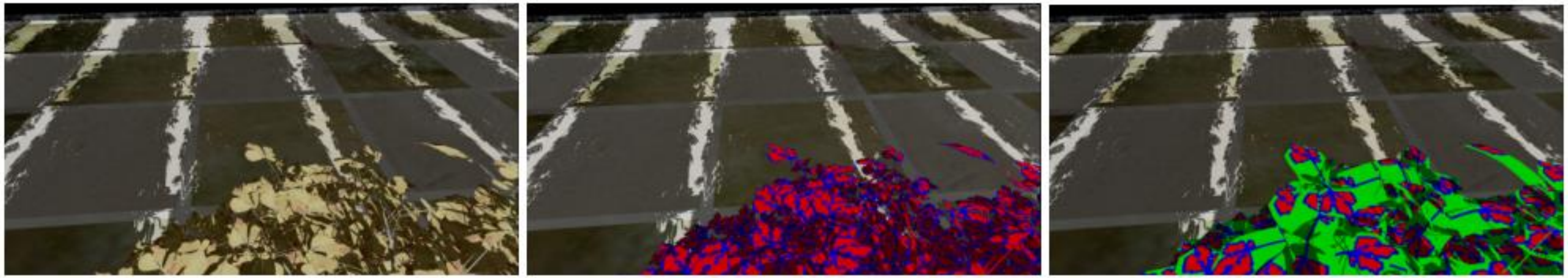
- **Fully implement a fast traversal shaders stage in hardware?**
    - See “Flexible Ray Traversal with an Extended Programming Model” by W. Lee, G. Liktor, K. Vaidyanathan
    - Traversal shaders can do flexible LOD selection and more!

Alpha testing is comparably slow

- Hardware traversal gets interrupted to run a shader that computes if a ray vs triangle intersection is valid

- See our talk on Wednesday:

  „Sub-triangle opacity masks for faster ray tracing of transparent objects”



Recognitions

Thanks to

Karthik Vaidyanathan,

Carsten Benthin,

Joshua Barczak

and Gabor Liktor from Intel

who contributed to the above slides.

Q&A