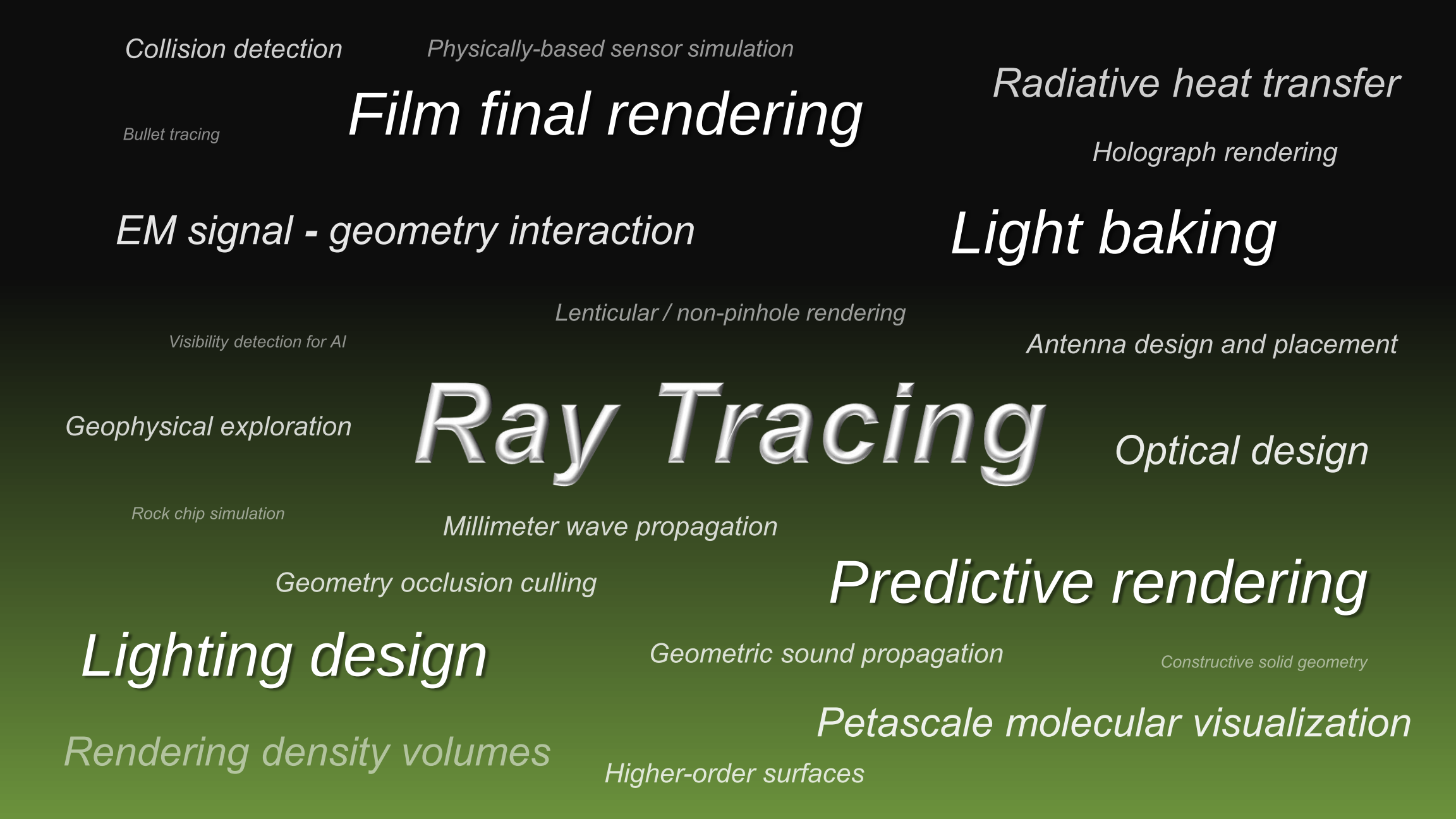
GPU TECHNOLOGY CONFERENCE

NVIDIA

ADVANCED OPTIX

David McAllister, James Bigler, Brandon Lloyd


Collision detection
Physically-based sensor simulation
Radiative heat transfer
Film final rendering
Bullet tracing
Holograph rendering
EM signal - geometry interaction
Light baking
Lenticular / non-pinhole rendering
Visibility detection for AI
Antenna design and placement
Geophysical exploration
Ray Tracing
Optical design
Rock chip simulation
Millimeter wave propagation
Geometry occlusion culling
Predictive rendering
Lighting design
Geometric sound propagation
Constructive solid geometry
Petascale molecular visualization
Rendering density volumes
Higher-order surfaces

OPTIX 3.5 WHAT'S NEW

- **OptiX Prime**
  for blazingly fast traversal & intersection **(±300m rays/sec/GPU)**
  - You give the triangles and rays, you get the intersections, in 5 lines of code
- **TRBVH Builder**
  builds **+100X faster**, runs about as fast as SBVH (previous fastest)
  - Part of OptiX Prime, also in OptiX core
  - Does require more memory (to be improved later this year)
  - Bug fixes coming in 3.5.2
- **GK110B Optimizations** (Tesla K40, Quadro K6000, GeForce GTX Titan Black)
  +**25%** more performance

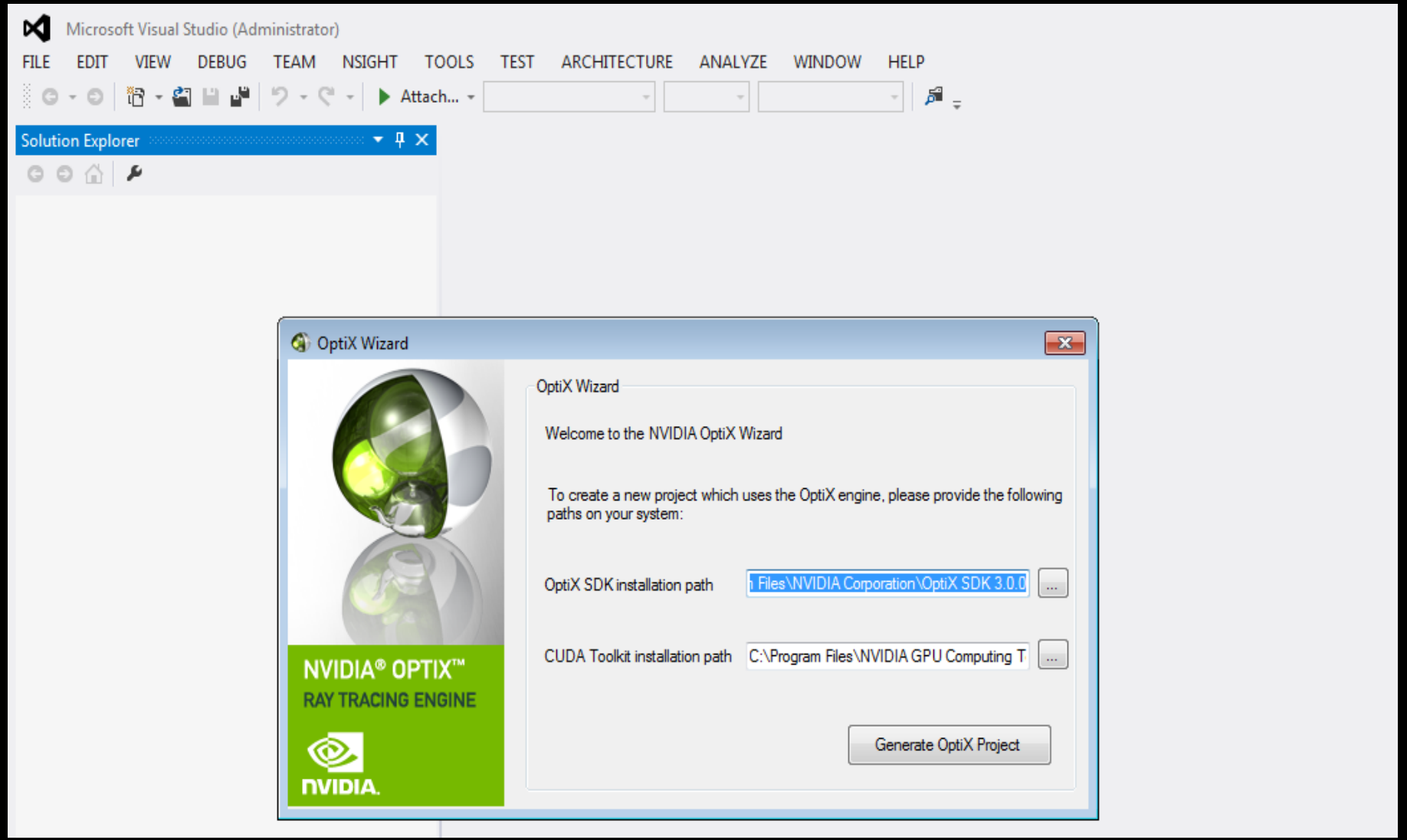
OPTIX IN BUNGIE

- Vertex Light Baking
  - Available publicly. *Just ask us.*
  - Kavan, Bargteil, Sloan, “Least Squares Vertex Baking”, EGSR 2011
- Compared to textures…
  - Less memory & bandwidth
  - No *u,v* parameterization
  - Good for low-frequency effects
- Over 250,000 baking jobs done

