HOLOGRAFIKA

# Modern applications of GPU assisted ray tracing

Attila Barsi
Holografika

Contributors:
Oleksii Doronin

GPU Day,
Budapest, 21. 06. 2018.

# Outline

- Ray tracing basics
- Acceleration structures and traversal
- Overview of existing CPU and GPU ray tracing libraries
- Designing GPU ray tracers
- GPU ray tracing examples & best practices
- Holografika ray tracer

# Ray tracing

- Renders images by evaluating the interaction of lightbeams of an emitter with surfaces of various materials.
- Ray definition in parametric form
  $p_o + t\ d = p$ where $p_o$ is the ray origin, d is the ray direction vector and $t \geq 0$ is the ray parameter.
- We need the following functionality:
  - Ray generation.
    For surface point on emitter, (s,t) evaluate ray.
    For pixel (x,y) of the camera image evaluate ray.
  - Any hit.
    Is the ray hitting any surfaces?
  - Miss.
    Is the ray missing all surfaces?
  - Closest hit.
    Which is the closest surface to the ray?
  - Hitpoint evaluation.
    If the ray is hitting the surface, what is the hitpoint?
    If the ray is hitting the acceleration structure, what is the incoming and the outgoing hitpoint?
  - Shading

# Ray tracing vs. rasterization

- Rasterization disadvantages:
  - Linear interpolation
  - Cannot change projection per fragment
- Rasterization advantages:
  - Fast
- Conclusion:
  1. Use rasterizer wherever linear interpolation works. Use ray tracer everywhere else.
  2. Use ray tracing sparingly.

# Ray-plane intersection

- Given two points $p_a$ and $p_b$ on the plane with normal n

  $(p_a - pb) \cdot n = 0$

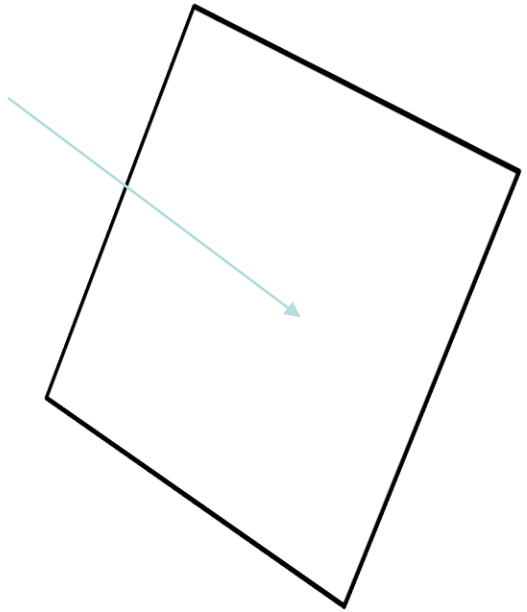
  and the definition of the ray we have that

  $(p_o + t\ d - pb) \cdot n = 0$

  Solving for t we get:

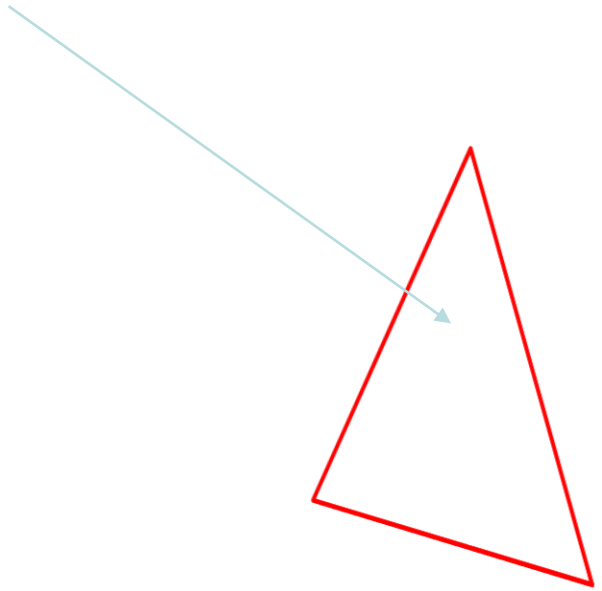
  $t * d \cdot n + (p_0 - pb) \cdot n = 0$

