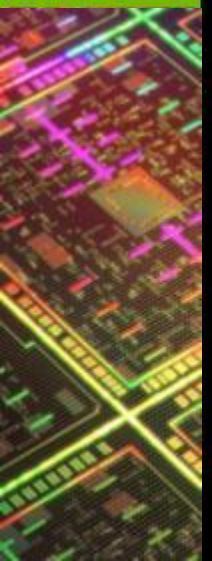
GPU Ray Tracing

May 14, 2012

S0603 - GPU Ray Tracing

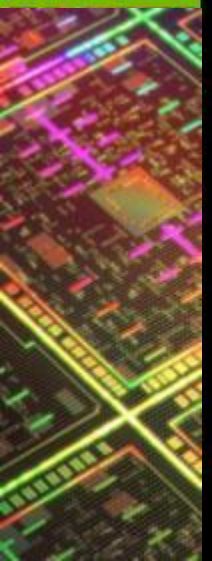
Learn the latest approaches in levering GPUs for the fastest possible ray tracing results from experts developing and leveraging the NVIDIA OptiX ray tracing engine, the team behind NVIDIA iray, and those making custom renderers. Multiple rendering techniques, GPU programming languages, out-of-core rendering, and optimal hardware configurations will be covered in this cutting-edge discussion.

Topic Areas: Ray Tracing
Session Level: Beginner



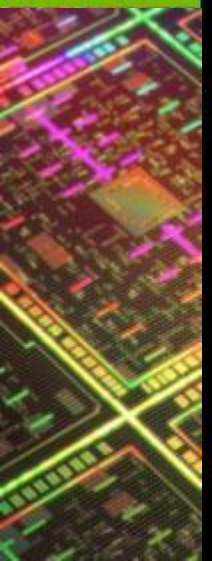
Agenda

- Introduction (with Phillip Miller)
- GPU Ray Tracing Basics
- Introduction to OptiX
- Deeper Dive on OptiX (with David McAllister)
- What's coming next in OptiX



NVIDIA Ray Tracing Options

- **CUDA** – language and computing platform
  - The basic choice for building *entirely custom solutions from scratch*
- **OptiX** – middleware for ray tracing developers
  - Good choice for developers *with domain expertise* building *custom solutions* which prefer *leaving GPU issues to NVIDIA*
- **mental ray & iray** – a licensed rendering products
  - Good choice for companies wanting a *ready-to-integrate solution* which is *maintained and advanced for them*



Evolving Views on GPU Ray Tracing (it)

2007: The future is ray tracing – and GPU's *can't do it*

2008: NVIDIA can do it, but *we can't* (NV demo)

2009: Now *everyone can* do it (Papers, OptiX)

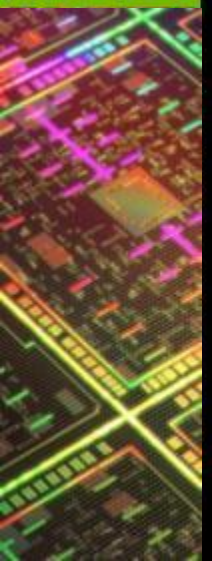
2010: *Many are* doing it (Demos, 3K downloads)

2011: It's becoming mainstream (Adobe, Autodesk, DS)

2012: It's limitations are fading (Paging, CPU Fallback)

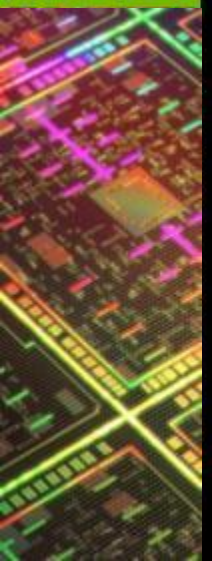
GPU Ray Tracing Examples

- Production:
  Delta Tracing
  (near Venice)
- Client:
  DVO
  (Della Valentina
  Office SpA)
- Virtual Catalogues



GPU Ray Tracing Myths

1. The only technique possible on the GPU is “path tracing”
   FALSE: Ray Tracing Techniques are only limited by C
2. You can only use (expensive) Professional GPUs
   FALSE: GPU computing languages run on all GPUs
3. A GPU farm is more expensive than a CPU farm
   FALSE: Much better Perf/$; and Perf/Watt on Kepler
4. A GPU isn’t that much faster than a good CPU
   FALSE: A single GPU is typically 4-12X a quad-core
5. GPU Ray Tracing is very difficult
   Possibly: OptiX speeds both ray tracing and GPU devel.
6. Scenes must fit into GPU memory – and that’s finite
   Not Always: Out-of-Core Support with OptiX 2.5

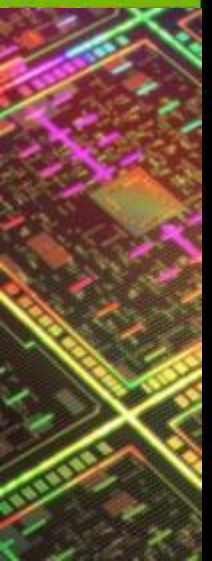


GPU Ray Tracing Facts

1. GPUs can accommodate any ray tracing technique a CPU can
2. Compute, and thus ray tracing, works on all GPUs
3. GPUs have superior performance (and maintenance) costs vs. CPUs
4. A single GPUs is considerably faster than multiple CPUs
5. OptiX makes both Ray Tracing and GPU development easier
6. Scenes can exceed GPU memory with OptiX 2.5 (up to system RAM)

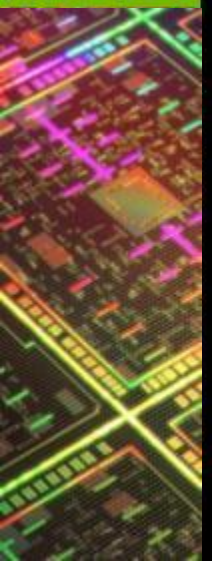
Demo – State of the Art Interaction

- GPU Ray Tracing and Physics



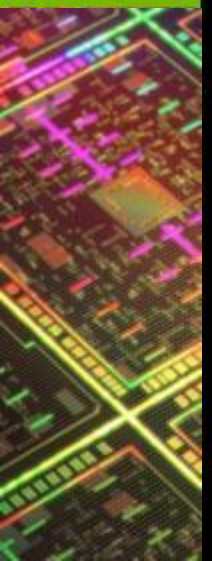
Commercial GPU Ray Tracing

▪ iray	CUDA C, C Runtime	
▪ V-Ray RT	CUDA C, Driver API and OpenCL	
▪ Arion	CUDA C, Driver API	
▪ Octane	CUDA C, Driver API	
▪ finalRender	CUDA C, C Runtime	
▪ LuxRender (open source)	OpenCL	
▪ CentiLeo	CUDA C, driver API	Out of Core
▪ Panta Ray (Weta)	CUDA C, driver API	Massive Out of Core
▪ OptiX (2.5)	CUDA C, driver API & PTX	(Out of Core)
▪ Adobe After Effects CS6	OptiX API	“ “
▪ Custom OptiX, Works Zebra, etc.	OptiX API	“ “
▪ mental ray 3.11 (in development)	OptiX API	“ “



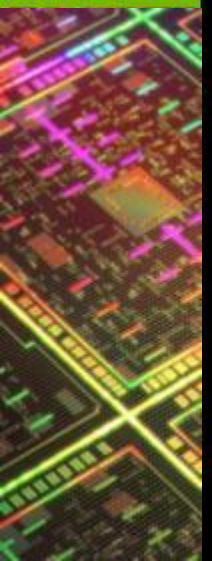
GPU Ray Tracing Similarities - Performance

- Single GPU Ray Tracing Speed
  - Usually linear to GPU cores and Core Clock – for a given GPU architecture
  - Gains between GPU generations will vary per solution, but they’re BIG
- Multi-GPU Ray Tracing Speed
  - Solution dependent, Common in Renderers, OptiX supports by default
  - Scaling efficiency varies by solution;
    slow techniques usually scale better than fast ones (e.g., AO vs. Whitted)
- Cluster Speed (multi-machine rendering)
  - Solution dependent, Uncommon in Renderers, OptiX doesn’t, Iray does



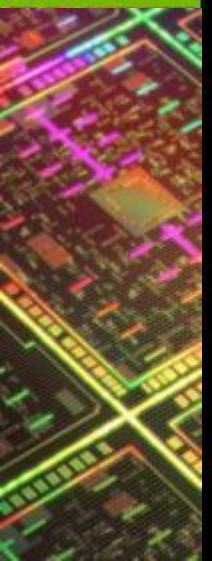
GPU Ray Tracing Similarities - Hardware

- "SLI" configuration is not needed for multi-GPU usage
- Nearly all renderers are Single Precision
- ECC driver choice (error correction) – **NOT** Recommended
  - No Accuracy Benefit; Slows Performance, Reserves ½ GB on a 3 GB board
- Windows 7 is a bit slower than Windows XP or Linux
- GPU memory size is often key
  - Entire scene must usually fit within GPU memory – to work AT ALL
  - Multiple GPUs can't "pool" memory; entire scene must fit onto each
  - If Out-of-Core is supported, performance degrades when paging
- Consumer GPUs not designed for "data center" usage



GPU Ray Tracing Similarities - Interaction

- GPU Computing (Ray Tracing) competes with system graphics
  - GPUs are still singularly focused: Compute **or** Graphics - not simultaneous
  - Often the **single biggest** design challenge for interactive app's
- Careful Application Design is needed to achieve balanced interaction
  - Gracefully stopping for user interaction and when app isn't focused
  - Controlling mouse pointers in the ray tracing app
- Or use Multi-GPU
  - One GPU for graphics, additional GPUs for compute (Ray Tracing)
  - Becoming mainstream with NVIDIA Maximus = Quadro + Tesla(s)

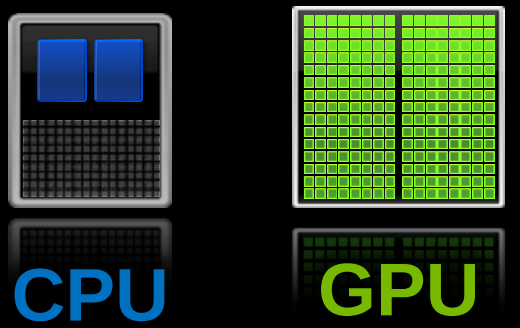


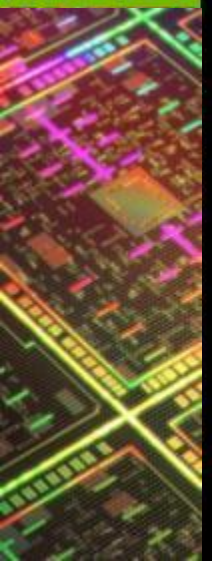
Multi-GPU Considerations for Development

- Differing GPUs can mean different Compute capabilities
  - Not just between architectures (e.g., Fermi vs. Kepler) but sometimes within an architecture (e.g., GF100 vs. GF104)
  - Either insist on consistency, program to lowest denominator, or have multiple code paths
- TCC (Tesla Compute Cluster) mode for Windows
  - Compute-only mode; GPU no longer a Windows graphics device
  - Not feature complete for multi-GPU memory accessing
  - Parity coming in CUDA 5.0 (this summer)

Solutions Vary in their GPU Exploitation

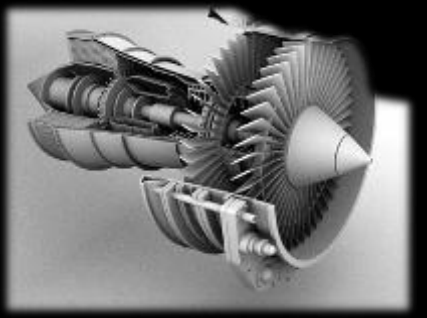
- Speed-ups vary, but a top end Fermi GPU will typically ray trace 6 to 15 times faster than on a quad-core CPU
- Constant CPU Compute challenge is to keep the GPU “busy”
  - Gains on complex tasks often greater than for simple ones
  - Particularly evident with multiple GPUs, where data transfers impact simple tasks more
  - Can mean the technique needs to be rethought in how it’s scheduling work for the GPU
  - Example OptiX 2.1: previous versions tuned for simple data loads, now tuned for complex loads, with a 30-80% speed increase





