Ray Tracing
Distribution Ray Tracing (Cook, 1984)
Path Tracing (Kajiya, 1986)

Figure 6. A sample image. All objects are neutral grey. Color on the objects is due to caustics from the green glass balls and color bleeding from the base polygon.
Why ray tracing?

- Ray tracing unifies rendering of visual phenomena
  - fewer algorithms with fewer interactions between algorithms
- Easier to combine advanced visual effects robustly
  - soft shadows
  - subsurface scattering
  - indirect illumination
  - transparency
  - reflective & glossy surfaces
  - depth of field
  - ...
Real Time Path Tracing

- What would it take?
  - 4 rays / sample
  - 50 samples / pixel
  - 2M pixels / frame
  - 30 frames / second
  - 12B rays / second

- GeForce GTX 680: 350M rays / second
  - Need 36X speedup

1 shading sample 1 AA sample
9 shading samples 1 AA sample
18 shading samples 2 AA samples
36 shading samples 4 AA samples
72 shading samples 8 AA samples
144 shading samples 16 AA samples
Ray Tracing Regimes

Real-time

Interactive

Batch

today
How to optimize ray tracing (or anything)

- Better hardware (GPUs)
- Better software (Algorithmic improvement)
- Better middleware (Tune for the architecture)
GPUs - the processor for ray tracing

- Abundant parallelism, massive computational power
- GPUs excel at shading
- Opportunity for hybrid algorithms
Acceleration Structures
OptiX Goals

- High performance ray tracing
- Simpler ray tracing
- Hide GPU-specific details
- Express most ray tracing algorithms
- Leverage CUDA architecture and compiler infrastructure
General Purpose Ray Tracing

- Rendering, baking, collision detection, A.I. queries, etc.
- Modern shader-centric, stateless and bindless design
- Is not a renderer but can implement many types of renderers
- Algorithm agnostic
  - User defined ray data
  - Programmable intersection
  - Interoperate with rasterization pipeline
Highly Programmable

- Shading with arbitrary ray payloads
- Ray generation/framebuffer operations
  - cameras, data unpacking, etc.
- Programmable intersection
  - triangles, NURBS, implicit surfaces, etc.
Easy to Program

- Write single ray code (no exposed ray packets)
- No need to rewrite shaders to target different hardware
Other OptiX features

- OptiX node graph
  - Programmable traversal
  - Instancing
  - Dynamic scenes
- Double precision arithmetic
- Interop with CUDA, OpenGL and D3D
  - Textures, VBOs, etc.
  - Hybrid rasterization and ray tracing
rtuTraversal API

- Extremely simple ray tracing API
- Useful when only ray hits are needed
  - No shading or recursion
- Works transparently on GPU and CPU

```c
rtuTraversalCreate( );
rtuTraversalMapRays( ); // Provide requested rays
  copy the rays...
rtuTraversalUnmapRays( );
rtuTraversalSetMesh( ); // Provide geometry
rtuTraversalTraverse( );
rtuTraversalMapResults( ); // Get results
  copy the hits...
rtuTraversalUnmapResults( );
rtuTraversalDestroy( ); // Cleanup
```
How OptiX Links Your Code

- Ray Generation
- Material Shading
- Object Intersection
- JIT Compiler
- Acceleration Structures
- Scheduling

CUDA C shaders from user programs

OptiX API
API Objects - Context

- **Manages API Object State**
  - Program Loading
  - Validation and Compilation

- **Manages Acceleration Structures**
  - Building and Updating

- **Provides Entry Points into the system**
  - `rtContextLaunch*D()`
Entry Points and Ray Types

Context

Entry Point 1
- Ray Generation 1
- Exception 1

Entry Point 2
- Ray Generation 2
- Exception 2

Launch
Entry Points and Ray Types

Launch

Ray Shading

Material Programs

Material

Ray Type

<table>
<thead>
<tr>
<th>Ray Type</th>
<th>Closest Hit</th>
<th>Any Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Closest Hit</td>
<td>Any Hit</td>
</tr>
<tr>
<td>1</td>
<td>Closest Hit</td>
<td>Any Hit</td>
</tr>
<tr>
<td>2</td>
<td>Closest Hit</td>
<td>Any Hit</td>
</tr>
<tr>
<td>3</td>
<td>Closest Hit</td>
<td>Any Hit</td>
</tr>
</tbody>
</table>
API Objects - Nodes