OPTIX 3.5 WHAT'S NEW

- **Compiles** 3-7X faster
  - Still, try to avoid recompiles
- **Hosted** Documentation
  - http://docs.nvidia.com
- **Platform** support
  - Visual Studio 2012 support
  - CUDA 5.5 support
- **Bindless** Buffers & Buffers of Buffers
  - More flexibility with callable programs (e.g., shade trees)

VISUAL STUDIO OPTIX WIZARD



BUFFER IDS (V3.5)

- Previously only attachable to Variables
- With a Buffer API object, request the ID (`rtBufferGetId`)
- Use ID
  - In a buffer
    - rtBuffer<rtBufferId<float3,1>, 1> buffers;
    - float3 val = buffers[i][j];
  - Passed as arguments*
    - float work(rtBufferId<float3,1> data);
  - Stored in structs*
    - struct MyData { rtBufferId<float3,1>; int stuff; };

* Can thwart some optimizations

OPTIX IN IRAY INTERACTIVE

- Iray Interactive mode
  - Running on OptiX
  - Approximated GI
  - Interactive frame rates

- Iray Photoreal mode
  - Running on CUDA

CALLABLE PROGRAM IDS (V3.6)

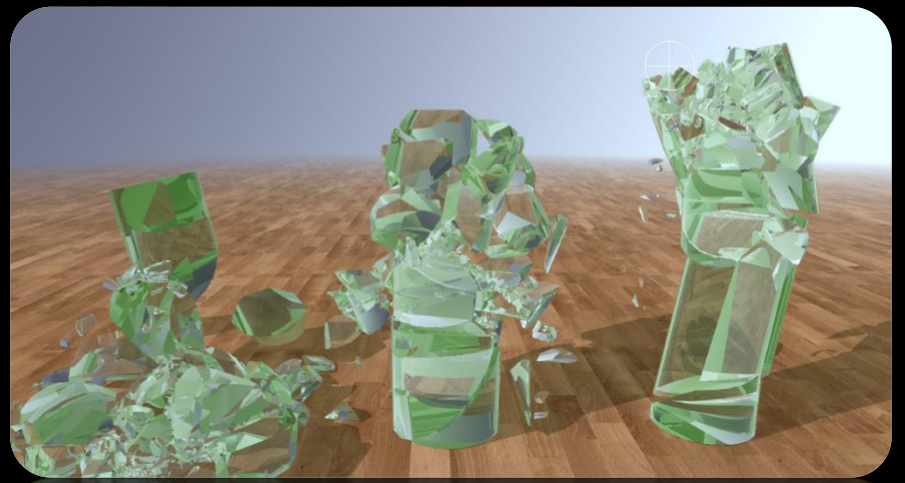
- Think of them as a functor (function pointer with data)
  - PTX (RTprogram)
  - Variables attached to RTprogram API object
- With a RTprogram API object, request the ID (`rtProgramGetId`)
- Use ID
  - In a buffer
    - `rtBuffer<rtCallableProgramId<int,int>, 1> programs;`
    - `int val = programs[i](4);`
  - As a variable
    - `typedef rtCallableProgramId<int,int> program_t;`
    - `rtDeclareVariable(program_t, program,,);`
    - `int val = program(3);`
  - Passed as arguments*
  - Stored in structs*

* Can thwart some optimizations

OPTIX IN PIXAR'S RAY TRACING PREVIEWER

OPTIX 3.5 SDK

- Available now: Windows, Linux, Mac
- http://developer.nvidia.com



OPTIX 3.5 REGISTERED DEVELOPER PROGRAM

Improvements to OptiX development as it enters its 4th year:

- OptiX Registered Developer Program (RDP)
  - Fill out our survey. Reviewed within 1 working day and you have access
  - Will become an actual RDP on the NVIDIA Developer Zone by summer

- OptiX Commercial Developer Program
  - For commercial applications needing to redistribute OptiX 3.5 binaries
  - Includes a Commercial OptiX version with commercial enhancements, and a higher level of support
  - Cost is primarily information, except for high-earning products which pay an annual fee for our highest level of support

OPTIX PRIME

- Specialized for ray tracing (no shading)
- Replaces rtuTraversal (rtuTraversal is still supported)
- Improved performance
  - Uses latest algorithms from NVIDIA Research
    - ray tracing kernels [Aila and Laine 2009; Aila et al. 2012]
    - Treelet Reordering BVH (TRBVH) [Karras 2013]
  - Can use CUDA buffers as input/output
  - Support for asynchronous computation
- Designed with an eye towards future features

API OVERVIEW

- C API with C++ wrappers
- API Objects
  - Context
  - Buffer Descriptor
  - Model
  - Query

CONTEXT

- Context tracks other API objects and encapsulates the ray tracing backend
- Creating a context

```
RTPresult
rtpContextCreate(RTPcontexttype type, RTPcontext* context)
```

- Context types

```
RTP_CONTEXT_TYPE_CPU
RTP_CONTEXT_TYPE_CUDA
```

- Default for CUDA backend uses all available GPUs
  - Selects “Primary GPU” and makes it the current device
  - Primary GPU builds acceleration structure

CONTEXT

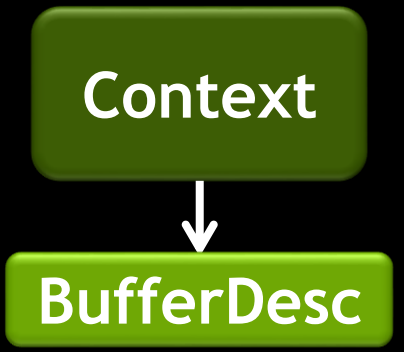
- Selecting devices:

```
rtpContextSetCudaDeviceNumbers(
   RTPcontext  context,
   int         deviceCount,
   const int*  deviceNumbers )
```

  - First device is used as the primary GPU
- Destroying the context
  - destroys objects created by the context
  - synchronizes the CPU and GPU

BUFFER DESCRIPTOR

- Buffers are allocated by the application
- Buffer descriptors encapsulate information about the buffers




```
rtpBufferDescCreate(
    RTPcontext      context,
    RTPbufferformat format,
    RTPbuffertype   type,
    void*           buffer,
    RTPbufferdesc*  desc )
```

- Specify region of buffer to use (in elements)

```
rtpBufferDescSetRange( RTPbufferdesc desc, int begin, int end )
```

BUFFER DESCRIPTOR

- Variable stride supported for vertex format

  `rtpBufferDescSetStride`

  – Allows for vertex attributes

BUFFER DESCRIPTOR

- Formats

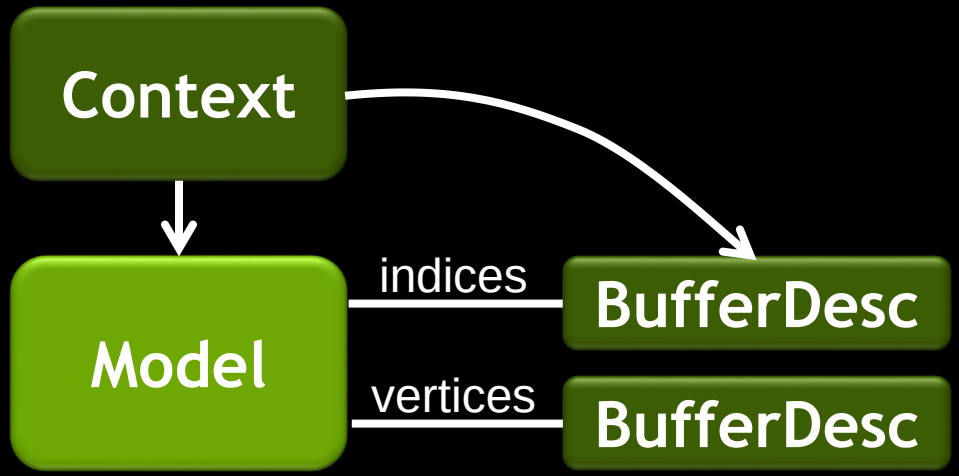
  ```
  RTP_BUFFER_FORMAT_INDICES_INT3
  RTP_BUFFER_FORMAT_VERTEX_FLOAT3,
  RTP_BUFFER_FORMAT_RAY_ORIGIN_DIRECTION,
  RTP_BUFFER_FORMAT_RAY_ORIGIN_TMIN_DIRECTION_TMAX,
  RTP_BUFFER_FORMAT_HIT_T_TRIID_U_V
  RTP_BUFFER_FORMAT_HIT_T_TRIID
  …
  ```