NVIDIA® OptiX™ ray tracing engine

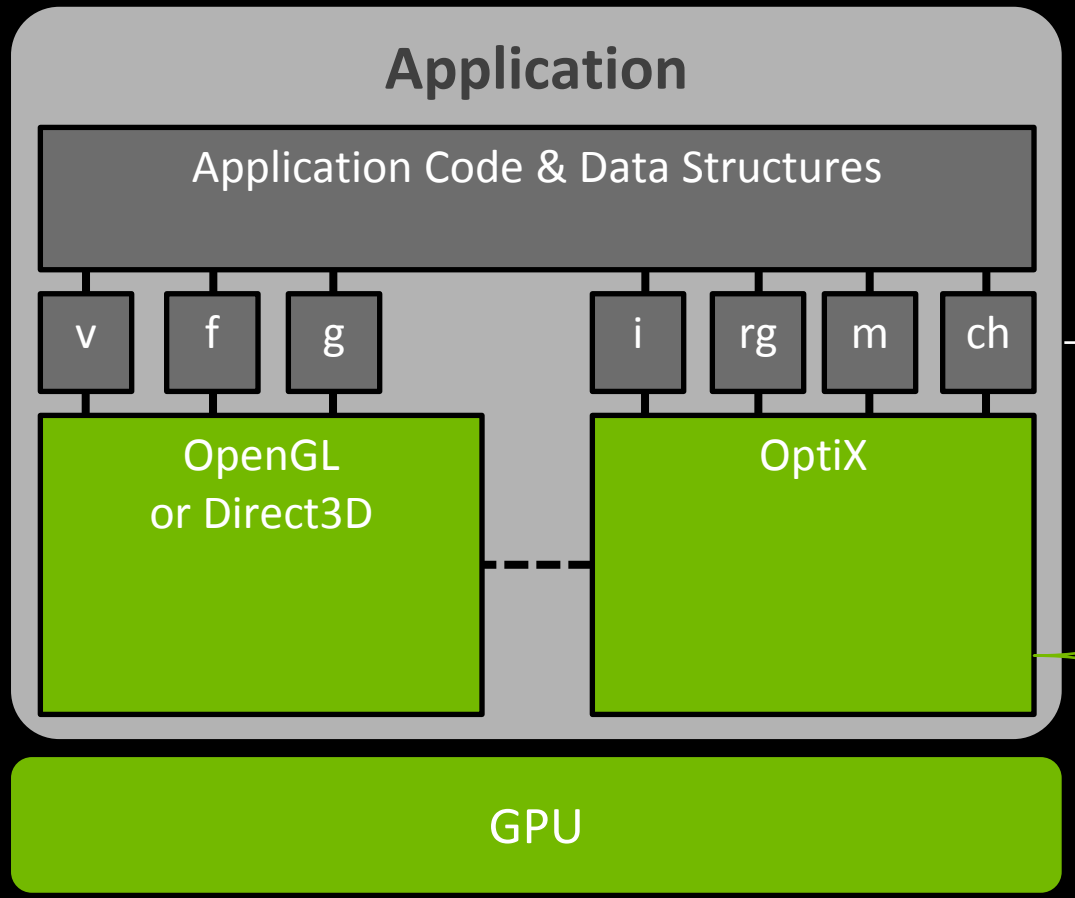
A programmable ray tracing framework enabling the rapid development of high performance ray tracing applications – from complete renderers to discrete functions
(collision, acoustics, ballistics, radiation reflectance, signals, etc.)



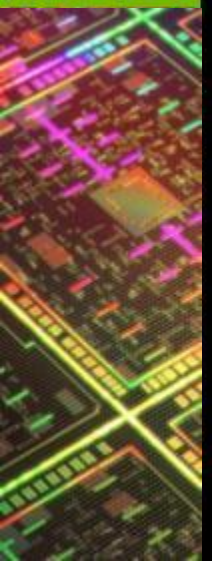
- Use your techniques, methods, and data for your application with simple programs –

- OptiX makes it fast on the GPU; abstracting both GPU interaction and the "heavy lifting" of ray tracing into easy-to-use APIs

OptiX - similar to OGL in "Approach"

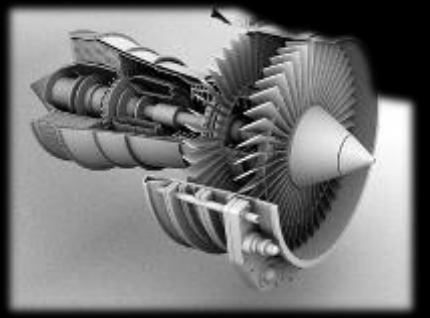


- C-based Shaders/Functions (minimal CUDA exp. reqd.)
  - Small, Custom Programs
  - Acceleration Structures Build & Traversal
  - Optimal GPU parallelism and Performance
  - Memory Management
  - Paging



NVIDIA® OptiX™ ray tracing engine

- **Optimal performance**, from unique insights and methods for the latest GPU capabilities –without needing to code for new GPU architectures.
- Easy to use, single ray programming model
- Supports custom ray generation, material shading, object intersection, scene traversal, ray payloads
- Programmable intersection for custom surface types (procedurals, patches, NURBS, displacement, hair, fur, etc.)
- No assumptions on technique, shading language, geometry type, or data structure



OptiX - in Use

+3k downloads per version

Adobe

LIGHTWORKS
Rendering Realism

WORKS ZEBRA

AUTHOR

ASPECTS

ARTISAN

TOYOTA

NEED FOR SPEED

Privately being used at companies doing:

- Content creation tools
- Post production
- Next-Generation Gaming
- Massive On-Line Player Games and Services
- Acoustics
- Ballistics
- Multi-Spectral Simulation
- Radiation & Magnetic Reflection

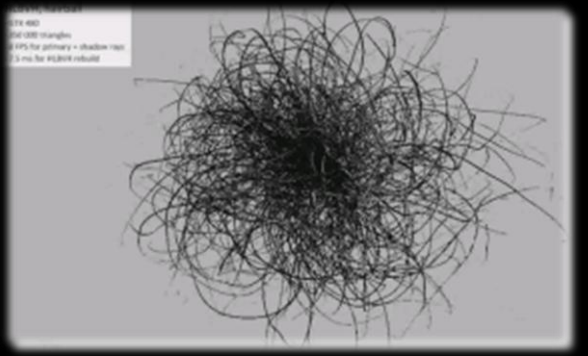
Adobe After Effects CS6 - using OptiX

New 3D compositing with ray traced production renderer

- From scratch, in 1 release cycle
- 100% OptiX – no x86 code
- Includes CPU Fallback
  - Via LLVM in OptiX
  - Currently unique to Adobe
  - Direct from PTX to X86 without the need of an NVIDIA driver

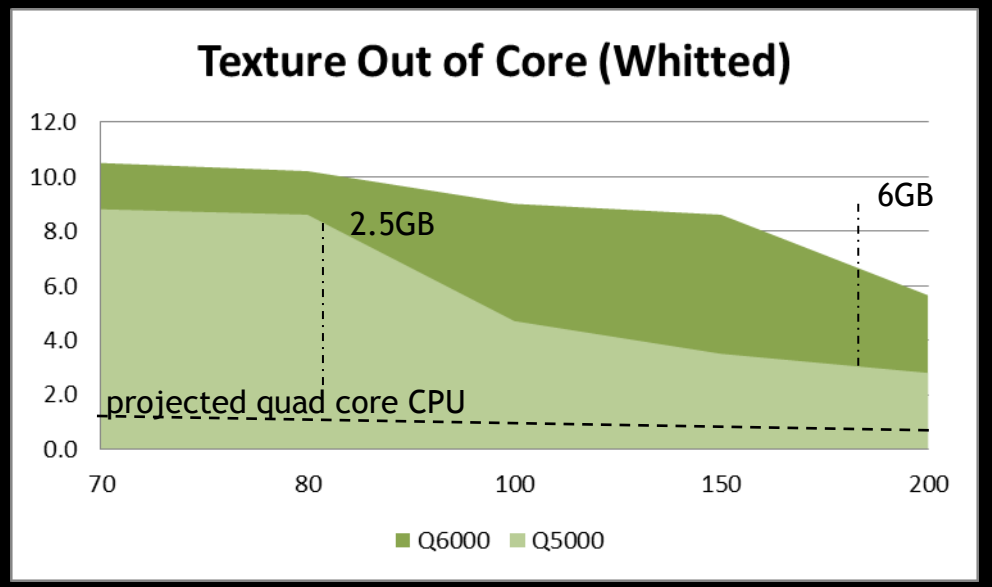
OptiX - Rapid Evolution

- Version 1, November 2009
  in use across many markets
- Version 2, August 2010
  exploited Fermi architecture for 2-5X speed increase
- Version 2.1, January 2011
  64-bit PTX, with +50% perf. on complex techniques, initial CPU fallback
- Version 2.5, April, 2012
  Memory paging, GPU accel. Structure build
- Version 2.5.1 Soon
  Kepler compatibility
- In progress, for summer 2012
  Features important for interaction, plus Kepler optimization

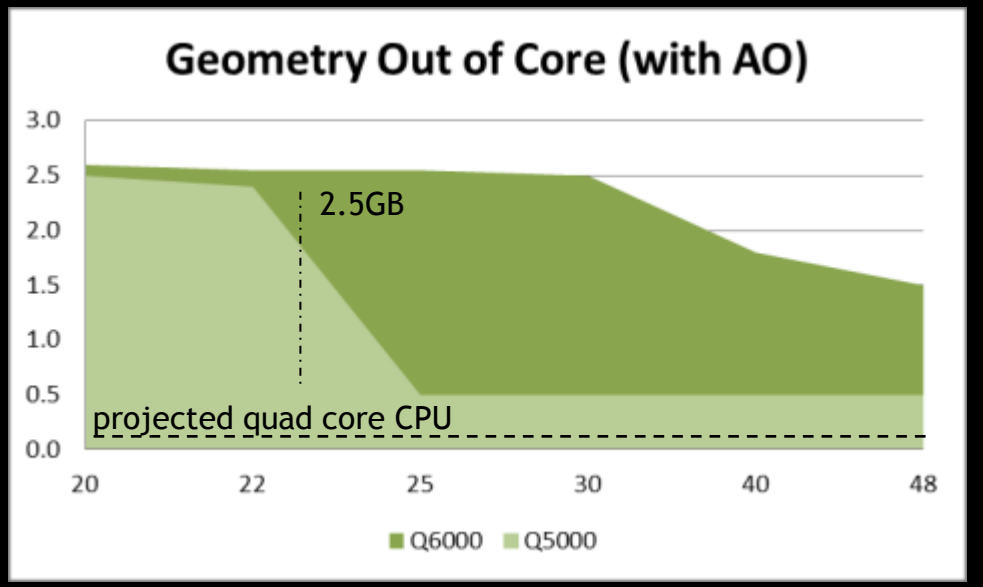


OptiX 2.5 Out of Core Performance

- Averaged results, as paging amount is view dependent



# of 4k Images



Millions of Textured & Smoothed Faces

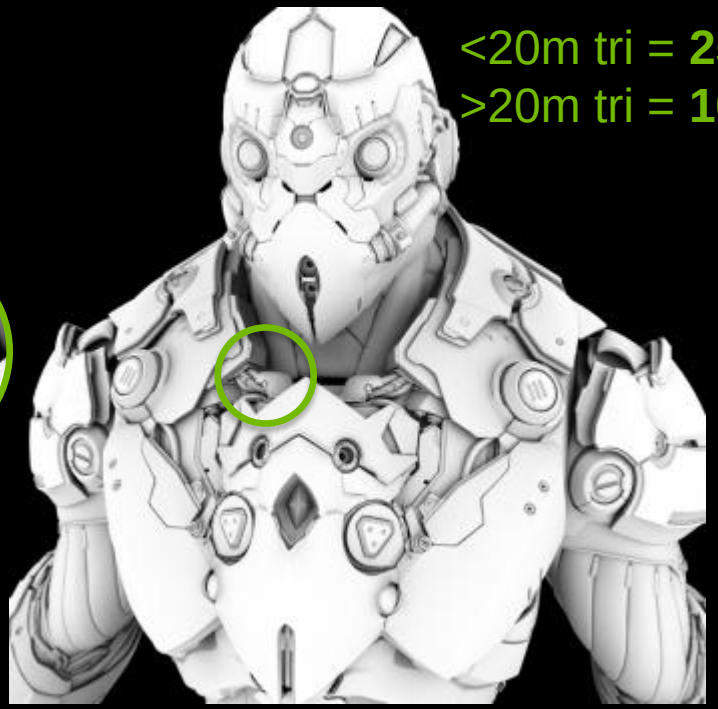
Quadro 6000 = 6GB on board memory

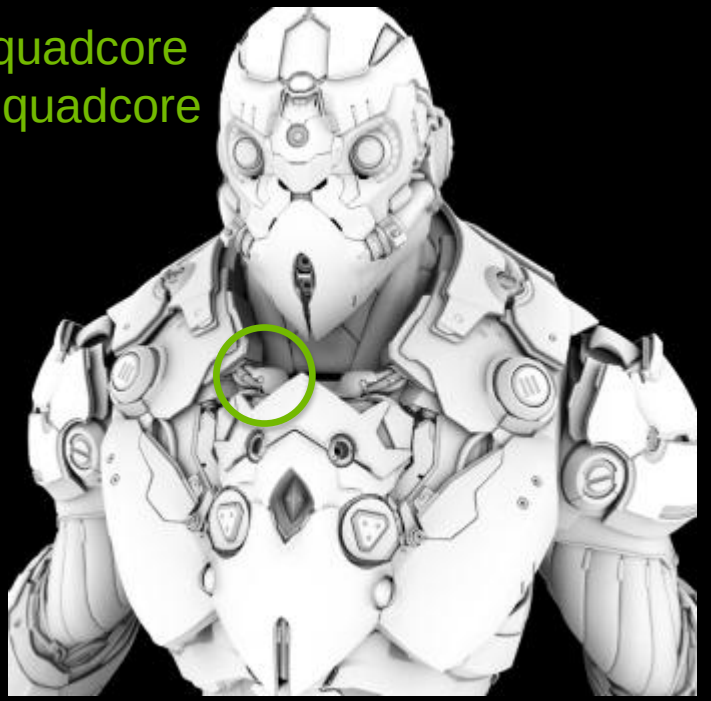
Quadro 5000 = 2.5GB on board memory

mental ray Ambient Occlusion

- mental ray* pipeline accelerated w/ OptiX

<20m tri = **25– 70X** quadcore
>20m tri = **10 – 20X** quadcore





Model courtesy NVIDIA Creative

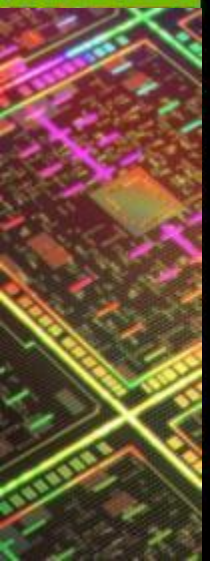
- 1.5sec HLBVH build + 15sec vs. 20 minutes on CPU