  $t = -\frac{(p0 - pb) \cdot n}{d \cdot n} = \frac{(pb - p0) \cdot n}{d \cdot n}$



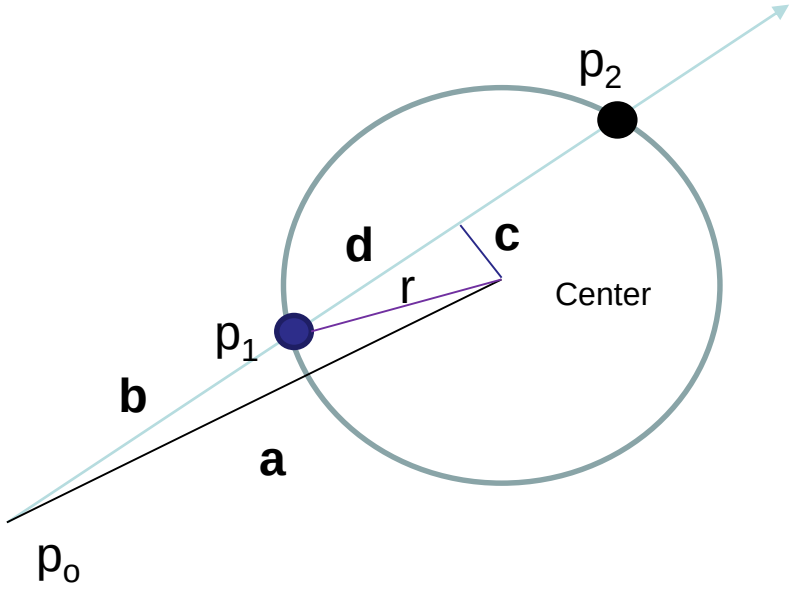
# Ray-triangle intersection

- Given a triangle with $v_0$, $v_1$, $v_2$.
- $edge_0 = v_1 - v_0$
- $edge_1 = v_2 - v_0$
- $c_0 = r_o - v_0$
- $p = d \times edge_1$
- $det = edge_0 \cdot p$
- Check det. Should be det>0 if culling is on. Should be abs(det)>0 if culling is off.
- $u = c_0 \cdot p \frac{1}{det}$
- $q = c_0 \times edge_0$
- $v = d \cdot q \frac{1}{det}$
- Check *u*,*v*. *u* should be between [0..1] *v* should be between [0..1-u] exclusive.
- $t_{ray} = edge_1 * q * \frac{1}{det}$



# Ray-sphere intersection

- Given a sphere with $c_0$ center and $r$ radius and ray with $p_o$ and $dir$
- $\boldsymbol{a} = c_0 - po$
  $\boldsymbol{b} = \boldsymbol{a} \cdot dir$
  $\mathbf{c} = \sqrt{\boldsymbol{a^2} - \boldsymbol{b^2}}$
  If **c** < $r$, there are no intersection points. If c=r there is only one intersection point. Otherwise:
  $\boldsymbol{d} = \sqrt{\boldsymbol{r^2} - \boldsymbol{c^2}}$
  $\boldsymbol{p_1} = \boldsymbol{po} + (\boldsymbol{d} - \boldsymbol{b}) * \boldsymbol{dir}$
  $\boldsymbol{p_2} = \boldsymbol{po} + (\boldsymbol{d} + \boldsymbol{b}) * \boldsymbol{dir}$



# Ray-box intersection

- Given an axis aligned box with min and max bounds $B_0$ and $B_1$. To find first and last intersection with ray *r*, solve the following:
- $t_{0x} = \frac{B_{0x} - po}{d_x}$

  $t_{1x} = \frac{B_{1x} - po}{d_x}$

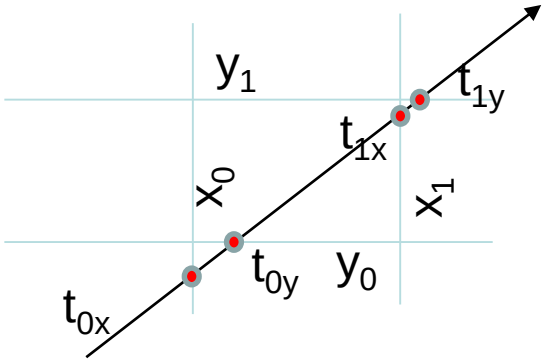
  $t_{0y} = \frac{B_{0y} - po}{d_Y}$

  $t_{1y} = \frac{B_{1y} - po}{d_Y}$

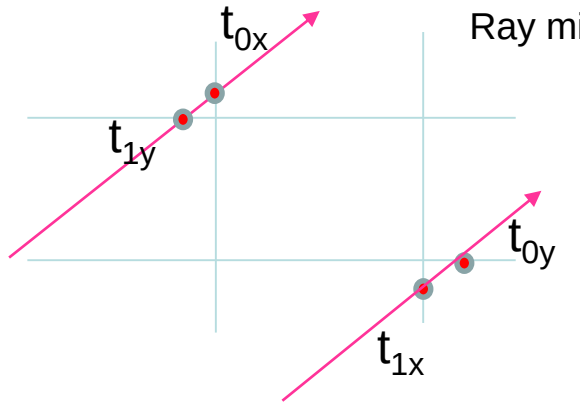
  $t_{0z} = \frac{B_{0z} - po}{d_z}$

  $t_{1z} = \frac{B_{1z} - po}{d_z}$
- Select smallest positive t per coordinate to find first intersection. Select largest positive t per coordinate for last intersection. Return no intersection for $min_{so\_far} > max_{current}$ || $max_{so\_far} > min_{current}$