- Types

  ```
  RTP_BUFFER_TYPE_HOST
  RTP_BUFFER_TYPE_CUDA_LINEAR
  ```

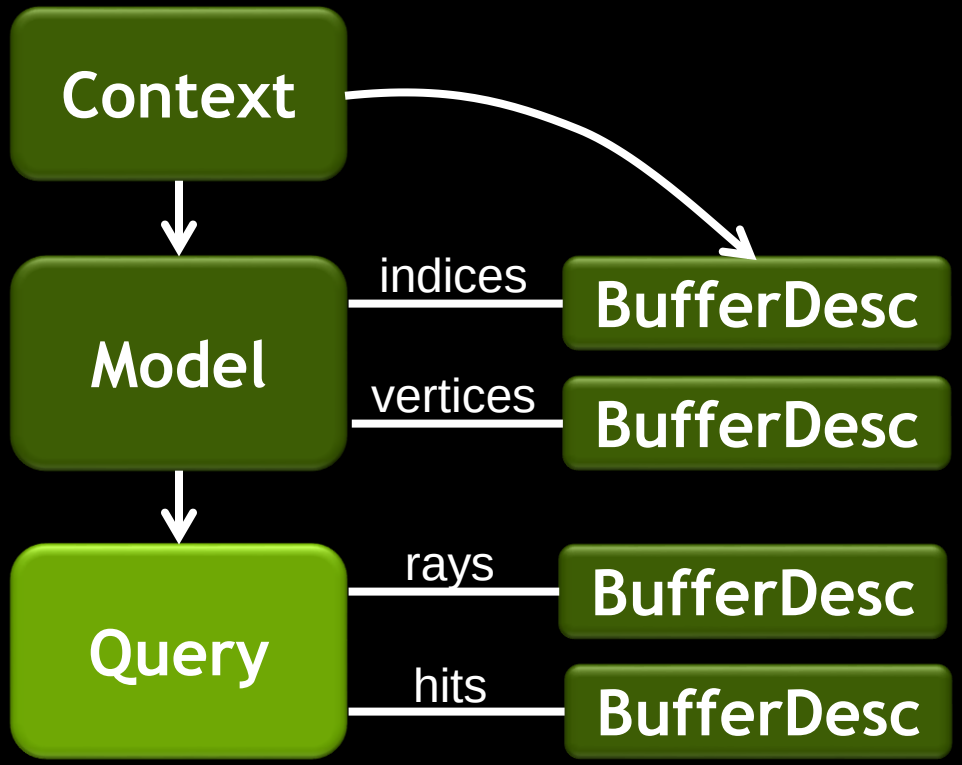
MODEL

- A model is a set of triangles combined with an acceleration data structure
  - `rtpModelCreate`
  - `rtpModelSetTriangles`
  - `rtpModelUpdate`
- Asynchronous update
  - `rtpModelFinish`
  - `rtpModelGetFinished`