*no availability information announced yet for this functionality in mental ray version

NVIDIA Design Garage Demo

- Photorealistic car configuration made in 2010 for the GeForce community
- Built on SceniX with OptiX shaders
- Uses pure GPU ray tracing
  - Est. 40-50X faster vs. a CPU core
  - 3-4X faster on GF100 than on GT200
  - Linear scaling over GPUs & CUDA Cores
- Rendering development speed
  - 5 weeks
  - 2 renderers, 5 shaders, tone mapping, DOF, etc.

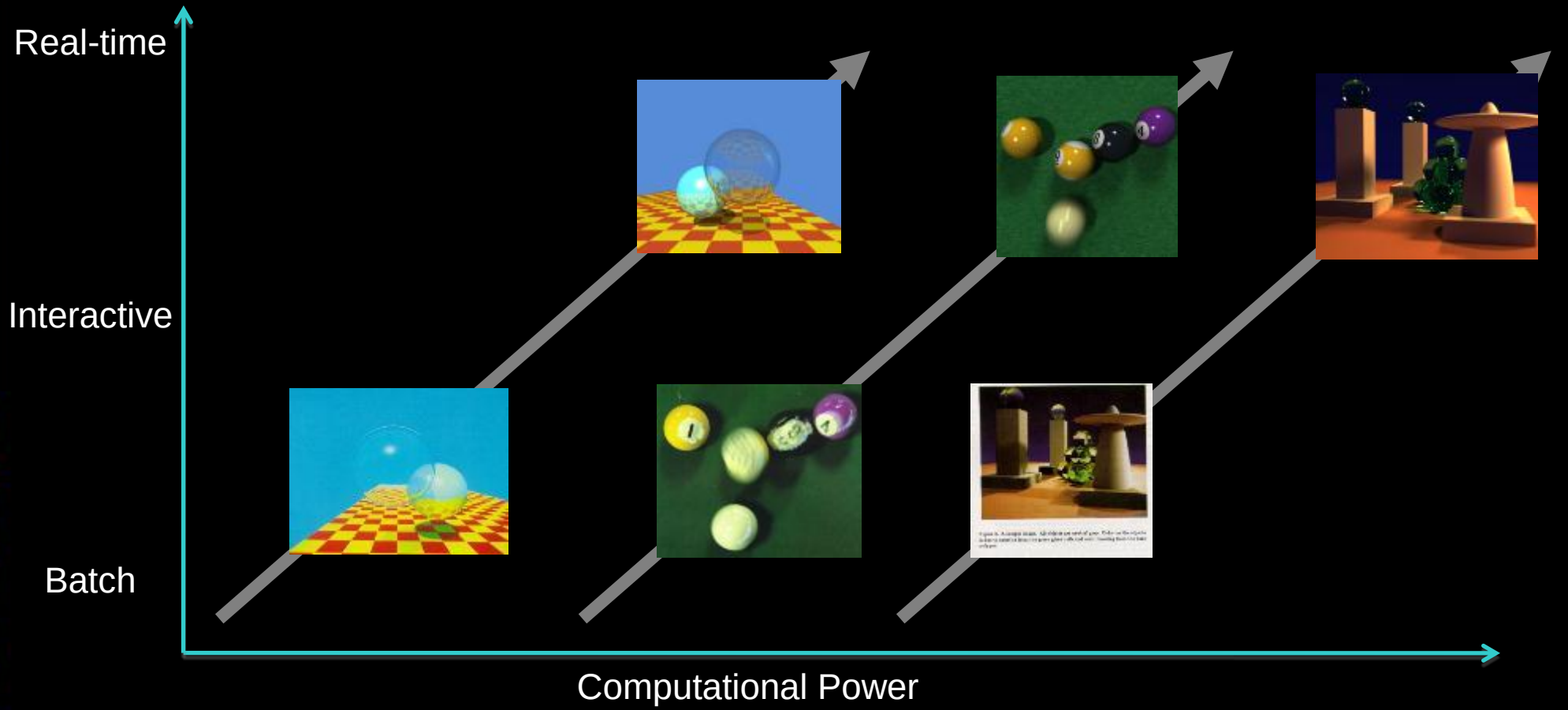
NEED FOR SPEED



OptiX - a bitter deeper dive

- David McAllister
  OptiX Development Manager
  NVIDIA

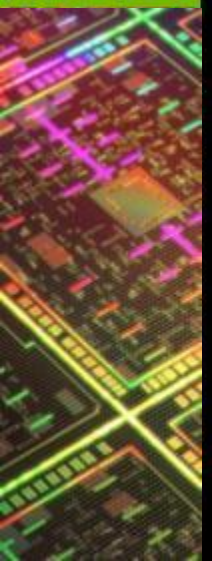
Ray Tracing Regimes



How to optimize ray tracing (or anything)

1. GPUs
2. Algorithmic improvement
3. Tune for the architecture

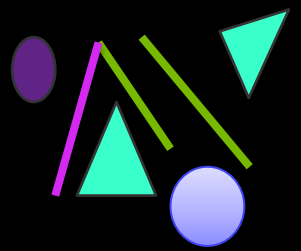
1. Better hardware
2. Better software
3. Better middleware



Acceleration Structures

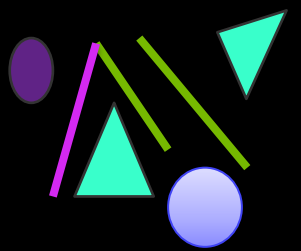
Bounding Volume Hierarchy

- Object centric
- Spatial redundancy
- Example: AABB BVH



Spatial Partitioning

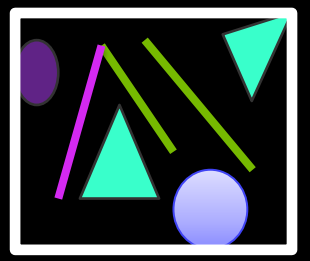
- Spatial centric
- Object redundancy



Acceleration Structures

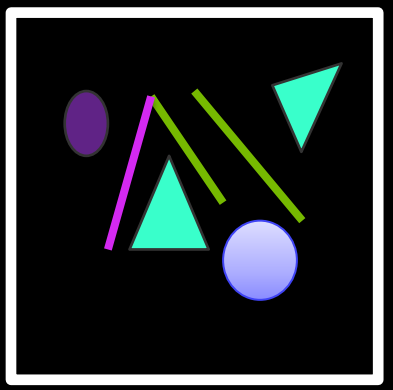
Bounding Volume Hierarchy

- Object centric
- Spatial redundancy



Spatial Partitioning

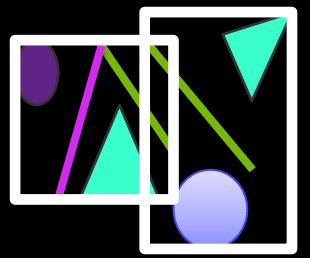
- Spatial centric
- Object redundancy



Acceleration Structures

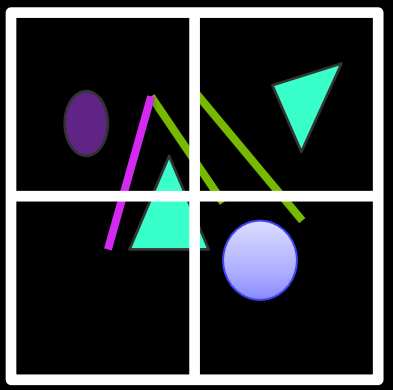
Bounding Volume Hierarchy

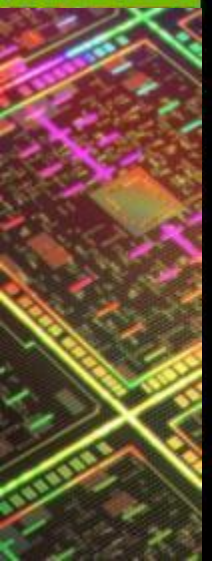
- Object centric
- Spatial redundancy



Spatial Partitioning

- Spatial centric
- Object redundancy

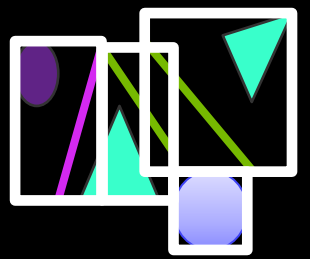




Acceleration Structures

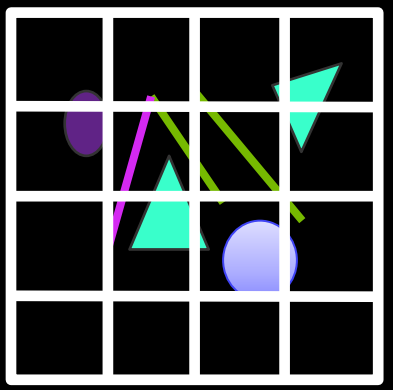
Bounding Volume Hierarchy

- Object centric
- Spatial redundancy



Spatial Partitioning

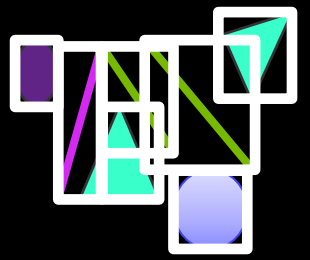
- Spatial centric
- Object redundancy



Acceleration Structures

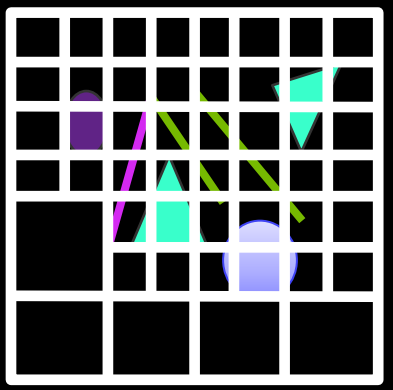
Bounding Volume Hierarchy

- Object centric
- Spatial redundancy



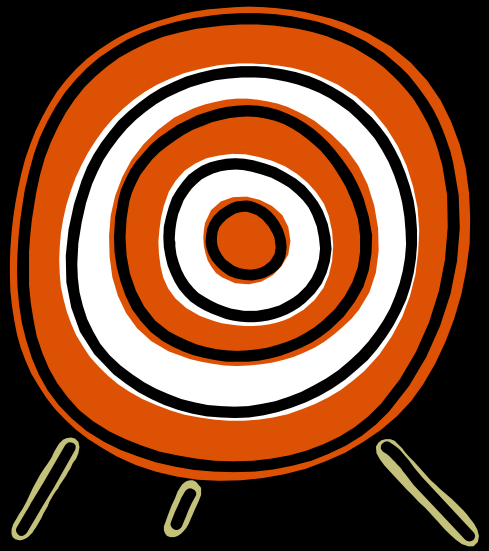
Spatial Partitioning

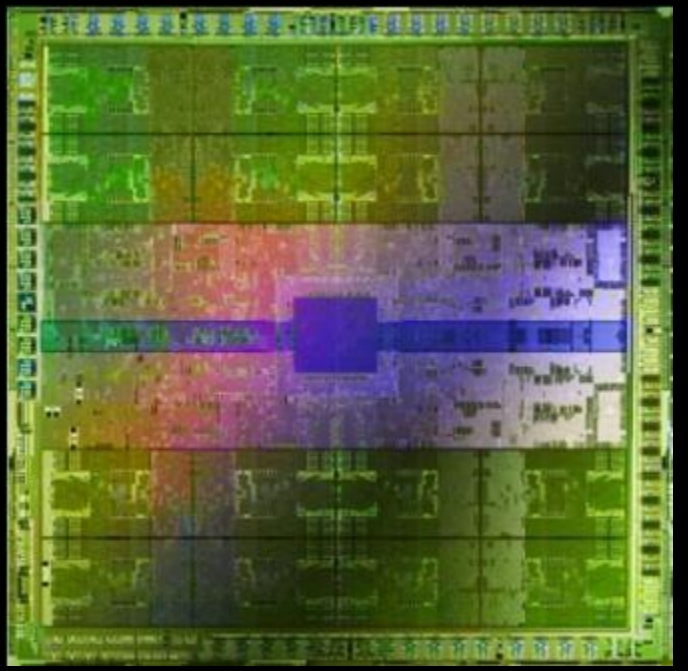
- Spatial centric
- Object redundancy



OptiX does the heavy lifting for you.