Ray hit

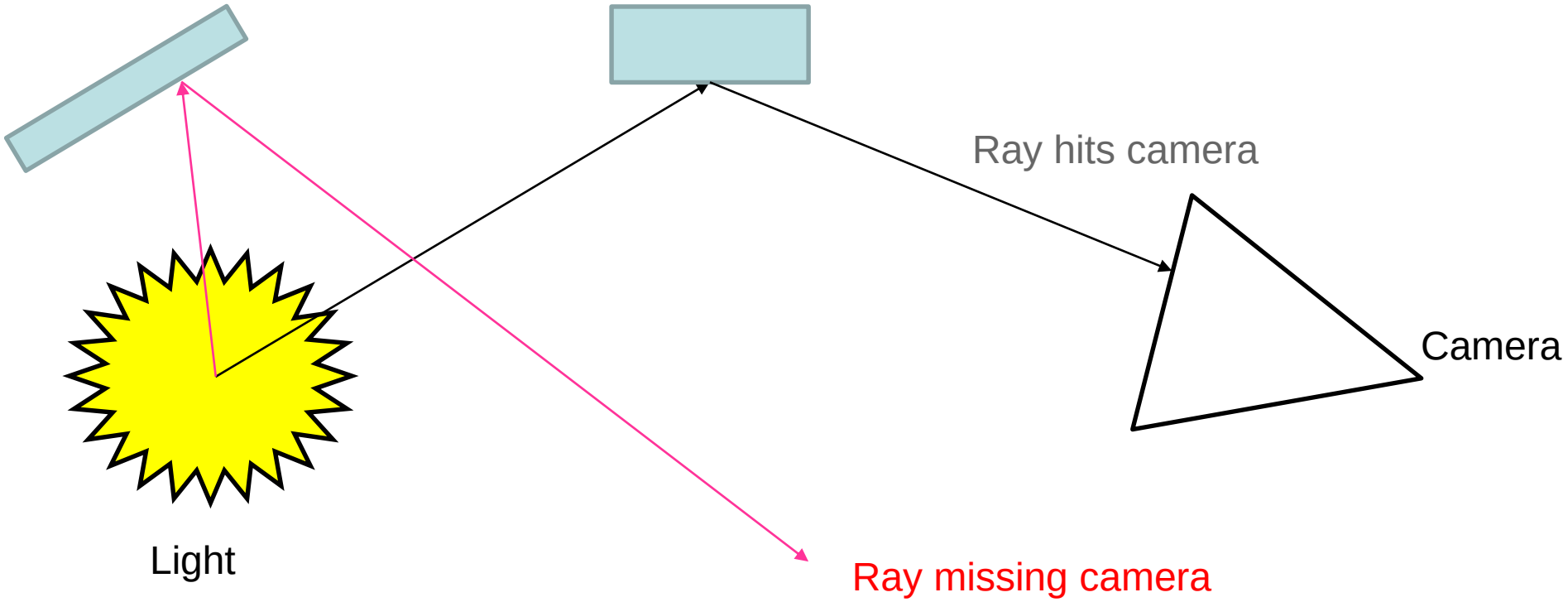



Ray miss



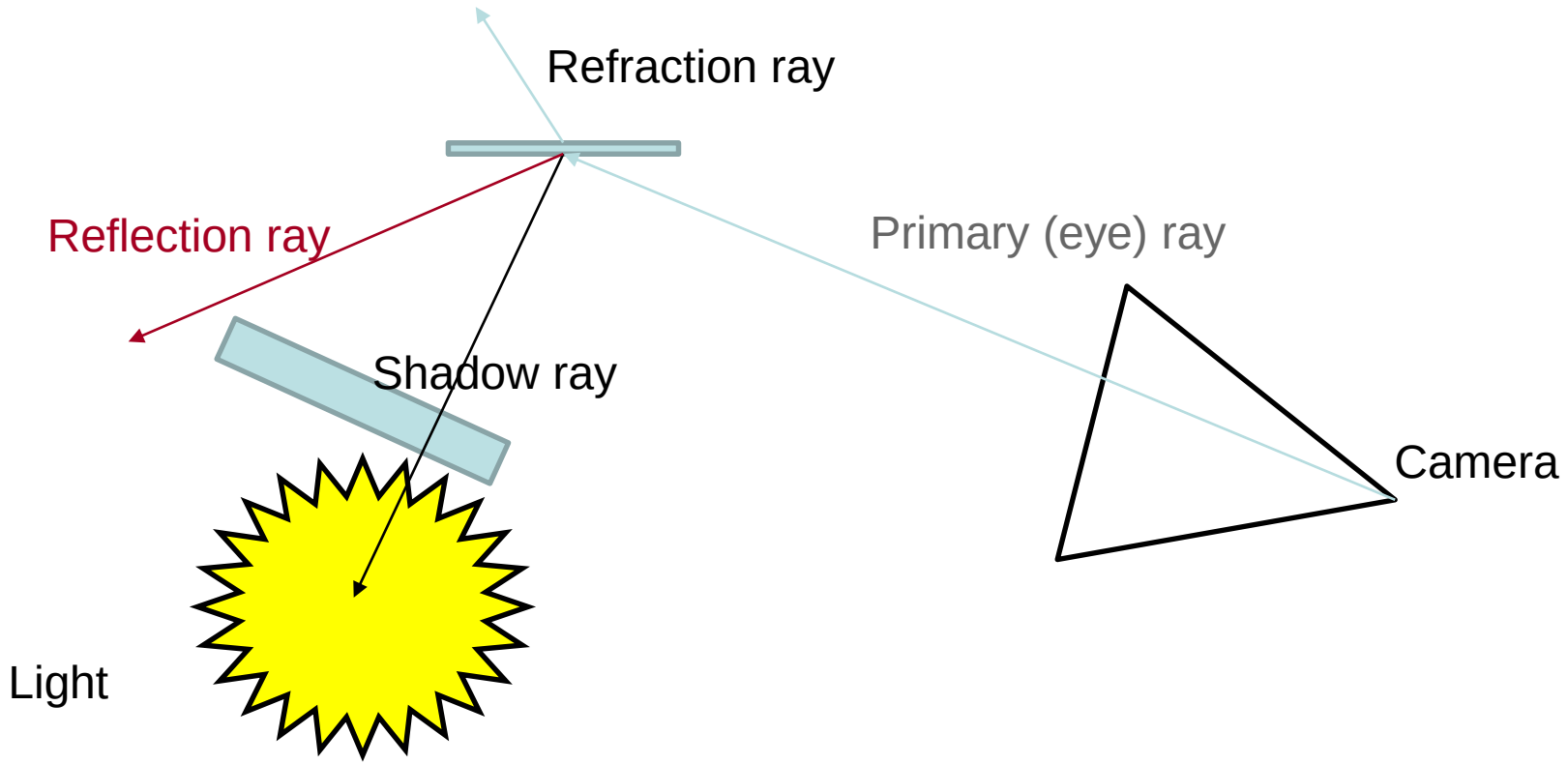
# Forward ray tracing

- Start shooting rays from the light sources. Follow light path through the geometry. Add light carried by rays that intersect the camera lens to the pixels of the final image.