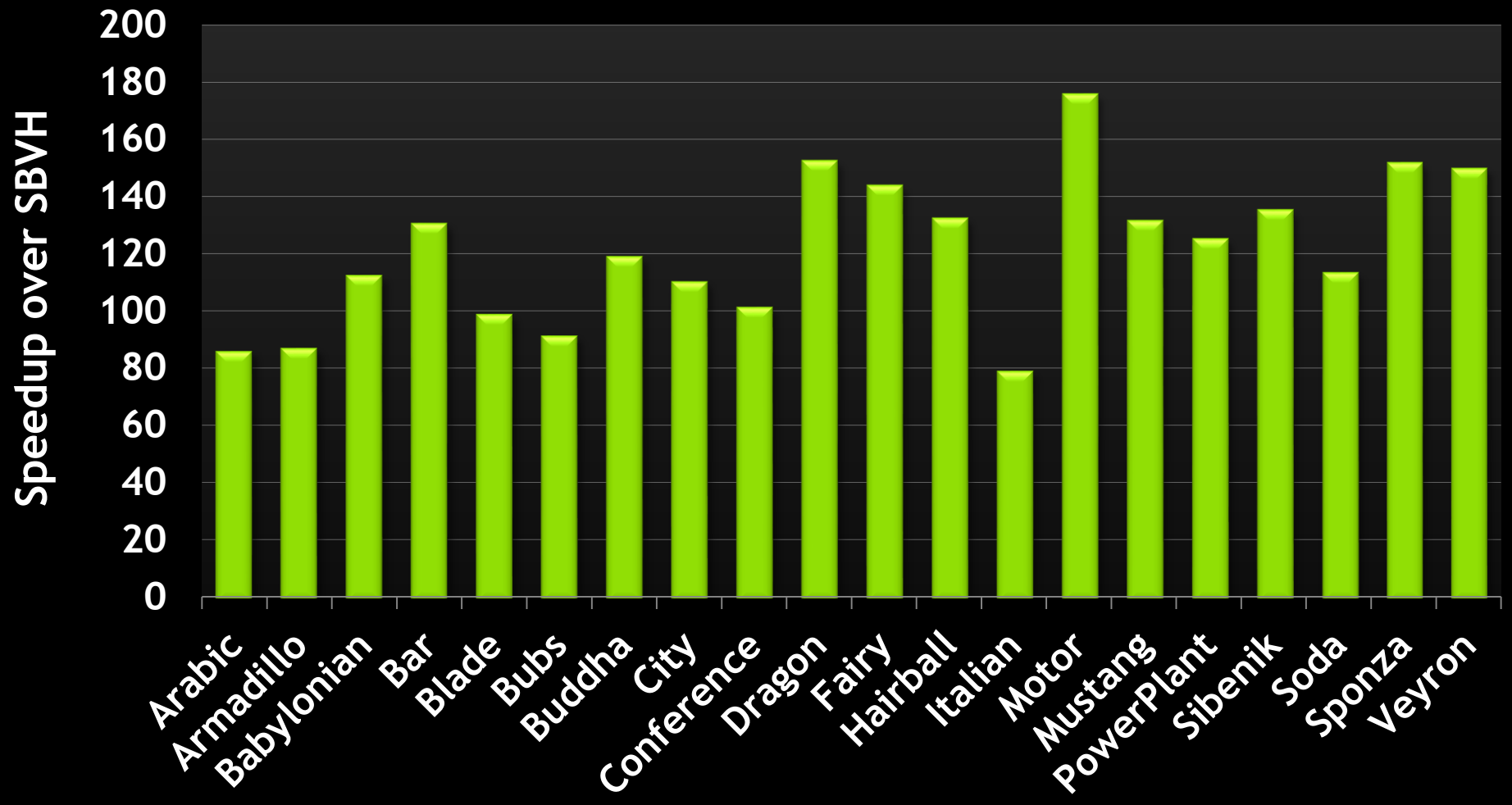


QUERY

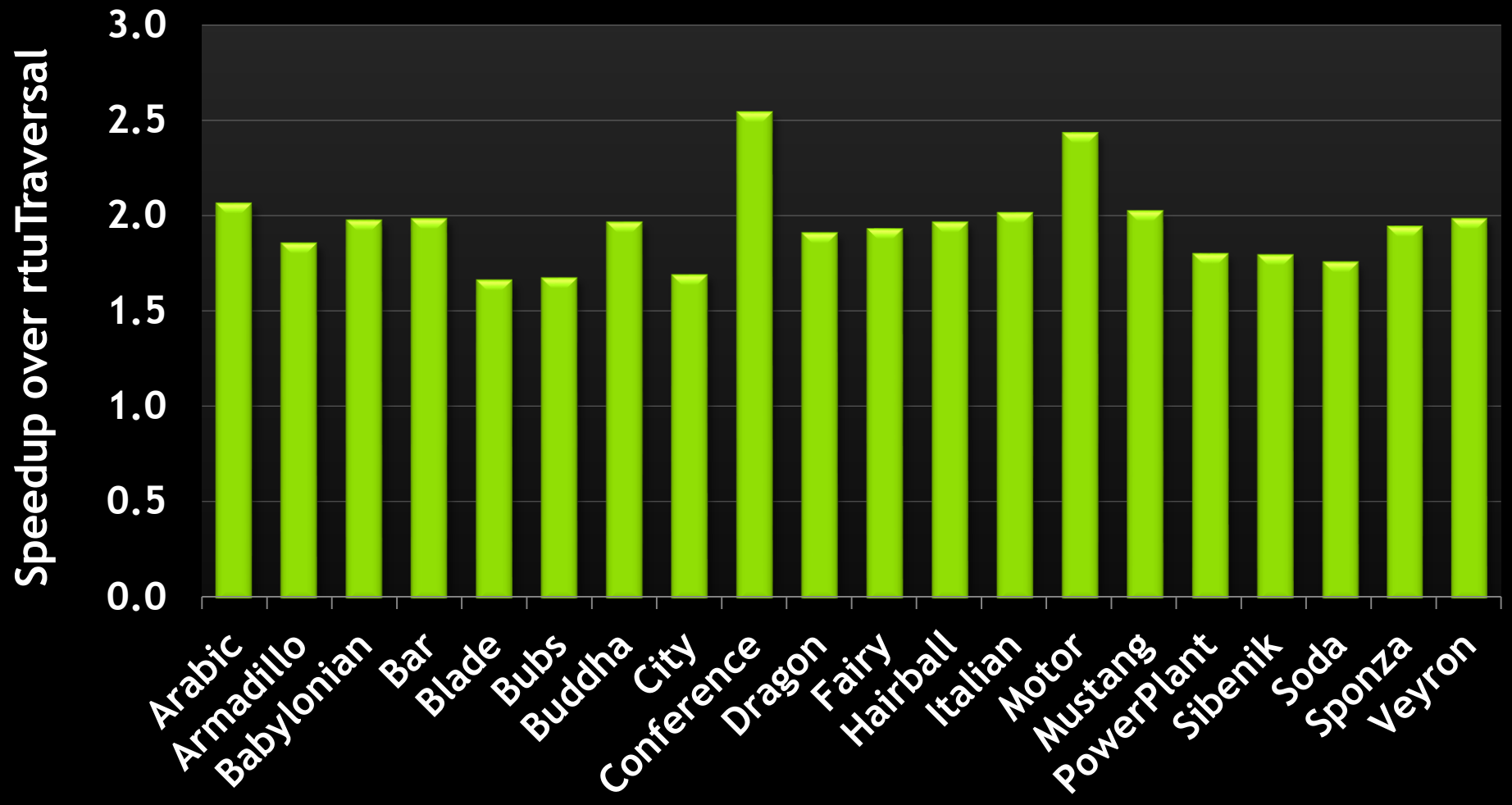
- Queries perform the ray tracing on a model
  - `rtpQueryCreate`
  - `rtpQuerySetRays`
  - `rtpQuerySetHits`
  - `rtpQueryExecute`
- Query types
  - `RTP_QUERY_TYPE_ANY`
  - `RTP_QUERY_TYPE_CLOSEST`
- Asynchronous query
  - `rtpQueryFinish`
  - `rtpQueryGetFinished`



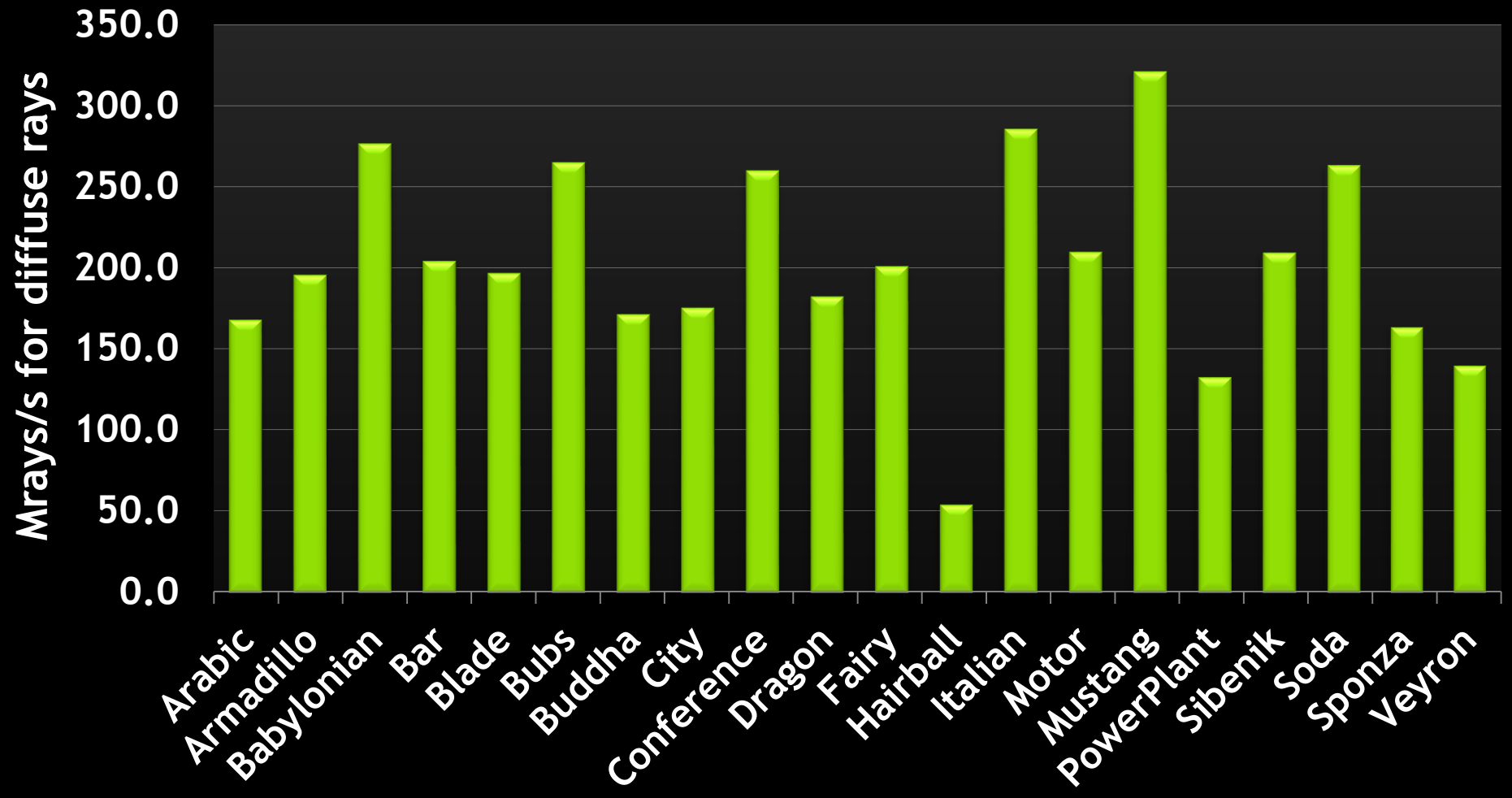
BUILD PERFORMANCE



RAY TRACING PERFORMANCE



RAYTRACING PERFORMANCE



OPTIX PRIME ROADMAP

- Features we want to implement
  - Animation support (refit/refine)
  - Instancing
  - Large-model optimizations

OPTIMIZING PRIME

- Use asynch API to keep the CPU busy
  - important for multi-threading (synchronous calls lock out other threads)
- Avoid host ↔ device copies
  - buffers on host for RTP_CONTEXT_TYPE_HOST
  - buffers on device for RTP_CONTEXT_TYPE_DEVICE
- Lock host buffers for faster host ↔ device copies
  - lockable memory is limited
  - use multi-buffering to stage through lockable memory (see simplePrimeppMultiBuffering sample)

OPTIMIZING PRIME – MULTI-GPU

- Multi-device context requires inter-device copies
  - Faster to put buffers in host memory (with current CUDA)
- Max performance: generate rays and process hits on the device
- Create context per device
  - Build model on one device and copy to others with rtpModelCopy
  - OR build the same model on all devices
- (See simplePrimeppMultiGPU sample)

OPTIX VS. OPTIX PRIME

OptiX

- Single ray programming model
- Includes shading
  - Native recursion
- Programmable primitives
- CPU backend unavailable
- Expressive API
- Virtualizes GPU resources