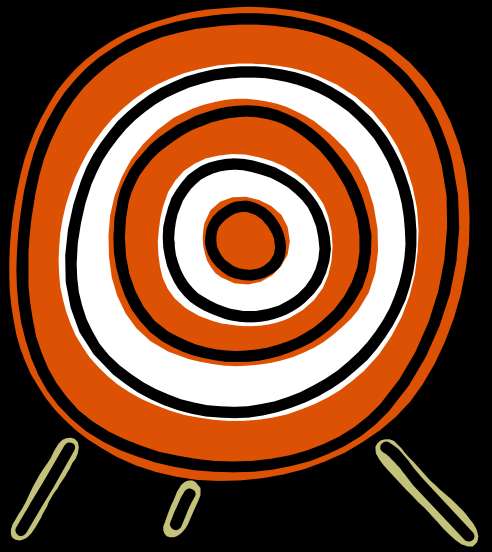
Target the specific architecture.

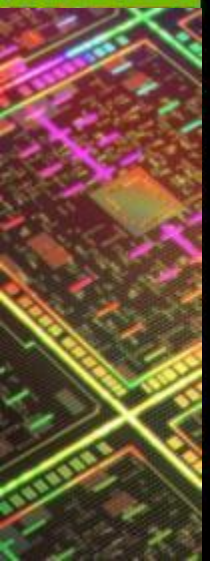


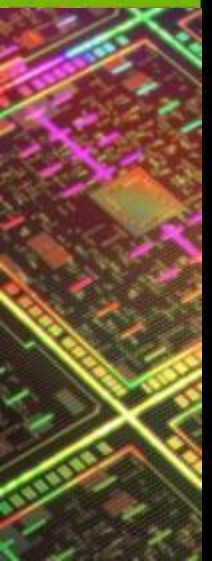


OptiX does the dirty work for you.

Target the next architecture.





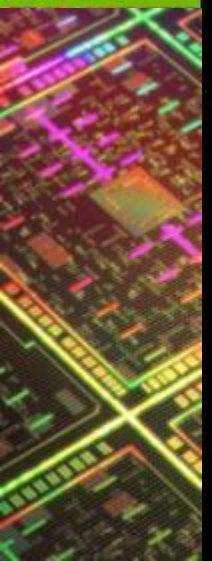


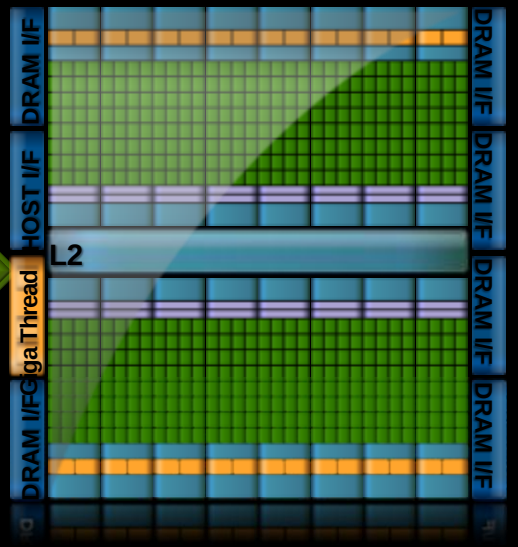
OptiX Goals

- Make GPU ray tracing simpler
- Function in a resource limited device
- Achieve high performance
- Express most ray tracing algorithms
- Leverage CUDA compiler infrastructure
  - No new shading language

Using OptiX

OptiX Functional Overview





Ray Generation

Material Shading

Object Intersection

Acceleration Structures

JIT Compiler

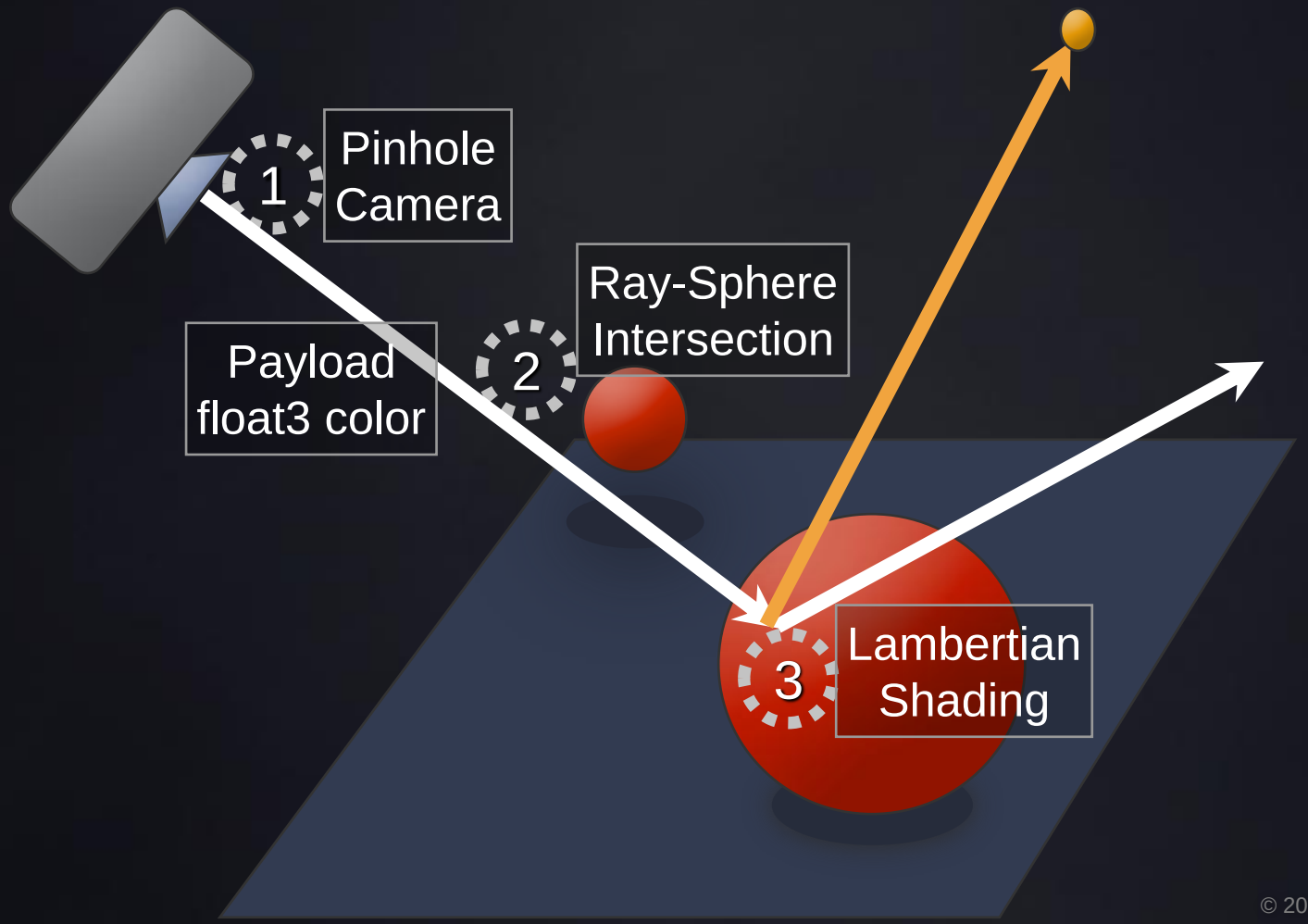
Scheduling

OptiX API

CUDA C shaders from user programs

GPU Execution via CUDA

Life of a ray



1 Ray Generation

2 Intersection

3 Shading

Life of a ray

1 Pinhole Camera

```
RT_PROGRAM void pinhole_camera()
{
  float2 d = make_float2(launch_index) / make_float2(launch_dim) * 2.f - 1.f;
  float3 ray_origin = eye;
  float3 ray_direction = normalize(d.x*U + d.y*V + W);

  optix::Ray ray = optix::make_Ray(ray_origin, ray_direction,
    radiance_ray_type, scene_epsilon, RT_DEFAULT_MAX);

  PerRayData_radiance prd;
  rtTrace(top_object, ray, prd);
  output_buffer[launch_index] = make_color( prd.result );
}
```

2 Ray-Sphere Intersection

```
RT_PROGRAM void intersect_sphere()
{
  float3 O = ray.origin - center;
  float3 D = ray.direction;
  float b = dot(O, D);
  float c = dot(O, O)-radius*radius;
  float disc = b*b-c;
  if(disc > 0.0f){
    float sdisc = sqrtf(disc);
    float root1 = (-b - sdisc);
    bool check_second = true;
    if( rtPotentialIntersection( root1 ) ) {
      shading_normal = geometric_normal = (O + root1*D)/radius;
      if(rtReportIntersection(0))
        check_second = false;
    }
    if(check_second) {
      float root2 = (-b + sdisc);
      if( rtPotentialIntersection( root2 ) ) {
        shading_normal = geometric_normal = (O + root2*D)/radius;
        rtReportIntersection(0);
      }
    }
  }
```

3 Lambertian Shading

```
RT_PROGRAM void closest_hit_radiance3()
{
  float3 world_geo_normal   = normalize( rtTransformNormal( RT_OBJECT_TO_WORLD, geometric_normal ) );
  float3 world_shade_normal = normalize( rtTransformNormal( RT_OBJECT_TO_WORLD, shading_normal ) );
  float3 ffnormal     = faceforward( world_shade_normal, -ray.direction, world_geo_normal );
  float3 color = Ka * ambient_light_color;

  float3 hit_point = ray.origin + t_hit * ray.direction;

  for(int i = 0; i < lights.size(); ++i) {
    BasicLight light = lights[i];
    float3 L = normalize(light.pos - hit_point);
    float nDl = dot( ffnormal, L);

    if( nDl > 0.0f ){
      // cast shadow ray
      PerRayData_shadow shadow_prd;
      shadow_prd.attenuation = make_float3(1.0f);
      float Ldist = length(light.pos - hit_point);
      optix::Ray shadow_ray( hit_point, L, shadow_ray_type, scene_epsilon, Ldist );
      rtTrace(top_shadower, shadow_ray, shadow_prd);
      float3 light_attenuation = shadow_prd.attenuation;

      if( fmaxf(light_attenuation) > 0.0f ){
        float3 Lc = light.color * light_attenuation;
        color += Kd * nDl * Lc;

        float3 H = normalize(L - ray.direction);
        float nDh = dot( ffnormal, H );
        if(nDh > 0)
          color += Ks * Lc * pow(nDh, phong_exp);
      }

    }
  }
  prd_radiance.result = color;
}
```

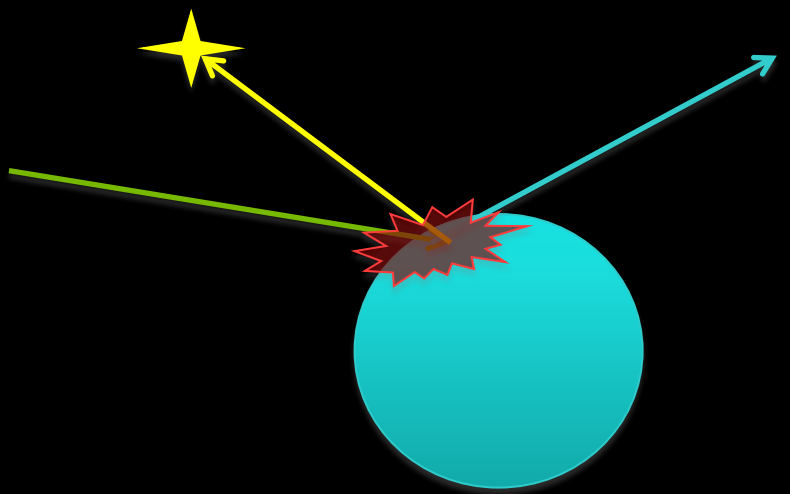
Program objects (shaders)

```
RT_PROGRAM void pinhole_camera()
{
  float2 d = make_float2(launch_index) /
    make_float2(launch_dim) * 2.f - 1.f;
  float3 ray_origin = eye;
  float3 ray_direction = normalize(d.x*U + d.y*V + W);

  optix::Ray ray = optix::make_Ray(ray_origin,
ray_direction,
    radiance_ray_type, scene_epsilon, RT_DEFAULT_MAX);

  PerRayData_radiance prd;
  rtTrace(top_object, ray, prd);
  output_buffer[launch_index] = make_color( prd.result );
}
```