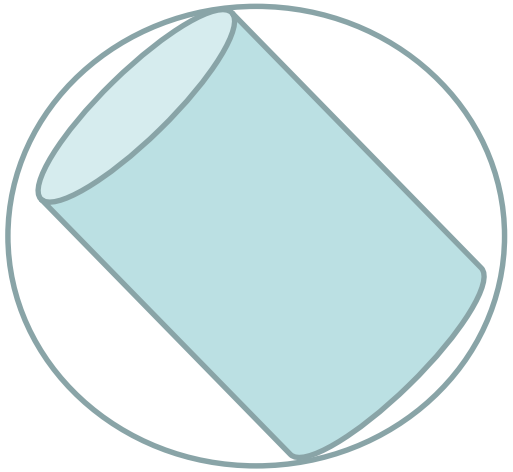
# Backward ray tracing

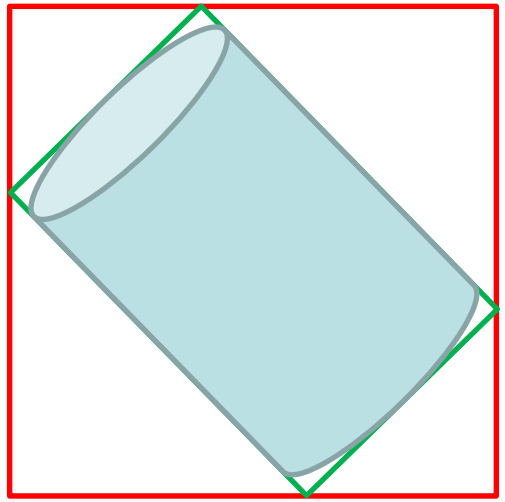
- Start shooting primary rays from the eye position(s) according to the lens model of the camera. Follow rays through the geometry. Collect contribution of light to the ray and write it to the image.



# Bounding volume hierarchy

- Aka object partitioning scheme
- Bounding spheres
  - easy to test intersection
  - Volume does not cover geometry very closely.
- Object aligned bounding boxes
- Axis aligned bounding boxes





# Binary space partitioning

- Aka spatial partitioning scheme
- Octree
  - Each internal node has 8 children. Splits space along the XY, XZ and YZ planes placed on the center of the current AABB. Children represent AABBs generated by the split.
- k-d tree
  - Select axis based on depth (cycle through all axes).
  - Select median by axis.
  - Split space into left and right child based on median.
  - Surface Area Heuristic (SAH) for node B and children $B_1$ and $B_2$

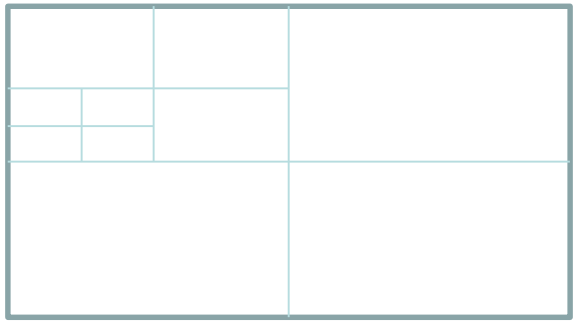
    P primitive count for node

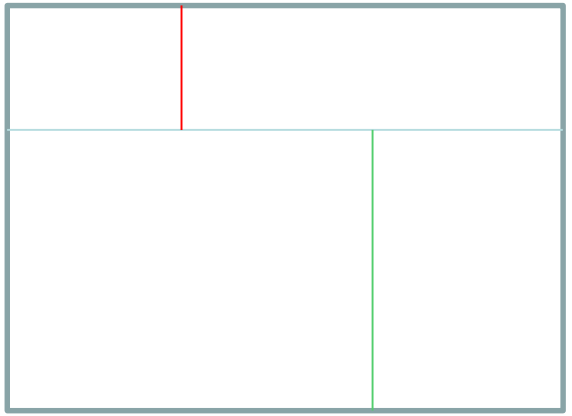
    SA sum of surface area for all triangles in P.

    $C_t$ is the cost of a traversal step

    $C_i$ cost of a single ray-primitive intersection

$$C = C_t + \frac{SA(B_1)}{SA(B)}|P_1|C_i + \frac{SA(B_2)}{SA(B)}|P_2|C_i$$





# SBVH

- http://www.nvidia.ca/docs/IO/77714/sbvh.pdf
- Split bounding volume hierarchy.
- Mixes object and space partitioning.
- Can reference the same triangle in multiple bounding boxes.
- Algorithm:
  - Find an object split
  - Find a spatial split
  - Select the winner based on SAH
- Binning is used to speed up spatial splits
- Holds the complete AABB (as opposed to kd-trees)

# TrBVH

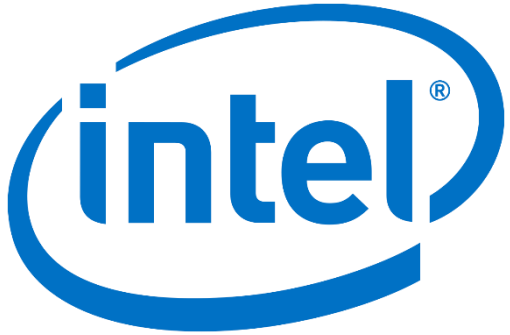
- http://research.nvidia.com/sites/default/files/pubs/2013-07_Fast-Parallel-Construction/karras2013hpg_paper.pdf
- Fast builds
- Constructs low quality BVH as first step
- Uses a novel heuristic for splitting triangles before BVH construction.
- Stores nodes in 60bit Morton code. Initially stores single triangle per node.
- Proceeds with searching for optimal topology at local neighborhood of nodes.
- Bottom up construction with dynamic programming.