- Nodes contain children
  - Other nodes
  - Geometry Instances

- Transforms hold matrices
  - Applied to all children

- Selectors have Visit programs
  - Provide programmable selection of children
  - Similar to “switch nodes”
  - Can implement LOD systems

- Acceleration Structures
  - Builds over children of attached node
The Object Hierarchy
API Objects - Geometry

- **GeometryInstance** binds:
  - Geometry object
  - A collection of Materials
    - Indexed by argument from intersection

- **Geometry**
  - A collection of primitives
    - Intersection Program
    - Bounding Box Program

- **Material**
  - Any Hit Program
  - Closest Hit Program
API Objects - Data Management

- Supports 1D, 2D and 3D buffers
- Buffer formats
  - RT_FORMAT_FLOAT3
  - RT_FORMAT_UNSIGNED_BYTE4
  - RT_FORMAT_USER, etc.
- Other API Interoperability
  - e.g. create buffers from CUDA, OpenGL or D3D buffer objects
- TextureSamplers reference Buffers
  - Attach buffers to MIP levels, array slices, etc.
API Objects - Programmability

- Runs on CUDA
  - Cg-like vectors plus pointers
  - Uses PTX, CUDA’s virtual assembly language
  - C++ wrapper for use with NVCC compiler
- Implements recursion and dynamic dispatch
  - Intrinsic functions: rtTrace(), rtReportIntersection(), etc.
- Programs reference variables by name
- Variables are defined by
  - Static initializers
  - Binding to API Objects in the hierarchy
Deformable Objects

1. Primitives
   - Deform

2. Groups and Acceleration Marked Dirty

3. Context updates Acceleration Structures

Diagram:
- Context
  - Group
    - Acceleration
    - GeometryGroup
      - GeometryInstance
Variables

- Variables are one of:
  - A small primitive type (float4, matrix, ...)
  - A small user defined type
  - A handle to a buffer (1D, 2D, 3D)
  - A texture
  - A handle to a callable program
Variable Scoping

Context

GeometryInstance

Material

Program

Definition: Color = blue

Definition: Color = red

Reference: Color

Material

Context

GeometryInstance

Closest Hit Program
C Host API Sample

RTresult RTAPI rtContextCreate (RTcontext* context);
RTresult RTAPI rtContextDestroy (RTcontext context);
RTresult RTAPI rtContextDeclareVariable (RTcontext context, const char* name, RTvariable* v);
RTresult RTAPI rtContextSetRayGenerationProgram (RTcontext context, unsigned int entry_point_index, RTprogram program);
RTresult RTAPI rtBufferCreate (RTcontext context, unsigned int bufferdesc, RTbuffer* buffer);
RTresult RTAPI rtBufferSetFormat (RTbuffer buffer, RTformat format);
RTresult RTAPI rtBufferMap (RTbuffer buffer, void** user_pointer);
RTresult RTAPI rtBufferUnmap (RTbuffer buffer);
RTresult RTAPI rtProgramCreateFromPTXString (RTcontext context, const char* ptx, const char* program_name, RTprogram* program);
RTresult RTAPI rtProgramCreateFromPTXFile (RTcontext context, const char* filename, const char* program_name, RTprogram* program);
RTresult RTAPI rtContextLaunch2D (RTcontext context, unsigned int entry_point_index, RTsize image_width, RTsize image_height);
C++ Host API Sample

Context* context = Context::create();
context["max_depth"]->setInt( 5 );
context["scene_epsilon"]->setFloat( 1.e-4f );

// Ray gen program
Program ray_gen_program = context->createProgramFromPTXFile( "myprogram.ptx", "pinhole_camera" );
context->setRayGenerationProgram( 0, ray_gen_program );

BasicLight lights[] = { ..... };
Buffer light_buffer = context->createBuffer(RT_BUFFER_INPUT);
light_buffer->setFormat(RT_FORMAT_USER);
light_buffer->setElementSize(sizeof(BasicLight));
light_buffer->setSize( sizeof(lights)/sizeof(lights[0]) );
memcpy(light_buffer->map(), lights, sizeof(lights));
light_buffer->unmap();
context["lights"]->set(light_buffer);
Life of a ray

1. Ray Generation
2. Intersection
3. Shading

- Pinhole Camera
- Payload float3 color
- Ray-Sphere Intersection
- Lambertian Shading
Program objects (shaders)