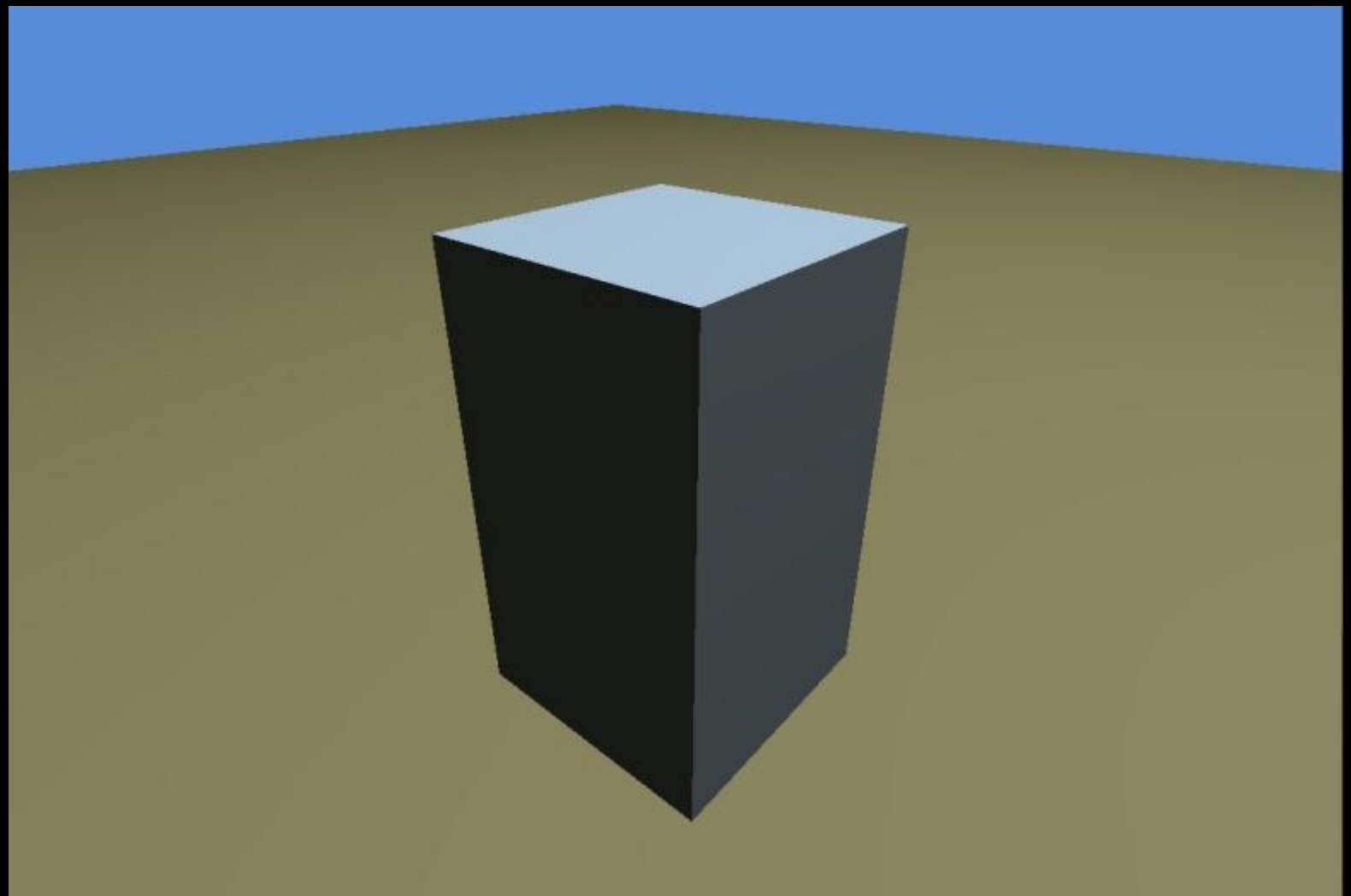
- Input “language” is based on CUDA C/C++
  - No new language to learn
  - Powerful language features available immediately
  - Can also take raw PTX as input
- Data associated with ray is programmable
- Caveat: still need to use it responsibly to get performance

Closest hit program (traditional “shader”)

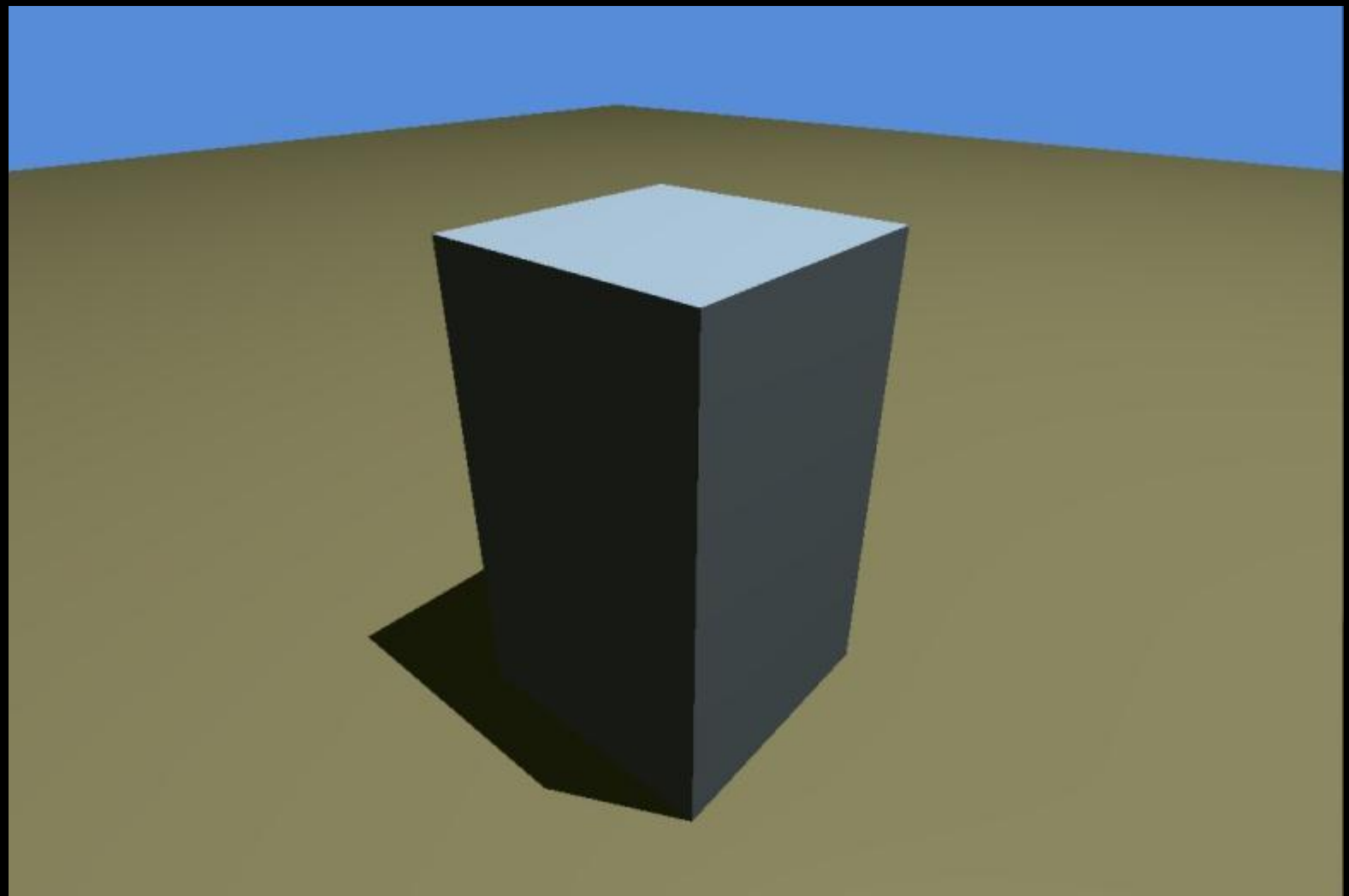
- Defines what happens when a ray hits an object
- Executed for nearest intersection (closest hit) along a ray
- Automatically performs deferred shading
- Can recursively shoot more rays
  - Shadows
  - Reflections
  - Ambient occlusion
  - Path tracing
- Most common