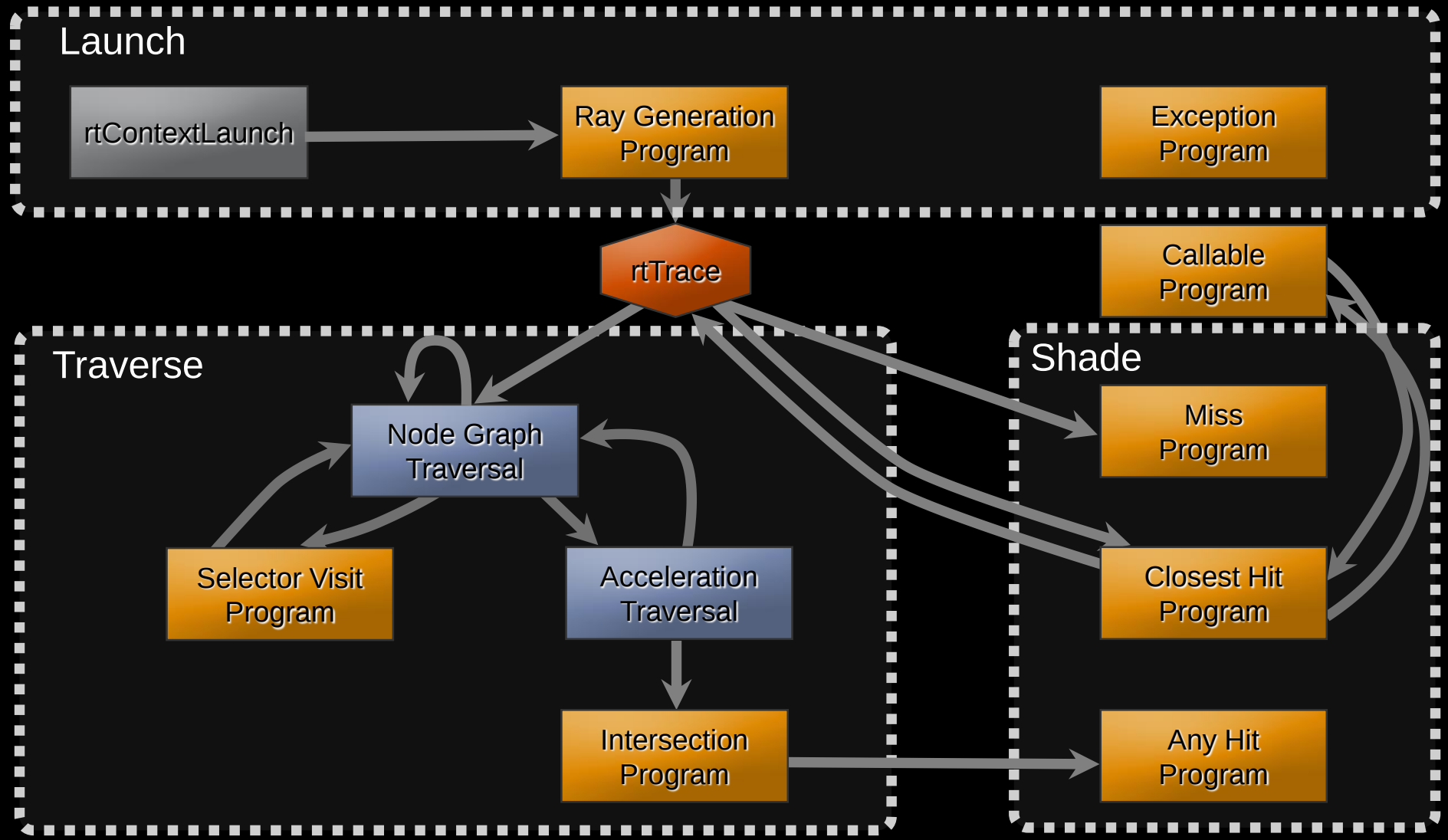
OptiX Prime

- Works best on large waves
- No shading support
  - Use CUDA
- Triangles only
- Performant CPU backend
- Constrained API, slow to evolve
- To the metal
- Performance++

OPTIMIZING OPTIX DEVICE CODE

- Maximize rays/second
  - Avoid gratuitous divergence
  - Avoid memory traffic
  - Improve single thread performance
- Minimize rays needed for given result
  - Improved ray tracing algorithms
    - MIS, QMC, MCMC, BDPT, MLT

OPTIX EXECUTION MODEL



MINIMIZE CONTINUATION STATE

- OptiX rewrites rtTrace, rtReportIntersection, etc. as:
  - Find all registers needed after rtFunction (continuation state)
  - Save continuation state to local memory
  - Execute function body
  - Restore continuation state from local memory
- Minimizing continuation state can have a large impact
  - Avoids local variable saves and restores
  - This also improves execution coherence
  - How?
    - Push rtTrace calls to bottom of function
    - Push computation to top of function

MINIMIZE CONTINUATION STATE

```
float3 light_pos, light_color, light_scale;
sampleLight(light_pos, light_color, light_scale); // Fill in vals

optix::Ray ray = ...;                    // create ray given light_pos

PerRayData shadow_prd;
rtTrace(top_object, ray, shadow_prd);          // Trace shadow ray

return light_color*light_pos*shadow_prd.attenuation;
```

light_pos and light_color saved to local stack

MINIMIZE CONTINUATION STATE

```
float3 light_pos, light_color, light_scale;
sampleLight(light_pos, light_color, light_scale); // Fill in vals
float3 scaled_light_color = light_color*light_scale;
optix::Ray ray = ...;                    // create ray given light_pos

PerRayData shadow_prd;
rtTrace(top_object, ray, shadow_prd);          // Trace shadow ray

return scaled_light_color*shadow_prd.attenuation;
```

MINIMIZE CONTINUATION STATE

```
RT_PROGRAM void closestHit() {
    float3 N  = rtTransformNormal( normal );
    float3 P  = ray.origin + t_hit * ray.direction;
    float3 wo = -ray.di

    // Compute direct
    float3 on_light  =
    float  dist = length(on_light-P)
    float3 wi = (on_light - P) / length;
    float3 bsdf = bsdfVal(wi, N, wo, bsdf_params);
    bool is_occluded = traceShadowRay(P, wi, dist);
    if( !is_occluded ) prd.result = light_col * bsdf;

    // Fill in values for next path trace iteration
    bsdfSample( wo, N, bsdf_params,
                prd.next_wi, prd.next_bsdf_weight );
}
```

Pulled above trace to reduce stack state

```
RT_PROGRAM void closestHit() {
    float3 N  = rtTransformNormal( normal );
    float3 P  = ray.origin + t_hit * ray.direction;
    float3 wo = -ray.direction;

    // Fill in values for next path trace iteration
    bsdfSample( wo, N, bsdf_params,
                prd.next_wi, prd.next_bsdf_weight );

    // Compute direct lighting
    float3 on_light = lightSample();
    float dist = length(on_light - P)
    float3 wi = (on_light - P) / length;
    float3 bsdf = bsdfVal(wi, N, wo, bsdf_params);
    bool is_occluded = traceShadowRay(P, wi, dist);
    if( !is_occluded ) prd.result = light_col * bsdf;
}
```

DESIGN GARAGE: ITERATIVE PATH TRACER

- Closest hit programs do:
  - Direct lighting (next event estimation with shadow query ray)
  - Compute next ray (sample BSDF for reflected/refracted ray info)
  - Return direct light and next ray info to ray gen program
- Ray gen program iterates

DESIGNGARAGE: ITERATIVE PATH TRACER

```
RT_PROGRAM void rayGeneration(){
    float3 ray_dir = cameraGetRayDir();
    float3 result = tracePathRay( camera.pos, ray_dir, 1 );
    output_buffer[ launch_index ] = result;
}
RT_PROGRAM void closestHit() {
    // Calculate BSDF sample for next path ray
    float3 ray_direction, ray_weight;
    sampleBSDF( wo, N, ray_direction, ray_weight );

    // Recurse
    float3 indirect_light = tracePathRay(P, ray_direction,
                                         ray_weight);
    // Perform direct lighting
    ...
    prd.result = indirect_light + direct_light;
}
```

DESIGNGARAGE: ITERATIVE PATH TRACER

```
RT_PROGRAM void rayGeneration() {
    PerRayData prd;
    prd.ray_dir = cameraGetRayDir();
    prd.ray_origin = camera.position;
    float3 weight = make_float3( 1.0f );
    float3 result = make_float3( 0.0f );

    for( i = 0; i < MAX_DEPTH; ++i ) {
        traceRay( prd.ray_origin,
                  prd.ray_dir, prd );
        result += prd.direct*weight;
        weight *= prd.ray_weight;
    }
    output_buffer[ launch_index ] = result;
}
```

```
RT_PROGRAM void closestHit() {
    // Calculate BSDF sample for next path ray
    float3 ray_direction, ray_weight;
    sampleBSDF( wo, N, ray_direction, ray_weight );

    // Return sampled ray info and let ray_gen
    // iterate
    prd.ray_dir = ray_direction;
    prd.ray_origin = P;
    prd.ray_weight = ray_weight;
    // Perform direct lighting
    ...
    prd.direct = direct_light;
}
```