# Performance

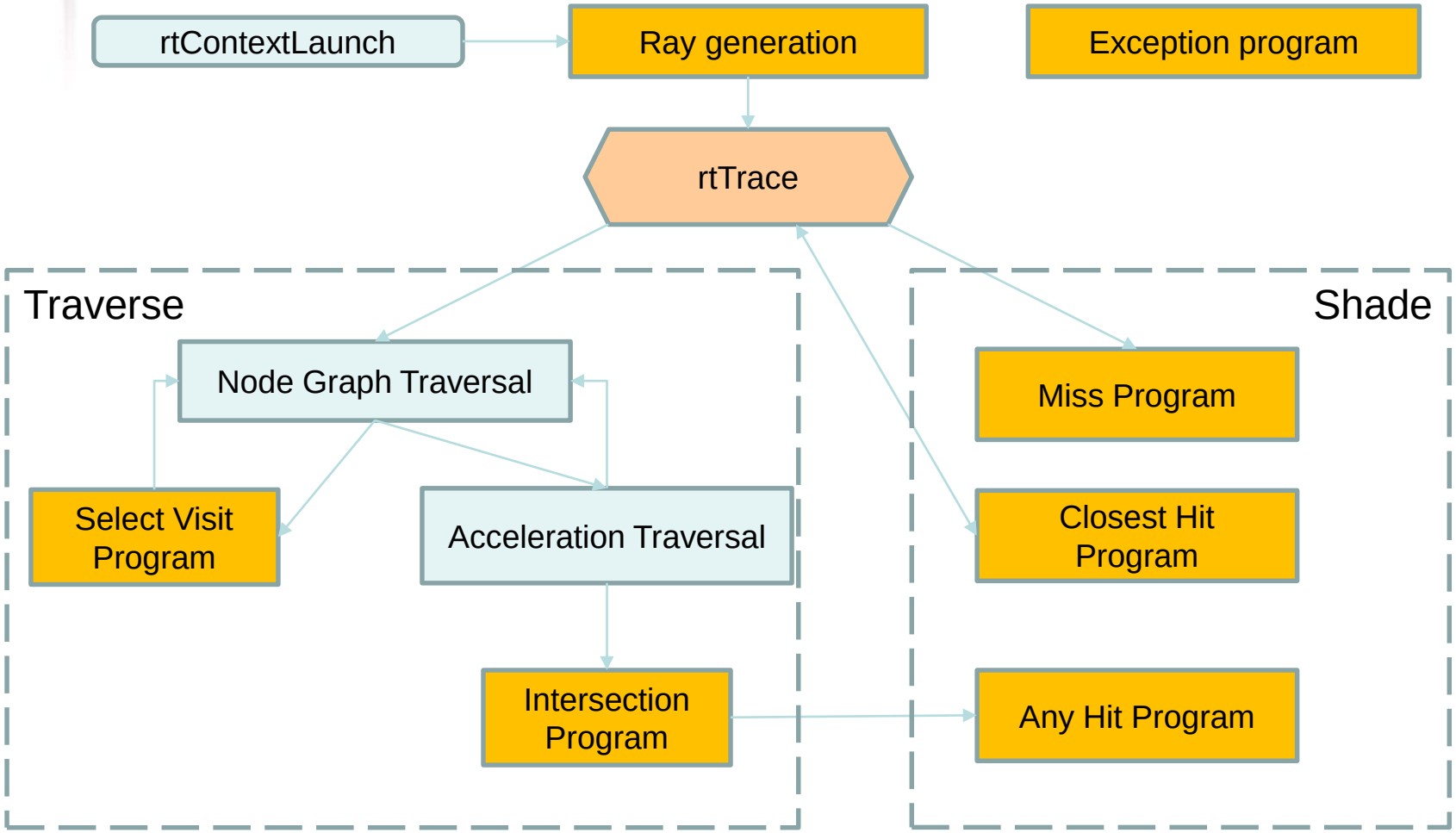
- Below 7M rays/frame use BVH
- 7M-60G rays/frame use TriBVH
- Above 60 G rays/frame use SBVH