- Interconnection of programs defines the outcome
- Data associated with ray is programmable
- Input “language” is CUDA C/C++
  - No new language to learn
  - Powerful language features available immediately
  - Can also take raw PTX as input
- Caveat: still need to use it responsibly to get performance
Closest Hit Programs: called once after traversal has found the closest intersection
   – Used for traditional surface shading
   – Deferred shading

Any Hit Programs: called during traversal for each potentially closest intersection
   – Transparency without traversal restart (can read textures):
     `rtIgnoreIntersection()`
   – Terminate shadow rays that encounter opaque objects:
     `rtTerminateRay()`

Both can be used for shading by modifying per ray state
Flexible Intersection

Intersection (miss)
Intersection (hit)
Any Hit
Intersection (miss)
Intersection (hit)
Any Hit: rtlIgnoreIntersection
Closest Hit
Flexible Intersection

- **Intersection (hit)**
  - Any Hit: rtIgnoreIntersection

- **Intersection (hit)**
  - Any Hit: rtTerminateRay

- **Closest Hit**
Per Ray Data and Attributes

- **Per Ray Data**
  - User-defined struct attached to rays
  - Can be used to pass data up and down the ray tree
  - Varies per Ray Type

- **Arbitrary Attributes**
  - Produced by Intersection Programs
  - Consumed by Any Hit and Closest Hit Programs
NVIDIA® OptiX™ Ray Tracing Engine

Quickstart Guide

Version 3.0

3/15/2013
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
- Most common
Normal shader
struct PerRayData_radiance
{
    float3 result;
};

rtDeclareVariable(PerRayData_radiance, prd_radiance, rtPayload,);
rtDeclareVariable(float3, shading_normal, attribute shading_normal,);

RT_PROGRAM void closest_hit_radiance()
{
    float3 worldnormal = normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, shading_normal));
    prd_radiance.result = worldnormal * 0.5f + 0.5f;
}
Normal shader
Lambertian shader
rtBuffer<BasicLight> lights;

rtDeclareVariable(optix::Ray, ray, rtIncomingRay, );
rtDeclareVariable(float, t_hit, rtIntersectionDistance, );

RT_PROGRAM void closest_hit_radiance()
{
  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 hit_point = ray.origin + t_hit * ray.direction;

  float3 color = Ka * ambient_light_color;
  for(int i = 0; i < lights.size(); ++i) { // Loop over lights
    BasicLight& light = lights[i];
    float3 L = normalize(light.pos - hit_point);
    float nDl = dot( ffnormal, L);

    if( nDl > 0 )
      color += Kd * nDl * light.color;
  }

  prd_radiance.result = color;
}
Lambertian shader
Adding shadows
Ray Payloads

- Can define arbitrary data with the ray
- Sometimes called the “per ray data”
- Data can be passed down or up the ray tree (or both)
- Just a user-defined struct accessed by all shader programs
- Varies per ray type
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 = 1.0f;
        float Ldist = length(light.pos - hit_point);
        Ray shadow_ray = make_Ray(hit_point, L, 1, scene_epsilon, Ldist);
        rtTrace(top_shadower, shadow_ray, shadow_prd);
        float light_attenuation = shadow_prd.attenuation;

        if( light_attenuation > 0.0f ){
            float3 Lc = light.color * light_attenuation;
            color += Kd * nDl * Lc;
        }
    }
}
}
Adding shadows
Adding reflections
struct PerRayData_radiance {
    float3 result;
    int depth;
};

... // reflection ray
if (prd.depth < max_depth) {
    float3 R = reflect( ray.direction, ffnormal );
    Ray refl_ray = make_ray( hit_point, R, 0, scene_epsilon, RT_DEFAULT_MAX );
    rtTrace(top_object, refl_ray, refl_prd);
    color += reflectivity * refl_prd.result;
}
struct PerRayData_radiance {
    float3 result;
    float importance;
    int depth;
};

...  

float importance = prd.importance * luminance( reflectivity );

// reflection ray
if( importance > importance_cutoff && prd.depth < max_depth ) {
    PerRayData_radiance refl_prd;
    refl_prd.importance = importance;
    refl_prd.depth = prd.depth + 1;
    float3 R = reflect( ray.direction, ffnormal );
    Ray refl_ray = make_ray( hit_point, R, 0, scene_epsilon, RT_DEFAULT_MAX );
    rtTrace( top_object, refl_ray, refl_prd );
    color += reflectivity * refl_prd.result;
}
Adding reflections
Environment map
Miss program