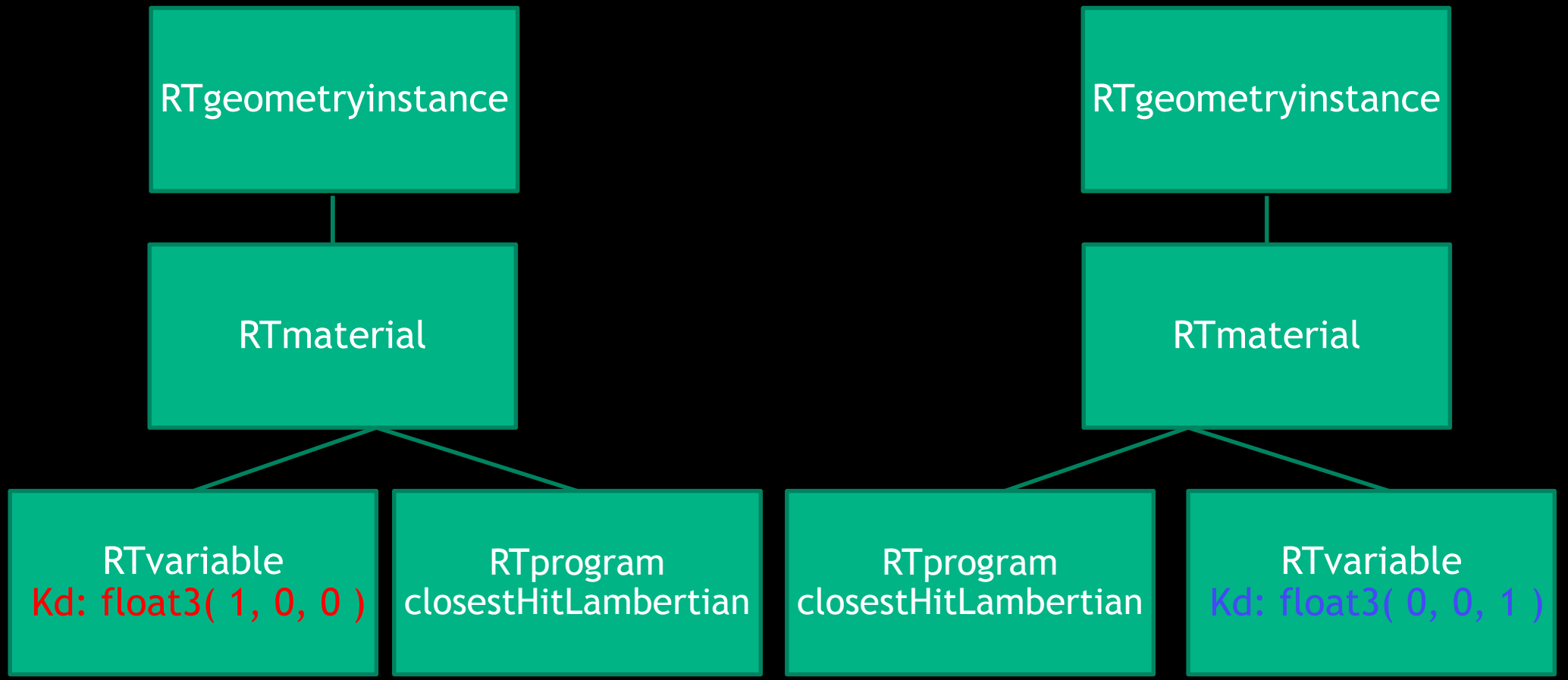
A FEW QUICK SUGGESTIONS

- Accessing stack allocated arrays through pointers uses local memory not registers
  - Change float v[3] to float v0, v1, v2
  - Avoid accessing variables via pointer
- Careful Arithmatic
  - nvcc --use_fast_math
  - Do not unintentionally use double precision math
    - 1.0 != 1.0f
    - cos() != cosf()
  - Search for ".f64" in your PTX files
- Take advantage of any hit programs and rtTerminateRay for fast boolean ray queries
- Use interop to share data (CUDA, OpenGL, DirectX)

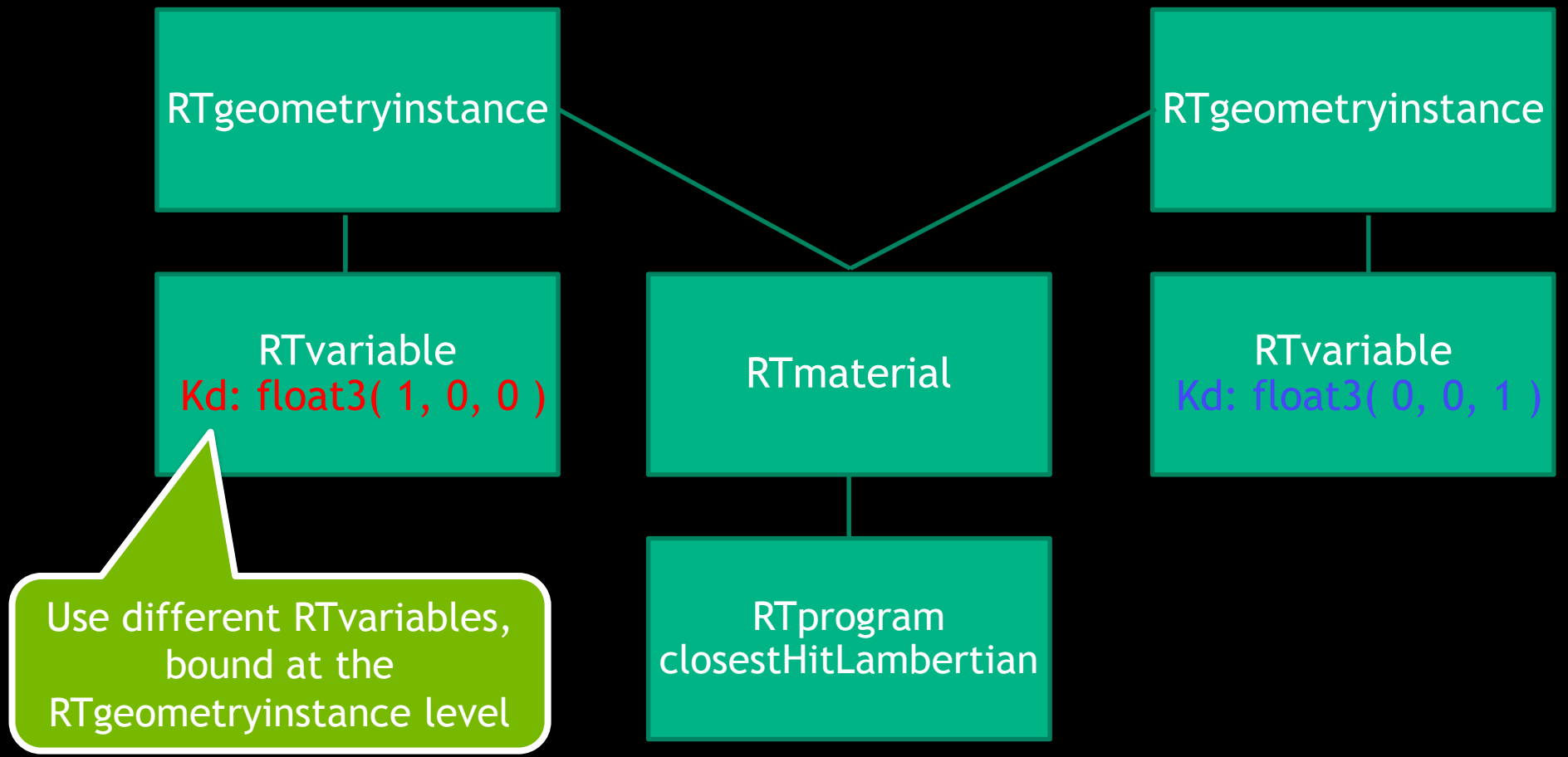
SHALLOW NODE HIERARCHIES

- Flatten node hierarchy
  - Collapse nested RTtransforms
  - Pre-transform vertices
  - Use RTselectors judiciously
- Combine multiple meshes into single mesh
- ☺ A single BVH over all geometry
- ☹ Per-mesh BVHes
- Reuse RTprograms
  - Use variables or control flow to reuse programs
    - ☹ singleSidedDiffuse closest hit and doubleSidedDiffuse closest hit
    - ☺ diffuse closest hit and RTvariable do_double_sided
  - Use in moderation: über-shaders cause longer compilation