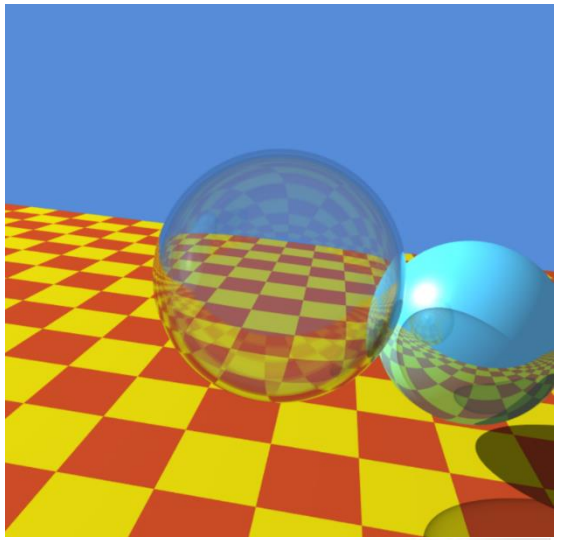
# Intel Embree

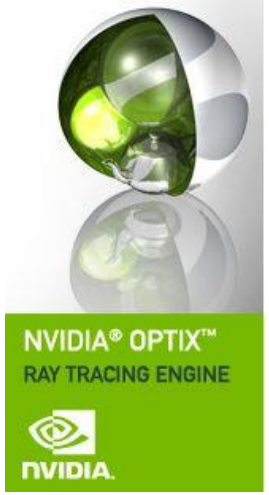
- http://embree.github.com
- CPU ray-tracer.
- Support for triangle and quad meshes, Catmull-Clark subdivision surfaces, cubic spline curves, user defined geometries
- Supports motion blur
- Supports LBVH, STBVH (4D Spatial-Temporal BVH)



# NVIDIA Optix



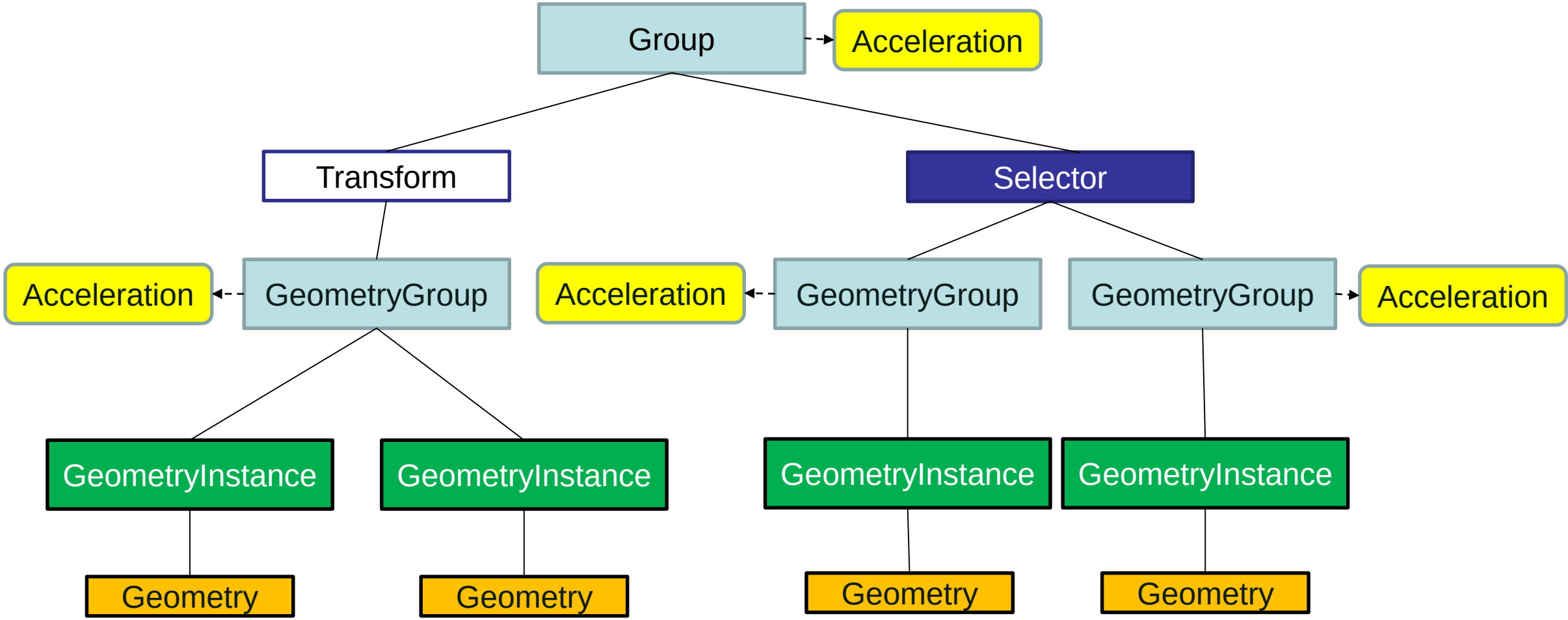





# NVIDIA Optix features

- https://developer.nvidia.com/optix
- Currently version: 5.1, based on CUDA
- Supports bounding box generation (no bounding spheres)
- Supports custom intersection (no default intersection available)
- Supports instancing
- Supports TrBVH, SBVH, and BVH
- Supports bindless buffers and bindless textures
- Supports cube, layered and mipmapped textures
- Supports motion blur
- Traverse and shade are done in the same call -> stack depth is limited
- Built-in AI-Accelerated denoiser (uses Tensor Cores)

# Optix Scene management



# Radeon-Rays

- https://github.com/GPUOpen-LibrariesAndSDKs/RadeonRays_SDK
- Intersection/occlusion query only!
- Supports OpenCL, Vulkan and Embree.
- Designed for AMD architectures, but works on anything with OpenCL support, e.g. NVIDA, Intel.
- Support for
  - BVH, QBVH, hierarchical linear BVH (fast build for dynamic scenes)
  - SAH, median

# Baikal renderer

- https://github.com/GPUOpen-LibrariesAndSDKs/RadeonProRender-Baikal
- High level ray tracer renderer based on Radeon Rays
- Support for materials, textures (no texture arrays), lights, volumes, etc.
- Support for Monte Carlo ray tracing, motion blur
- Post processing (denoisers)
- Traverse and shade are done in separate calls, no stack depth!