Lambertian shader

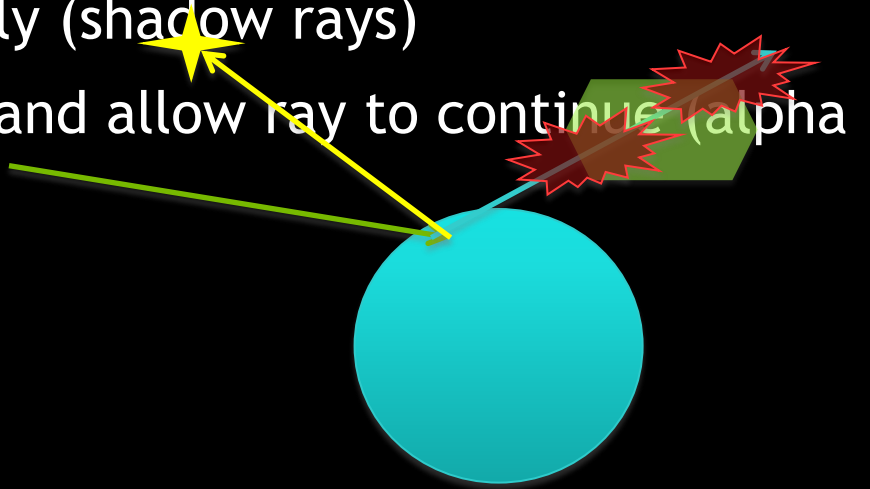


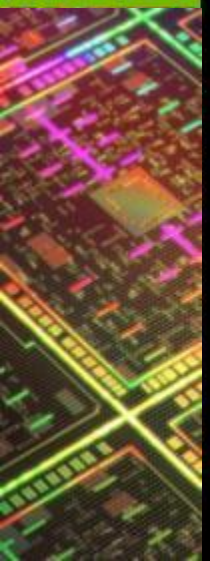
Adding shadows



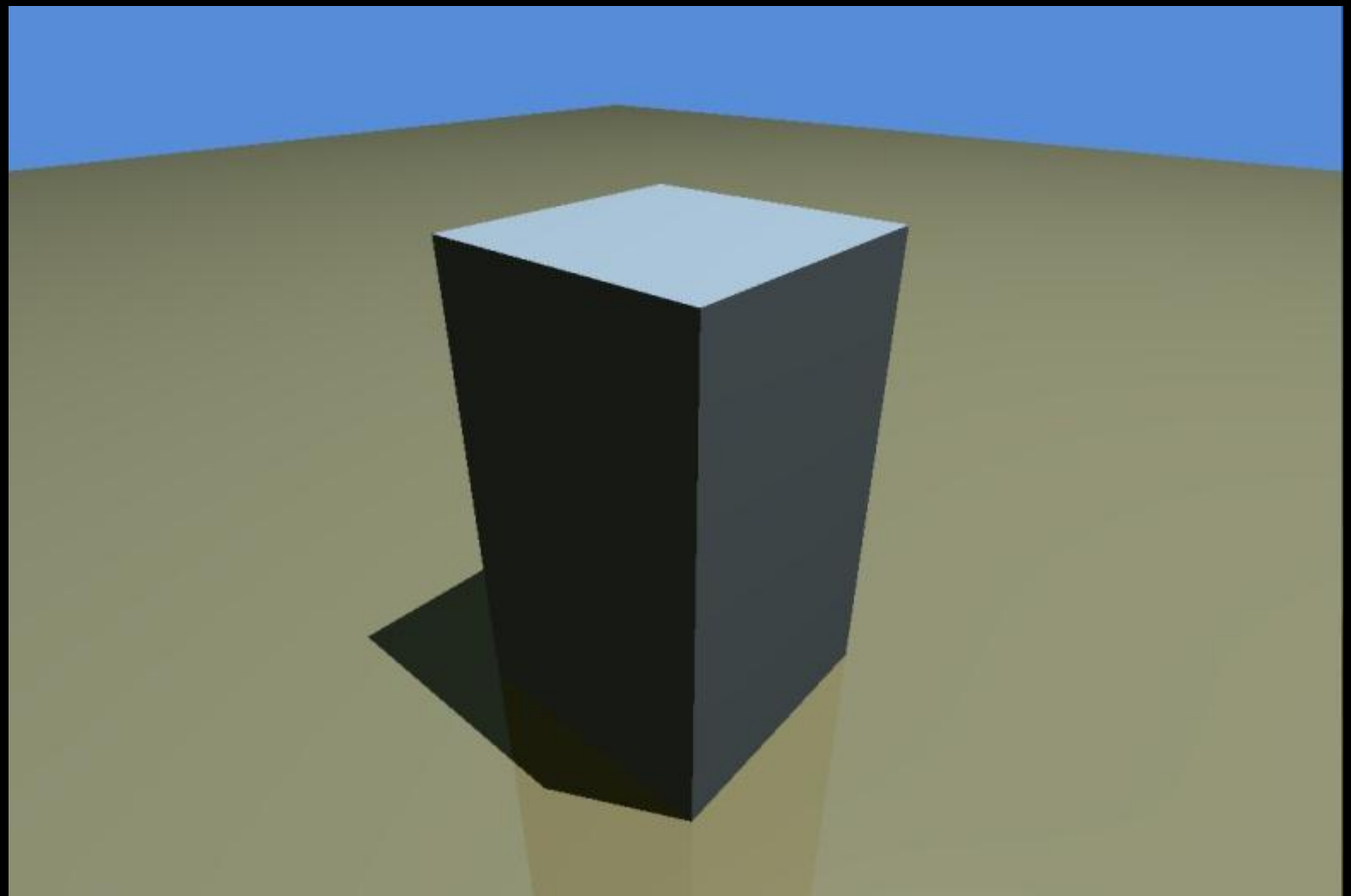
Any hit program

- Defines what happens when a ray attempts to hit an object
- Executed for all intersections along a ray
- Can optionally:
  - Stop the ray immediately (shadow rays)
  - Ignore the intersection and allow ray to continue (alpha transparency)