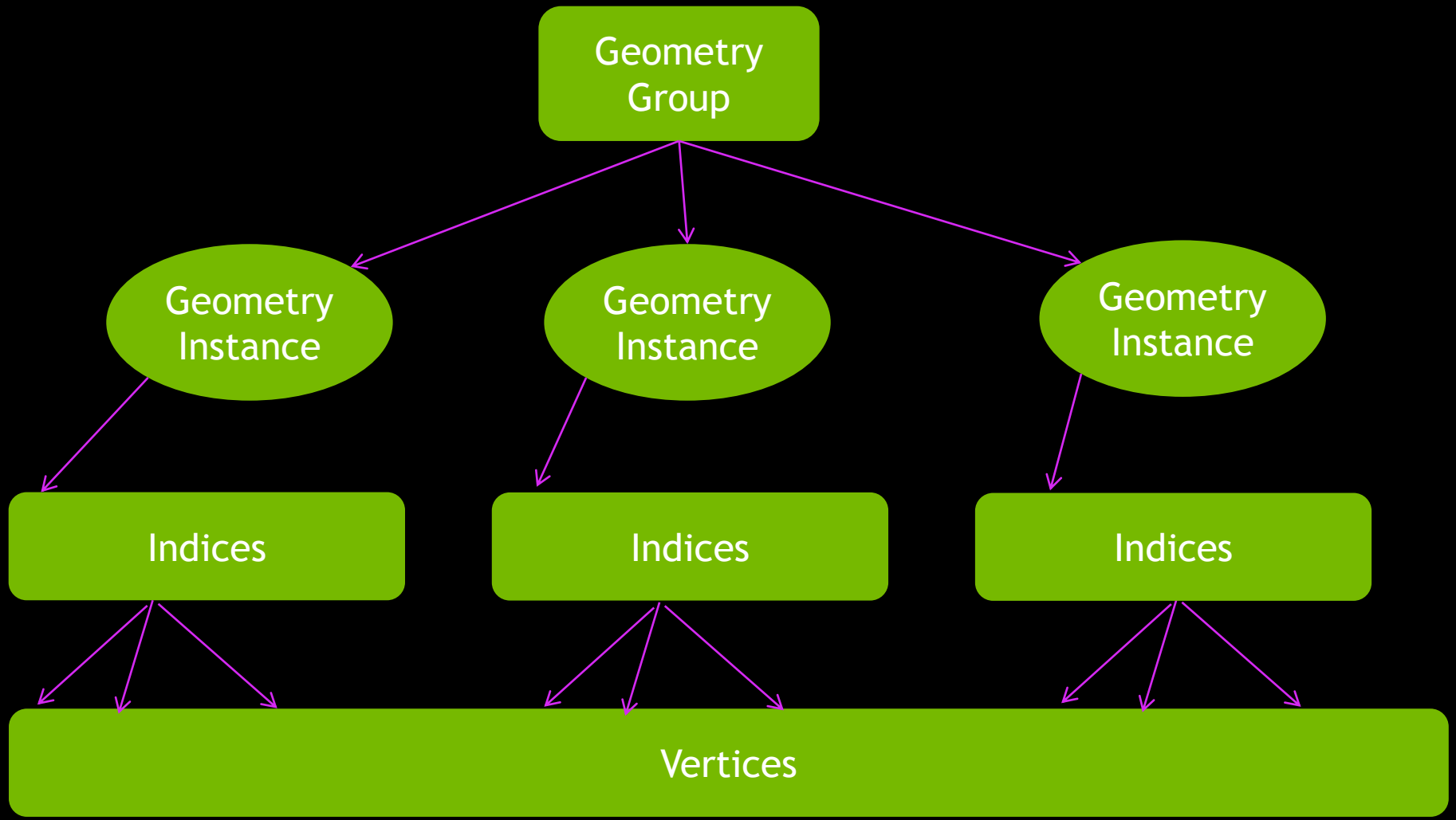
SHARE GRAPH NODES



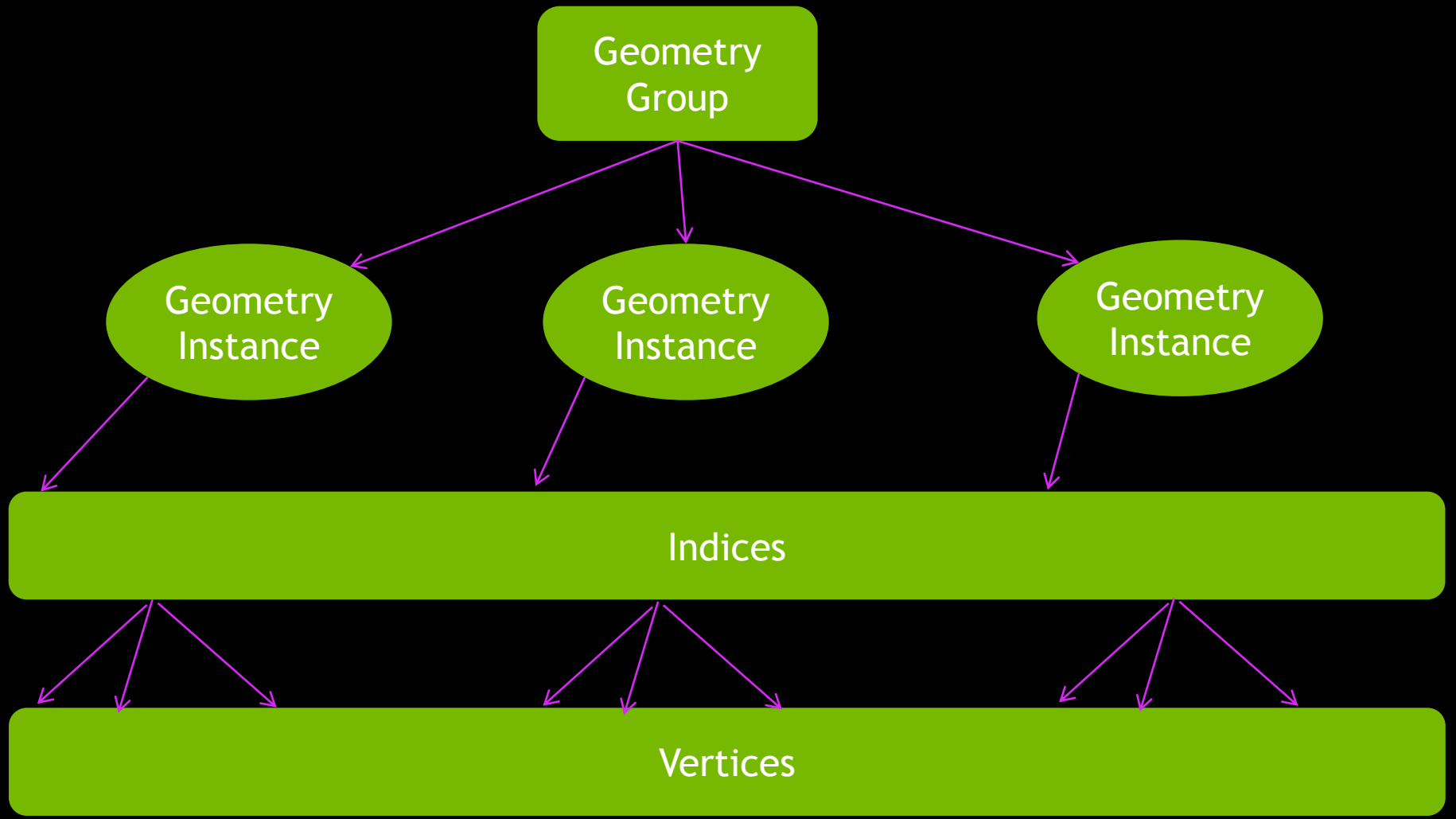
SHARE GRAPH NODES



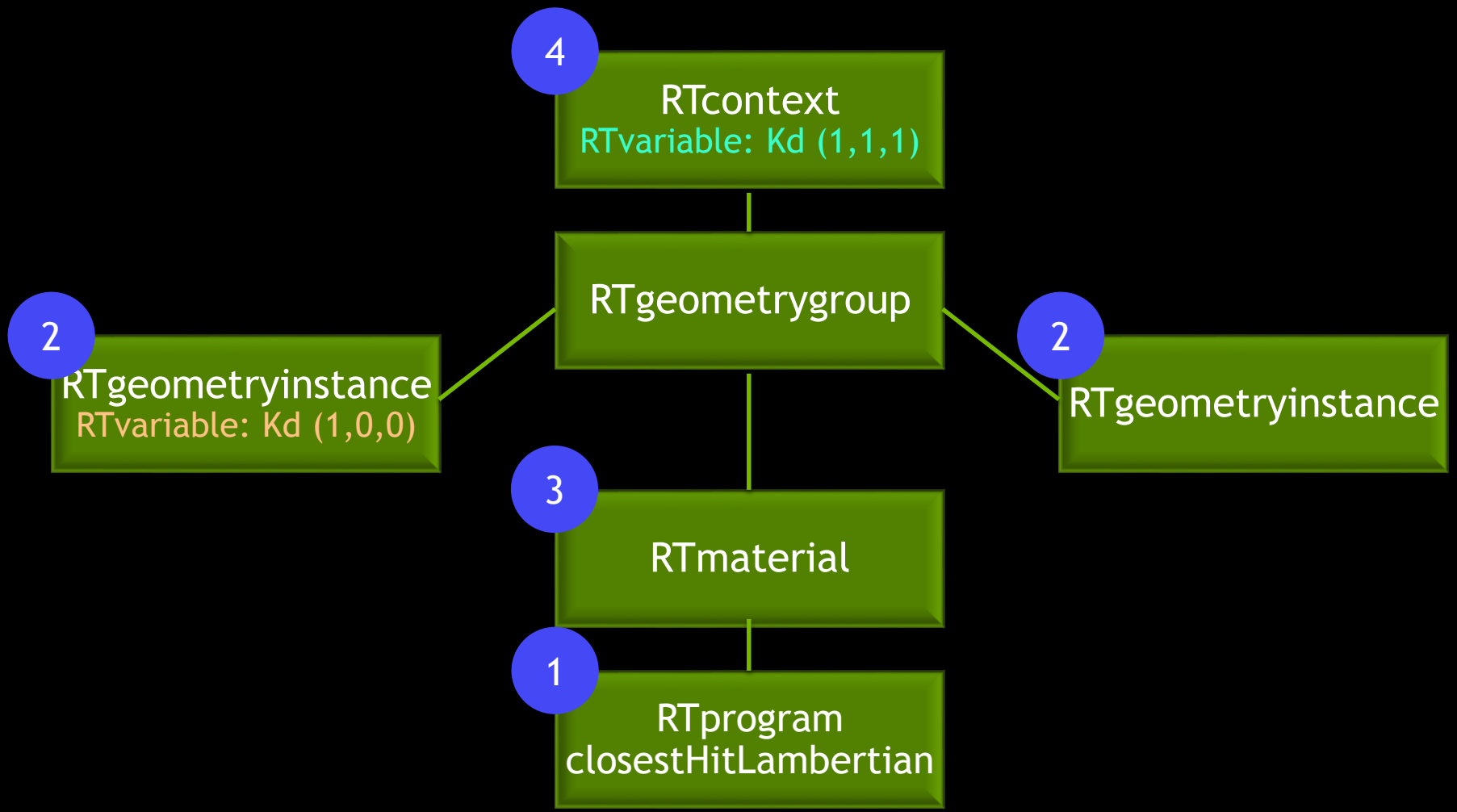
REDUCING NODE GRAPH



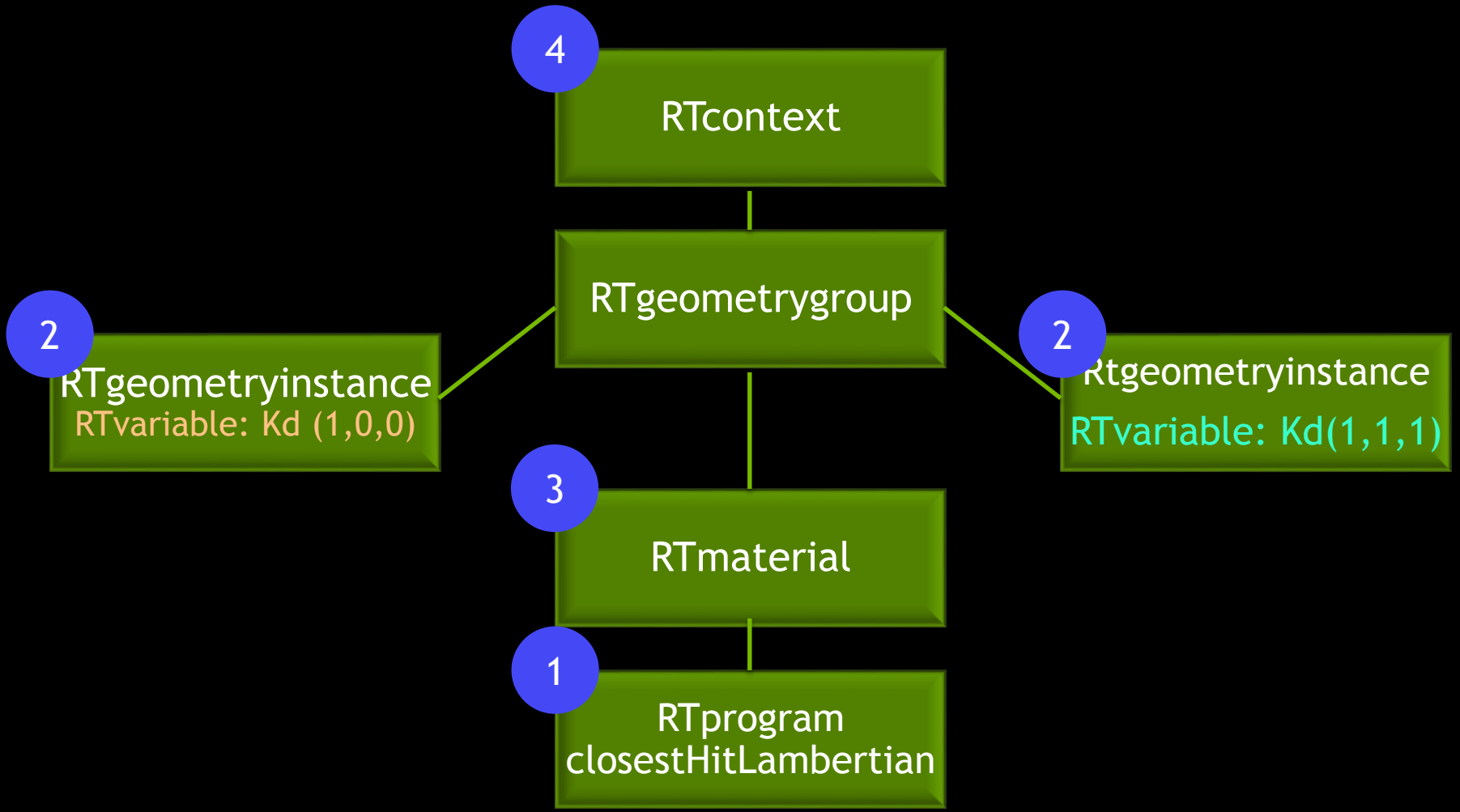
rtGeometrySetPrimitiveIndexOffset (v3.5)



DYNAMIC VARIABLE LOOKUPS



DYNAMIC VARIABLE LOOKUPS



DATA TRANSFER – MAKING IT FAST

- PCI Express throughput can be terrible for non-power-of-two element sizes
  - float3 buffer can transfer significantly slower than float4 buffer
- Working with INPUT_OUTPUT buffers on multiple GPUs:
  - By default OptiX will store the buffer on the host in zero-copy memory
  - Often want to write to a buffer on GPU but never map back to host (e.g. accumulation buffers, variance data, random seed data)
  - Mark the buffer as RT_BUFFER_GPU_LOCAL and OptiX will keep the buffer on the device.
  - Each device can only see the results of buffer elements it has written (usually ok since most threads only read/write to their own rtLaunchIndex)

CALLABLE PROGRAMS SPEED UP COMPILATION

- OptiX inlines all CUDA functions
- ☺ Fast execution
- ☹ Large kernel to compile
- Use callable programs

- Callable programs reduce OCG compile times
- Small rendering performance overhead
- Enables shade trees and plugin rendering architectures

CALLABLE PROGRAMS SPEED UP COMPILATION

```
RT_CALLABLE_PROGRAM float3 checker_color(float3 input_color, float scale)
{
  uint2 tile_size = make_uint2(launch_dim.x / N, launch_dim.y / N);
  if (launch_index.x/tile_size.x ^ launch_index.y/tile_size.y)
    return input_color * scale;
  else
    return input_color;
}

rtCallableProgram(float3, get_color, (float3, float));

RT_PROGRAM camera()
{
  float3 initial_color;
  // … trace a ray, get the initial color …
  float3 final_color = get_color( initial_color, 0.5f );
  // … write new final color to output buffer …
}
```

HIGH PERFORMANCE GRAPHICS 2014

- Lyon, France
- June 23-25
- Paper Submissions Due: April 4
- Poster Submissions Due: May 16
- **Hot3D Submissions Due: May 23**

www.highperformancegraphics.org