# Baikal example

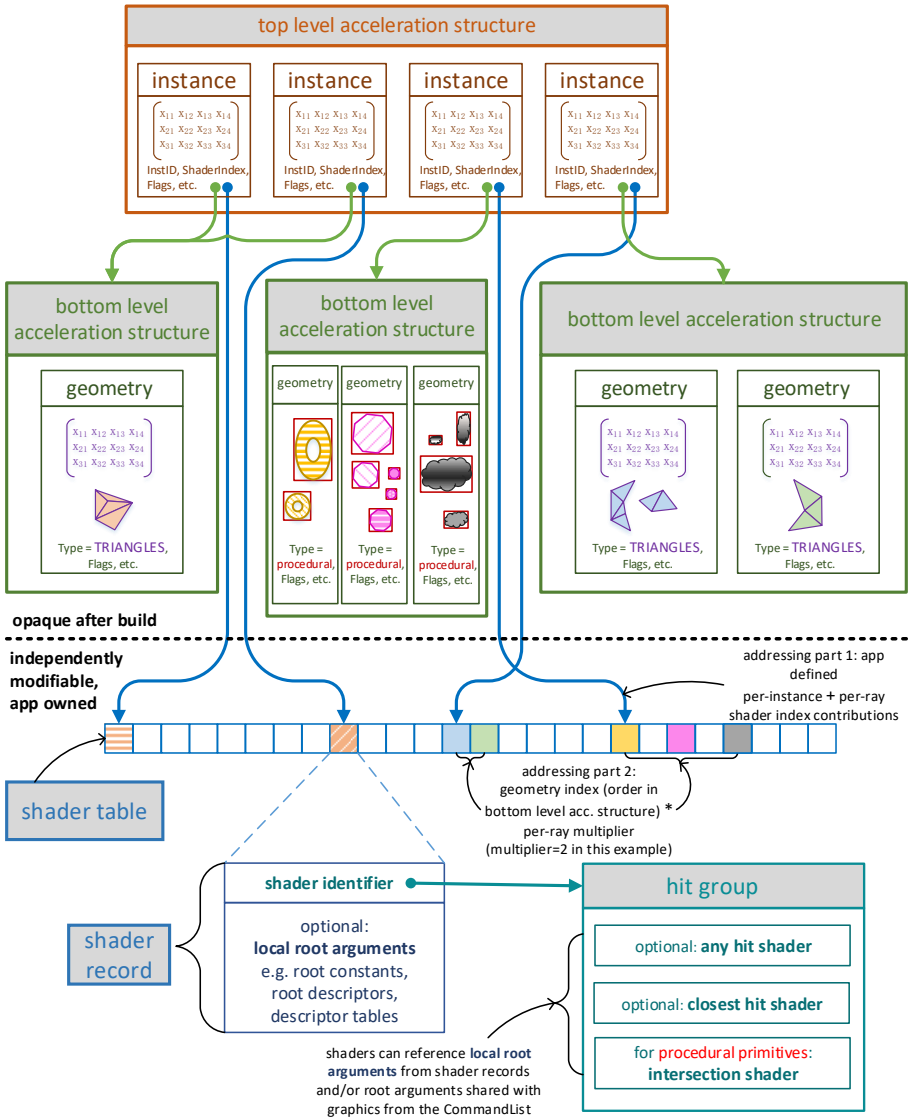
# Microsoft DirectX Raytracing API

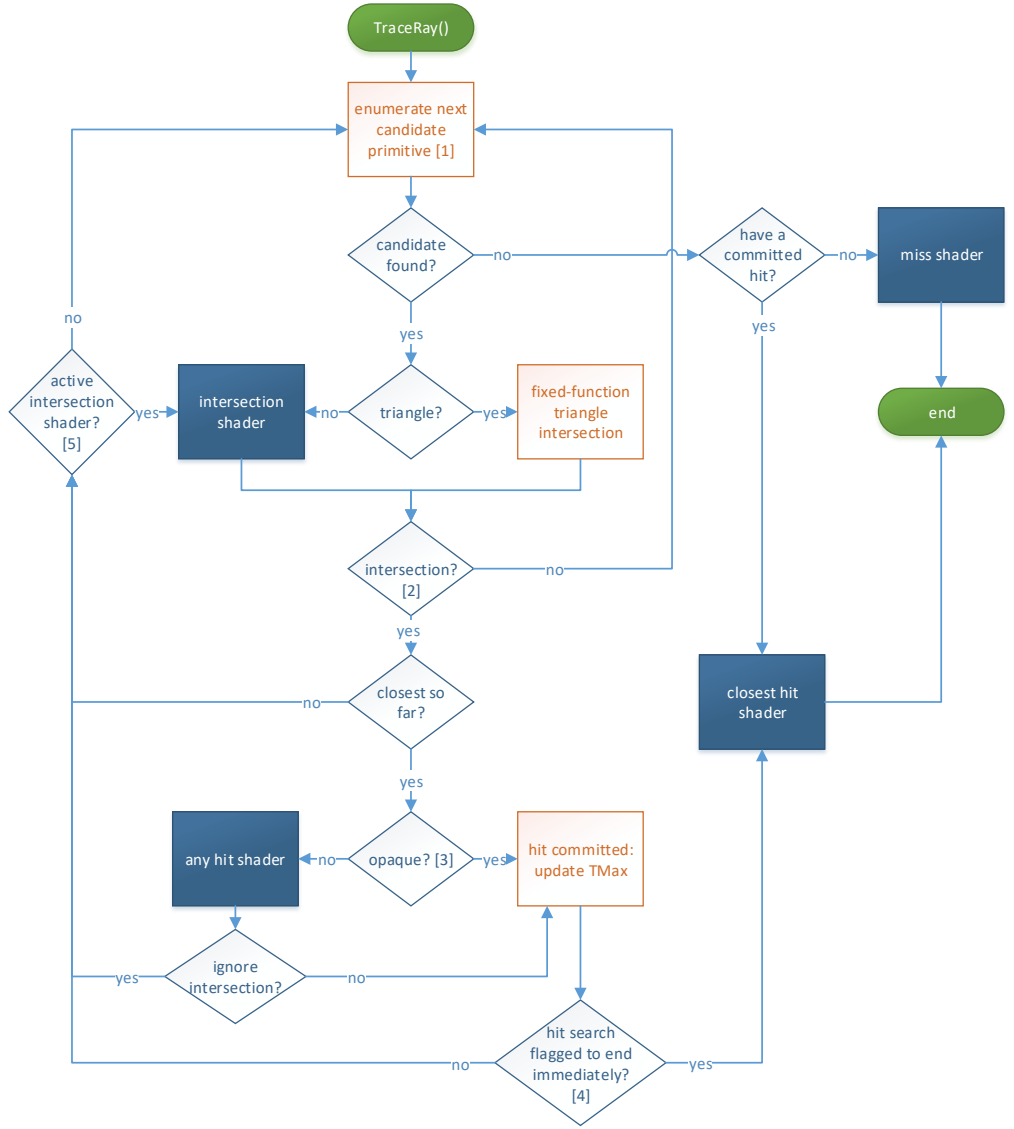
- https://github.com/Microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Raytracing
- Building and traversing the acceleration structure is left to the driver
- Triangle geometry is handled automatically, attributes are barycentrics
- Allows for custom (procedural) geometry, which need intersection shaders
- Everything allowed that is allowed in DirectX compute shaders
- Additional functions for ray generation, ray traversal accept/abort, etc.
- Writes results into UAVs
- Traverse and shade are done in the same call -> stack depth is limited
- Hardware acceleration on NVIDIA Volta hardware
- Fallback layer for other hardware
- Profiling is available, PIX

Microsoft

DirectX 12

# Microsoft DirectX Raytracing API



# DirectX Whitted ray tracing example

# Designing GPU ray tracers

- Different functions task the GPU differently
  - Ray generation load depends on camera model
  - BVH traversal is heavy on memory, scalar
  - Shading is heavy on arithmetic (transcendental), vector
- Ideally, we would split workloads to separate traversal and shading (only done on RadeonRays).
- From rasterizer engine to tracer engine.
  1. Transform feedback geometry to common vertex buffer. Separate static and dynamic scene
  2. Pack materials and lights to SSBOs, use bindless textures for material maps, light masks, etc.
  3. Use CUDA/OpenCL/D3D interop to transfer data if necessary
  4. Use GPU ray tracer library
  5. Use interop to transfer data back to rasterizer if necessary
  6. Upload to texture from pixel unpack buffer
  7. Use in subsequent draw calls/compute launches

# Rendering techniques for linear cameras

- Render G-buffer and use per pixel values to add effects
  - World space positions (or depth buffer)
  - World space normals
  - Material parameters
- Raytrace at half resolution + spatiotemporal filter