- Defines what happens when a ray misses all objects
- Accesses ray payload
- Usually - background color
rtDeclareVariable(float3, bg_color, ,);
rtDeclareVariable(PerRayData_Radiance, prd_radiance, ,);
RT_PROGRAM void miss()
{
    prd_radiance.result = bg_color;
}
rtTextureSampler<float4, 2> envmap;
rtDeclareVariable(PerRayData_Radiance, prd_radiance, ,);

RT_PROGRAM void envmap_miss()
{
    float theta = atan2f(ray.direction.x, ray.direction.z);
    float phi = M_PI * 0.5f - acosf(ray.direction.y);
    float u = (theta + M_PI) * (0.5f * M_1_PI);
    float v = 0.5f * (1.0f + sin(phi));
    prd_radiance.result = make_float3(tex2D(envmap, u, v));
}
Environment map
Schlick approximation
float3 r = fresnel_schlick(-dot(ffnormal, ray.direction), reflectivity_n);
float importance = prd.importance * luminance( r );

// reflection ray
if( importance > importance_cutoff && prd.depth < max_depth) {
    PerRayData_radiance refl_prd;
    refl_prd.importance = importance;
    refl_prd.depth = prd.depth+1;
    float3 R = reflect( ray.direction, ffnormal );
    Ray refl_ray = make_ray( hit_point, R, 0,
        scene_epsilon, RT_DEFAULT_MAX );
    rtTrace(top_object, refl_ray, refl_prd);
    color += r * refl_prd.result;
}
Schlick approximation
Tiled floor
... float t_hit = incoming_ray_t.get();
float3 hit_point = ray.origin + t_hit * ray.direction;

float v0 = dot(tile_v0, hit_point);
float v1 = dot(tile_v1, hit_point);
v0 = v0 - floor(v0);
v1 = v1 - floor(v1);

float3 local_Kd;
if( v0 > crack_width && v1 > crack_width ){
    local_Kd = Kd;
} else {
    local_Kd = crack_color;
}
...
Tiled floor
Rusty metal
Direct port of Larry Gritz's rusty metal shader

```c
rtDeclareVariable(float, metalKa) = 1;
rtDeclareVariable(float, metalKs) = 1;
rtDeclareVariable(float, metalKa) = .1;
rtDeclareVariable(float, rustKa) = 1;
rtDeclareVariable(float, rustKd) = 1;
rtDeclareVariable(float, rustcolor) = (.437, .084, 0);
rtDeclareVariable(float3, metalcolor) = (.7, .7, .7);
rtDeclareVariable(float, rustiescale) = .02;
rtDeclareVariable(float, rustbump) = 0.85;

#define MAXOCTAVES 6

RT_PROGRAM void box_closest_hit_radiance() {
    PerRayData_radiance& prd = prd_radiance_reference();
    Ray ray = incoming_ray.get();

    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 );
    float t_hit = incoming_ray_t.get();
    float3 hit_point = ray.origin + t_hit * ray.direction;

    // Number of octaves by the estimated change in PP between adjacent shading samples.
    float3 PP = txtscale * hit_point;
    float a = 1;
    float sum = 0;
    for(int i = 0; i < MAXOCTAVES; i++){
        sum += a * fabs(snoise(PP));
        PP *= 2; a *= 0.5;
    }

    // Scale the rust appropriately, modulate it by another noise computation, then sharpen it by squaring its value.
    float rustiness = step(1-rusty, clamp(sum,0.0f,1.0f));
    rustiness *= clamp(abs(snoise(PP)), 0.0f, .08f) / .08f;
    rustiness *= rustiness;

    if (rustiness > 0) {
        float3 Nrust = normalize(ffnormal + rustbump * snoise(PP));
        Nrust = faceforward(Nrust, -ray.direction, world_geo_normal);

        float3 color = mix(metalcolor * metalKa, rustcolor * rustKa, rustiness) * ambient_light_color;
        for(int i = 0; i < lights.size(); ++i) {
            BasicLight light = lights[i];
            float3 L = normalize(light.pos - hit_point);
            float nmDl = dot( ffnormal, L );
            float nlDl = dot( rustKd * rustcolor * nmDl, L );
            float r = nmDl * (1.0f-rustiness);
            if(nmDl > 0.0f) {
                float3 H = normalize(L - ray.direction);
                float nmDh = dot( ffnormal, H );
                if(nmDh > 0) {
                    color += r * metaKs * Lc * pow(nmDh, 1.f/metalroughness);
                }
            }
        }

        float3 R = schlick(dot(ffnormal, ray.direction), reflectivity_n * (1-rustiness));
        float importance = prd.importance * luminance( r );

        if( light_attenuation > 0.0f ) {
            float3 Lc = light.color * light_attenuation;
            nmDl = max(nmDl * rustiness, 0.0f);
            color += rustKd * rustcolor * nmDl * Lc;
        }
    }

    if (importance > importance_cutoff && prd.depth < max_depth) {
        PerRayData_radiance refl_prd;
        refl_prd.importance = importance;
        refl_prd.depth = prd.depth+1;
        float3 R = reflect( ray.direction, ffnormal );
        Ray refl_ray = make_ray( hit_point, R, 0, scene_epsilon, RT_DEFAULT_MAX );
        rtTrace(top_shadower, refl_ray, refl_prd);
        color += r * refl_prd.result;
    }
}

prd.result = color;
```
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.
Convex hull object