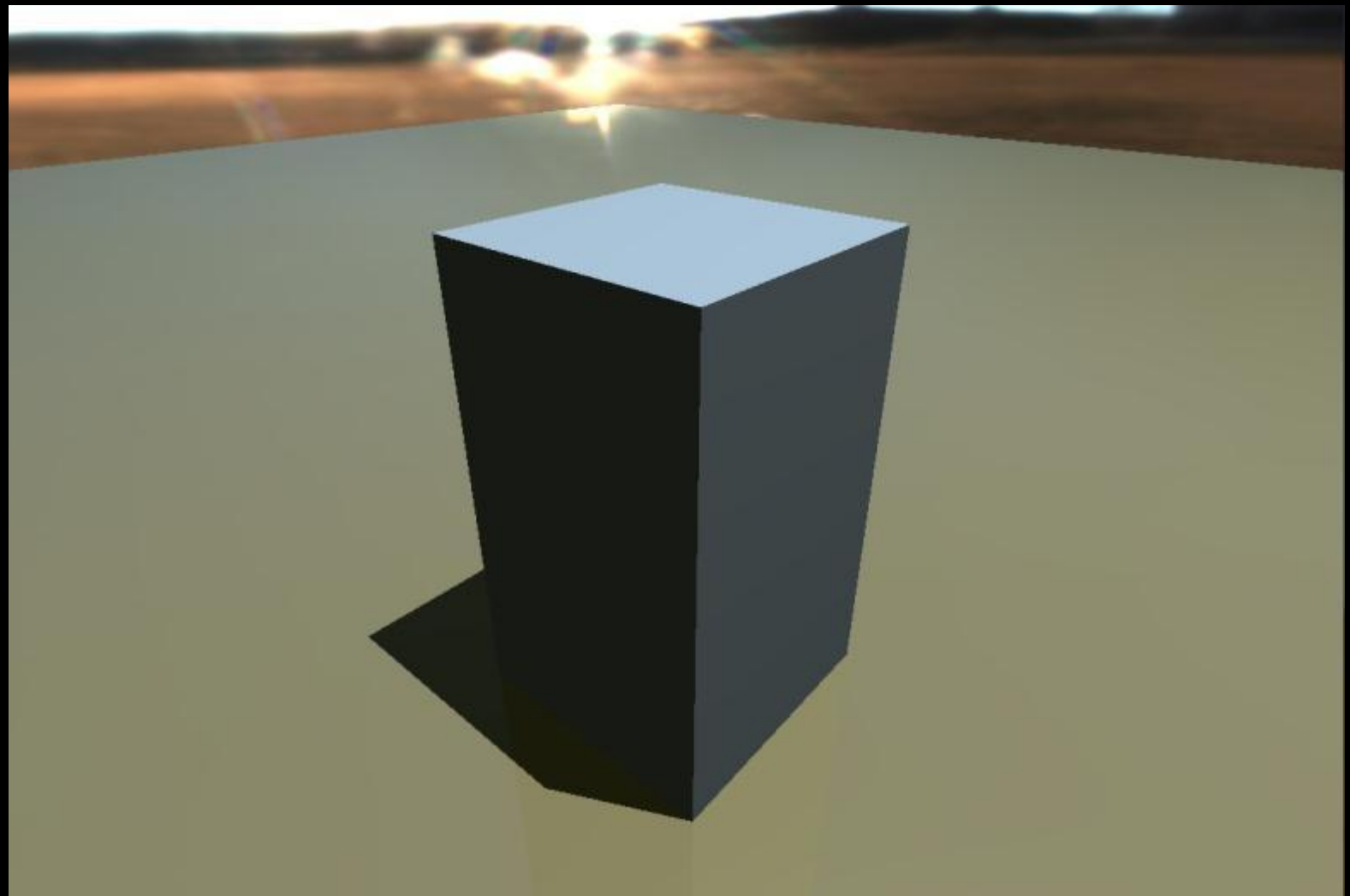




Adding reflections

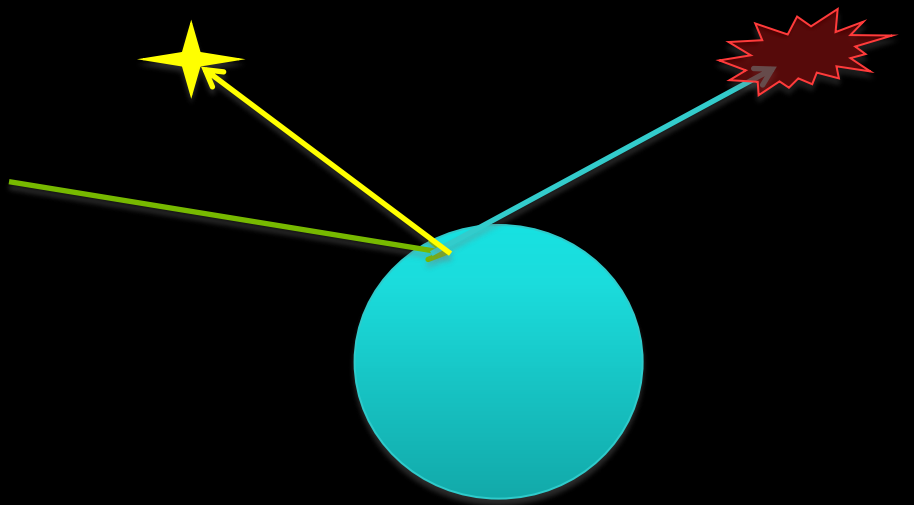


Environment map

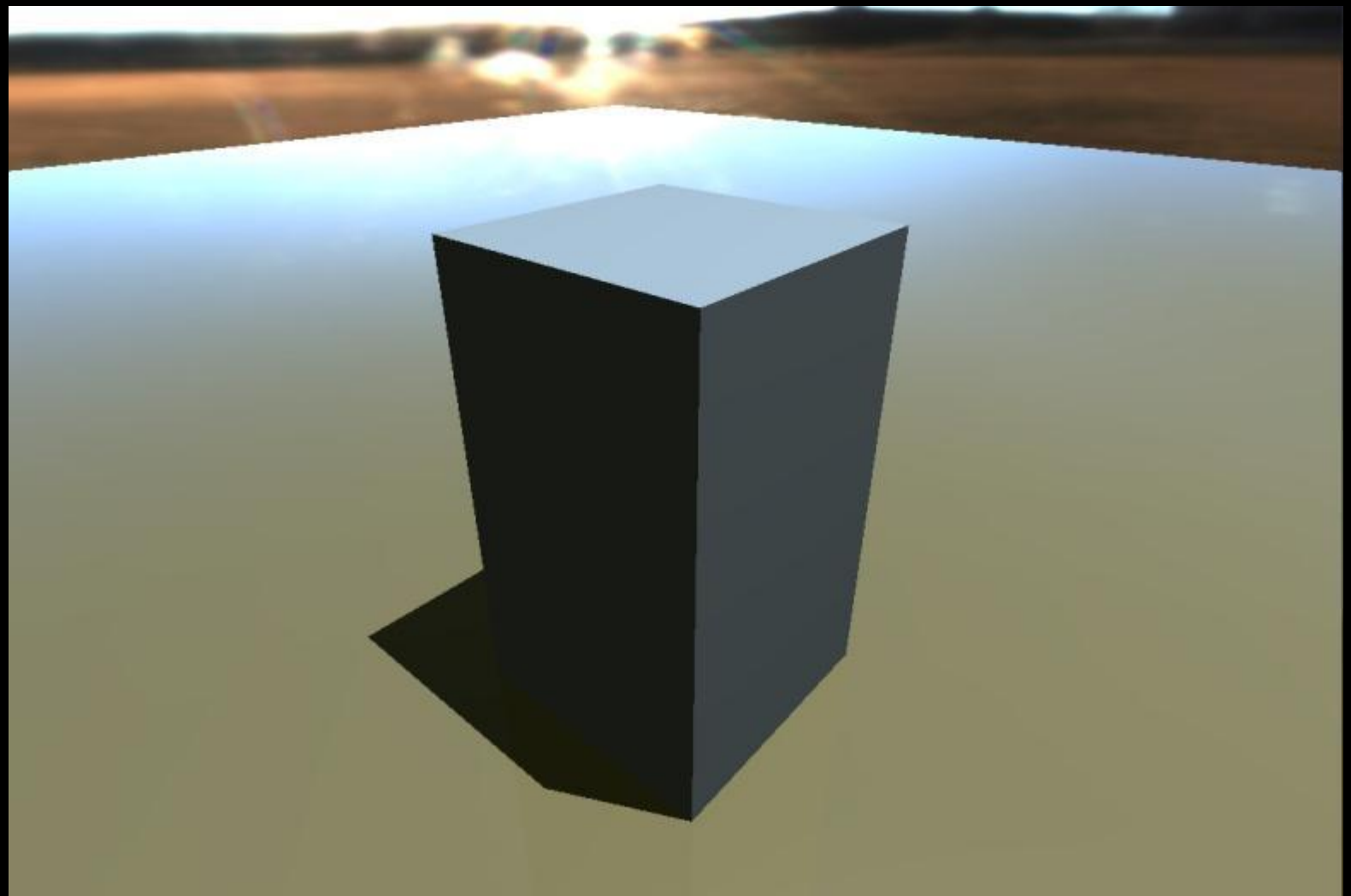


Miss program

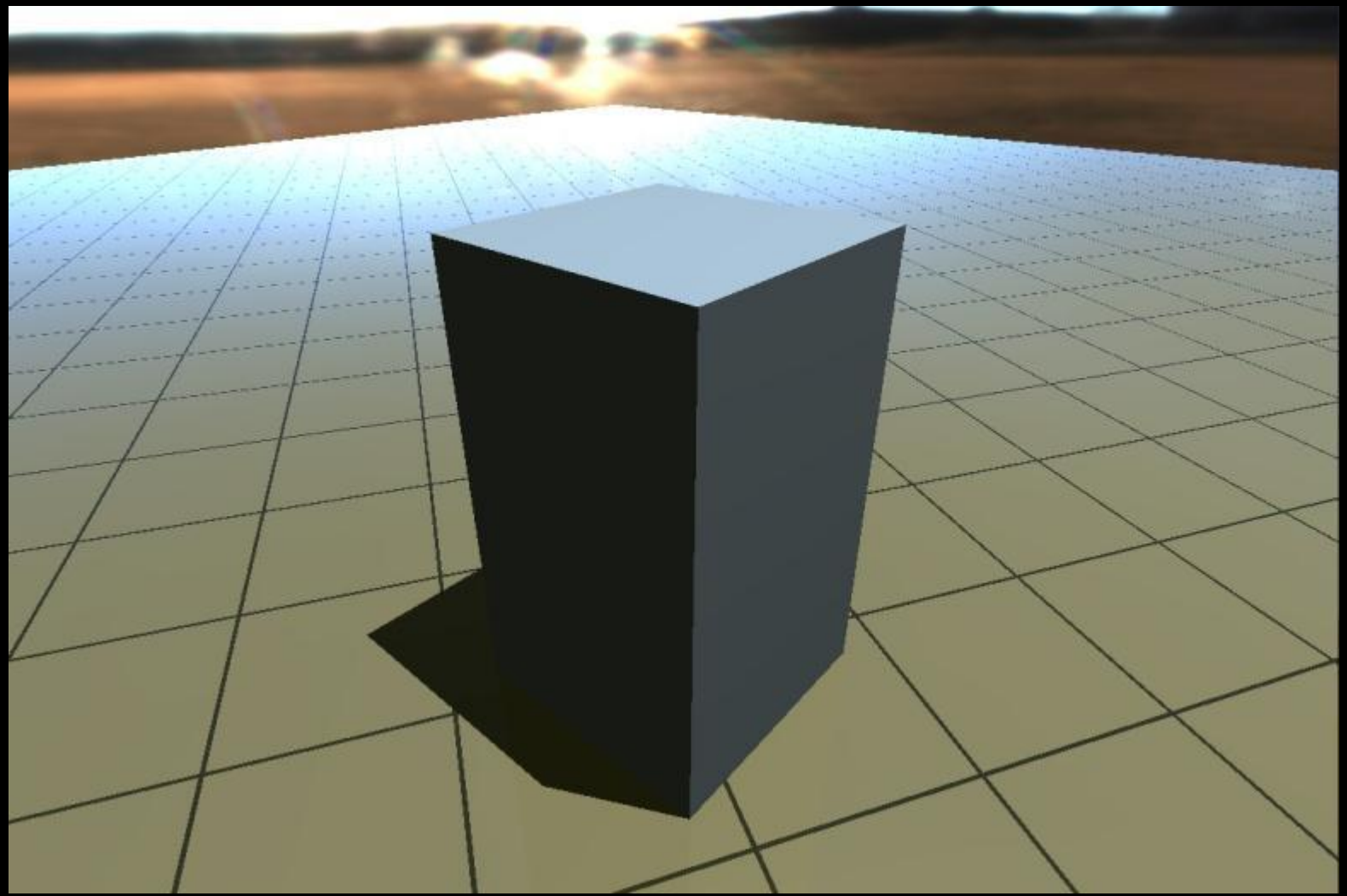
- Defines what happens when a ray misses all objects
- Accesses ray payload
- Usually – background color