# Ambient occlusion example

- Trace rays from visible surface geometry to check for occlusions.

# Shadows

- Trace shadow rays to lights
- Trace multiple rays for area lights/soft shadows
- Filter traced shadow rays.

# Reflections

- Add single reflection ray + additional rays where luminance is high

# Real-time ray tracing at Holografika

- Implemented on OpenGL/GLSL
- Support for primary ray generation of Holografika light fields with multisampling
- Whitted ray tracer support
- No need for interop!

# Anatomy of the Holografika tracer

- Generation of primary rays: OpenGL compute shader.
- Generation of BVH tree: C++ code for CPU (unoptimized).
- Method: binned SAH division (see Wald et al.).
- After being constructed, BVH is uploaded into OpenGL buffer.
- Update of BVH tree: none (static scenes only).
- BVH tree traversal: OpenGL compute shader.
- Method: improved version of stack-less traversal from Hapala et al.
- Intersection data is collected in tree-like structure inside OpenGL buffer.
- Evaluation of color: another OpenGL compute shader.
- Intersection data is obtained from traversal pass.
- After this, the intersections are traversed recursively (similar to conventional ray tracing).

# Holografika real-time ray tracing example

# Acknowledgements


- This presentation has been created with the support of NKFIH, in the project KFI_16-1-2017-0015 Medical Holo-platform, funded by the Company R&D&I (Vállalati KFI_16) program.
- (Ez az előadás a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal (NKFIH) támogatásával, a Vállalatok K+F+I tevékenységének támogatása (Vállalati KFI_16) program által finanszírozott KFI_16-1-2017-0015 Orvosi Holo-platform projekt keretében készült.)
- The work in this paper was funded from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 676401, European Training Network on Full Parallax Imaging.
- The research in this paper was done as a part of and was funded from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 676401, ETN-FPI, and No 643072, Network QoE-Net.

# Questions?