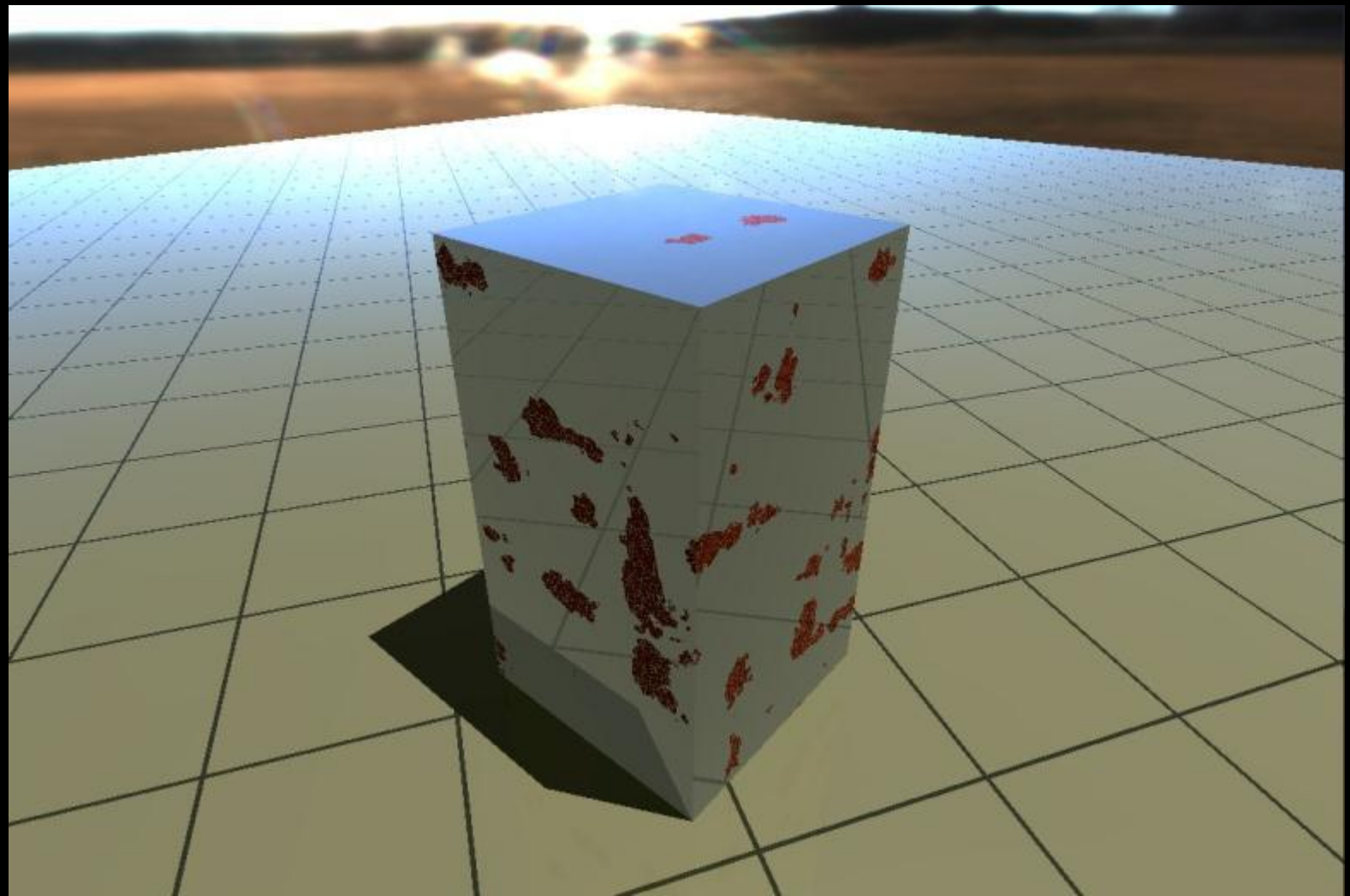
Schlick approximation



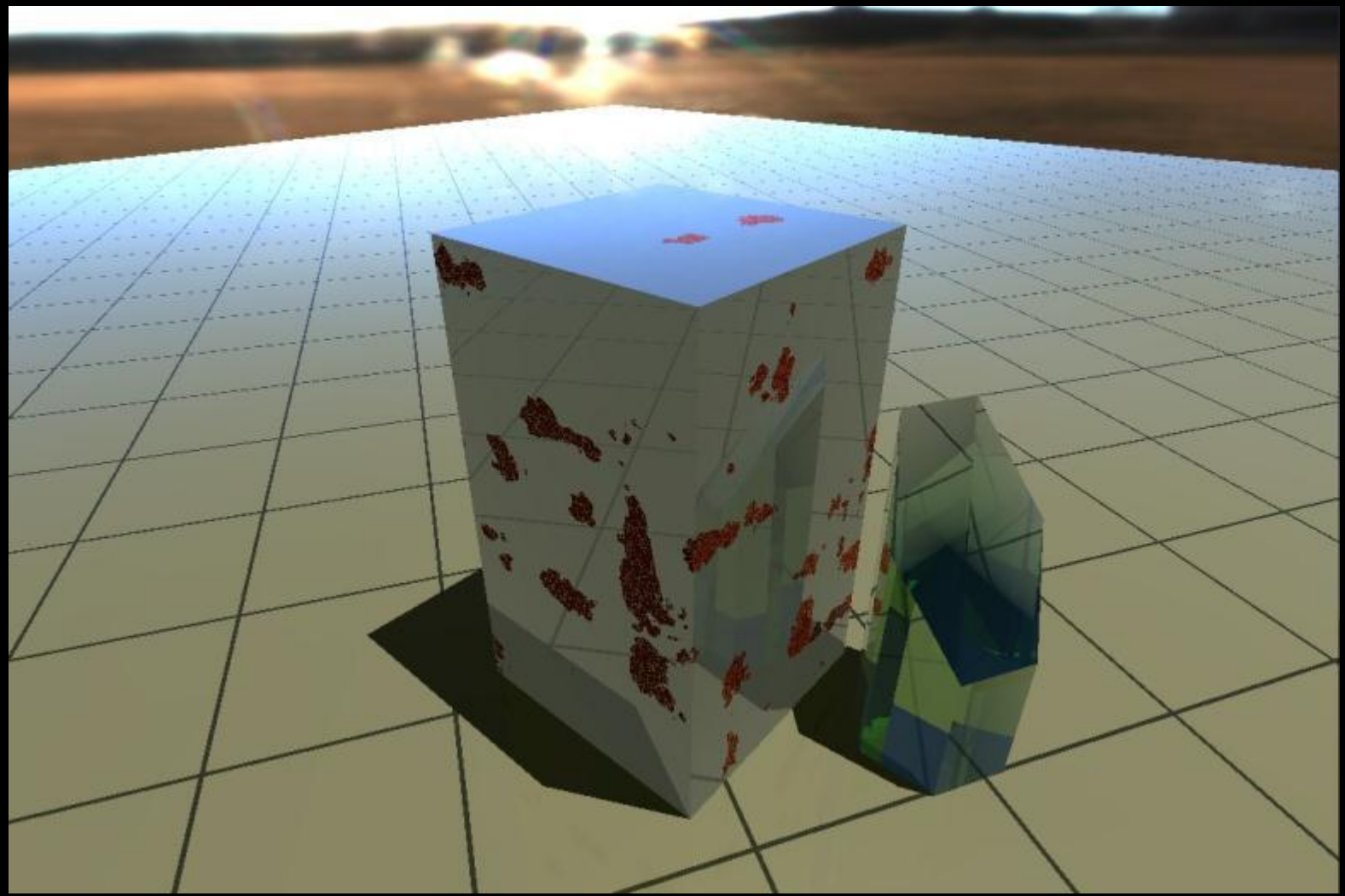
Tiled floor



Rusty metal

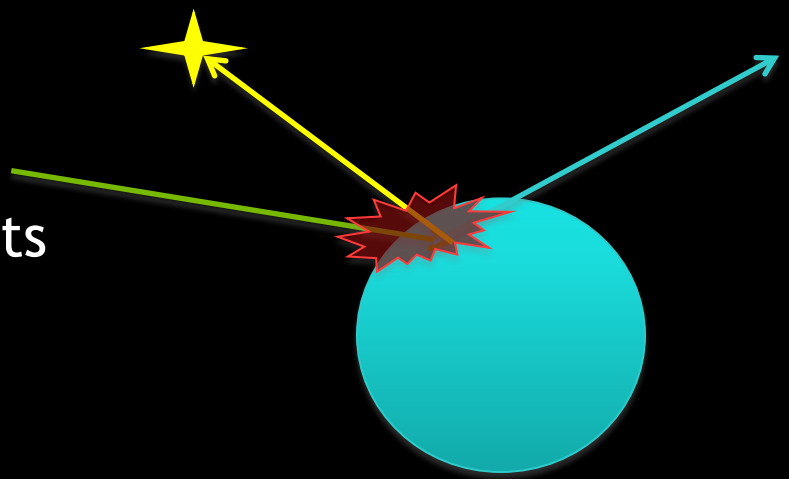


Adding primitives

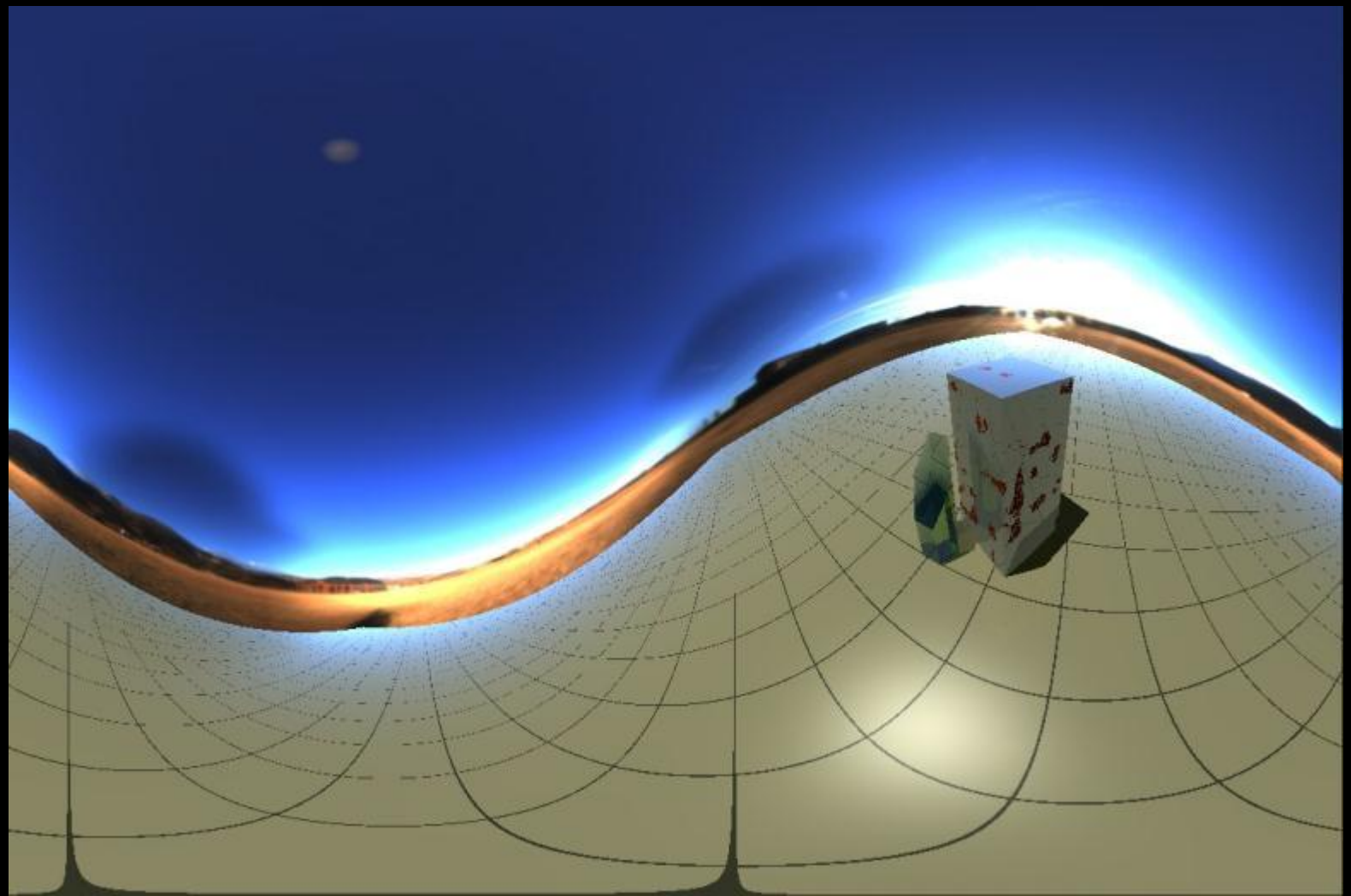


Intersection program

- Determines if/where ray hits an object
- Sets attributes (normal, texture coordinates)
  - Used by closest hit shader for shading
- Selects which material to use
- Used for
  - Programmable surfaces
  - Allowing arbitrary triangle buffer formats
  - Etc.

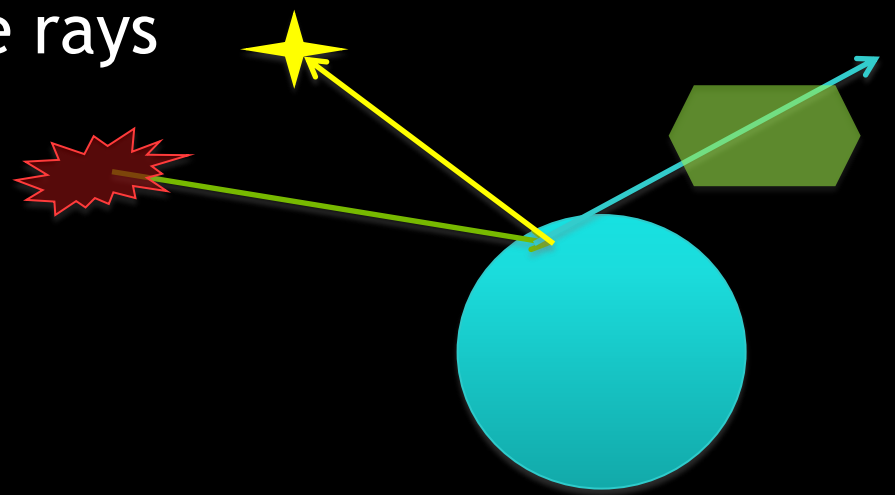


Environment map camera



Ray generation program

- Starts the ray tracing process
- Used for:
  - Camera model
  - Output buffer writes
- Can trace multiple rays
- Or no rays

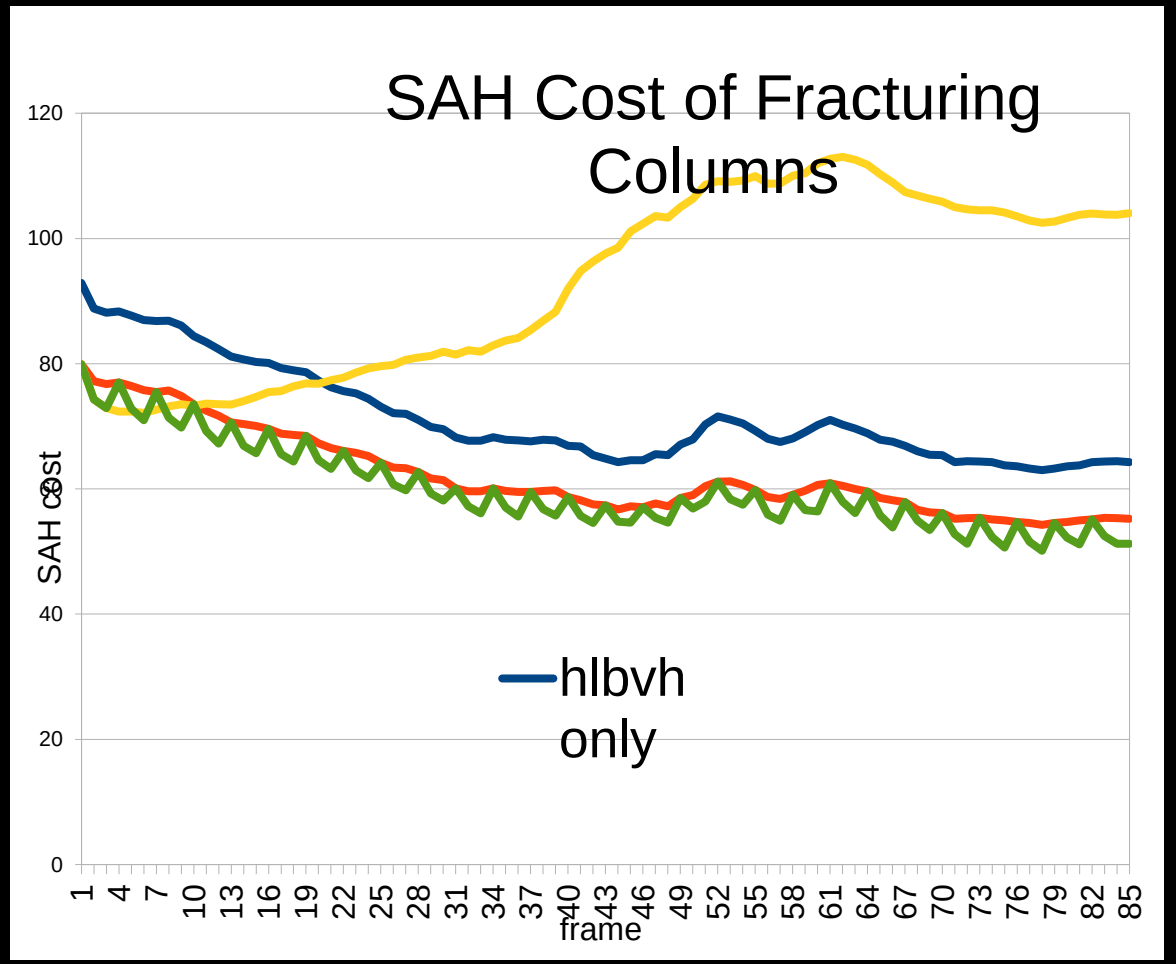


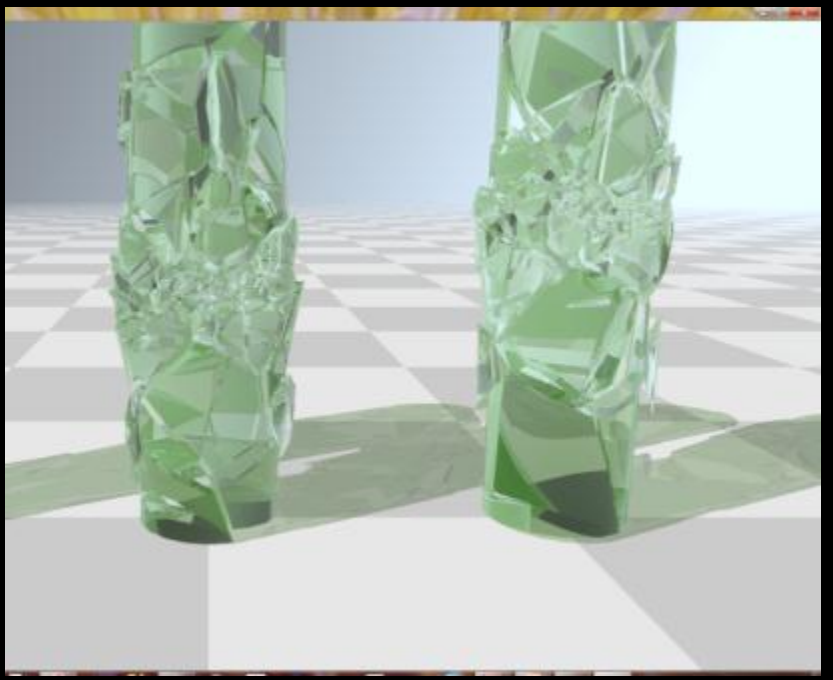
OptiX - What's Next?

Acceleration Structures++

- “Sbvh” is up to 8X faster
- “Lbvh” is extremely fast and works on very large datasets
- BVH Refinement optimizes the quality of a BVH
  - Smoother scene editing
  - Smoother animation

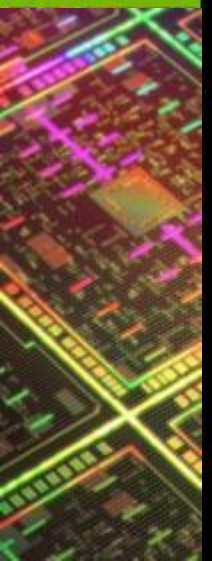
BVH Refinement

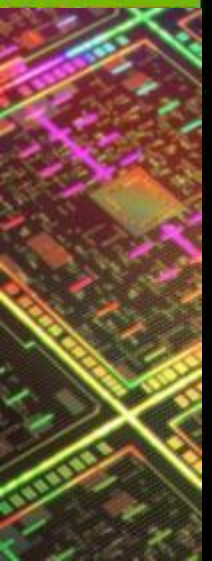




CUDA-OptiX Interoperability

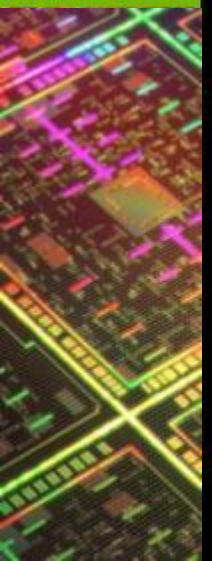
- Share a CUDA context between OptiX and CUDA runtime
- Share buffers on one device without memory copies
- Copy buffers from device to device peer-to-peer
  - Avoid round-trip through host





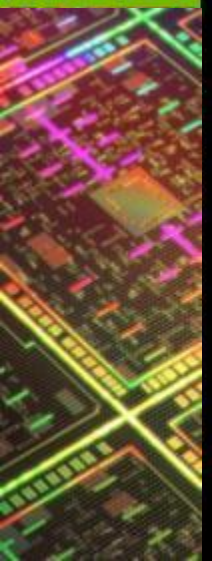
Shade Tree Support

- User Functions
- Bindless Texture



Paging

- Use cases:
  - Mildly oversubscribed:
    - (513MB dataset, 512MB card)
  - Largely oversubscribed:
    - (20GB dataset, 6GB card)
- Approach: Use OptiX Compiler to implement virtual memory system in OptiX kernel



Software Texture

- Texture hardware is massive speedup
- Compiler pass replaces TEX instructions
- Sometimes a speedup (float1, NEAREST)
- Usually a slowdown
- Choose which textures to fall back to SW
- Best 127 textures stay in HW

Thanks for Attending! OptiX SDK

- Free to acquire and use: Windows, Linux, Mac
- http://developer.nvidia.com