- Defined by a set of planes
- Created by the host
- Simple algorithm can handle any number of planes
  - Find last plane “entered”
  - Find first plane “exited”
  - Degenerate interval: miss
rtBuffer<float4> planes;
RT_PROGRAM void hull_intersect(int primIdx)
{
    const Ray ray = incoming_ray.get();

    int n = planes.size();
    float t0 = -FLT_MAX;
    float t1 = FLT_MAX;
    float3 t0_normal = make_float3(0);
    float3 t1_normal = make_float3(0);
    for(int i = 0; i < n; ++i ) {
        float4 plane = planes[i];
        float3 n = make_float3(plane);
        float d = plane.w;
        float denom = dot(n, ray.direction);
        float t = -(d + dot(n, ray.origin))/denom;
        if( denom < 0){
            // enter
            if(t > t0){
                t0 = t;
                t0_normal = n;
            }
        }}
    if(t0 > t1)
        return;
    if(rtPotentialIntersection( t0 )){
        shading_normal = geometric_normal = t0_normal;
        rtReportIntersection(0);
    } else if(rtPotentialIntersection( t1 )){
        shading_normal = geometric_normal = t1_normal;
        rtReportIntersection(0);
    } else {
        //exit
        if(t < t1){
            t1 = t;
            t1_normal = n;
        }
    }
    if(t0 > t1)
        return;
Adding primitives
Tweaking the shadow
**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)
rtDeclareVariable(PerRayData_shadow, prd_shadow, ,);

RT_PROGRAM void any_hit_shadow()
{
    // this material is opaque,
    // so it fully attenuates all shadow rays
    prd_shadow.attenuation = 0;

    rtTerminateRay();
}
rtDeclareVariable(float, shadow_attenuation);

RT_PROGRAM void glass_any_hit_shadow()
{
    Ray ray = incoming_ray.get();
    float3 world_normal = normalize(rtTransformNormal(RT_OBJECT_TO_WORLD, shading_normal));
    float nDi = fabs(dot(world_normal, ray.direction));

    prd.attenuation *= 1-fresnel_schlick(nDi, 5, 1-shadow_attenuation, 1);

    rtIgnoreIntersection();
}
Tweaking the shadow
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
rtDeclareVariable(uint2, launchIndex, rtLaunchIndex);

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

    Ray ray = make_ray(ray_origin, ray_direction, radiance_ray_type, 
                       scene_epsilon, RT_DEFAULT_MAX);

    PerRayData_radiance prd;
    prd.importance = 1.f;
    prd.depth = 0;

    rtTrace(top_object, ray, prd);

    output_buffer[index] = make_color(prd.result);
}
RT_PROGRAM void env_camera()
{
  float2 d = make_float2(launchIndex) / make_float2(output_buffer.size());
  d = d * make_float2(2.0f * M_PI, M_PI) + make_float2(M_PI, 0);
  float3 angle = make_float3(cos(d.x) * sin(d.y), -cos(d.y), sin(d.x) * sin(d.y));

  float3 ray_origin = eye;
  float3 ray_direction = normalize(angle.x*normalize(U) + angle.y*normalize(V) +
                                   angle.z*normalize(W));

  Ray ray = make_ray(ray_origin, ray_direction, radiance_ray_type, scene_epsilon,
                     RT_DEFAULT_MAX);

  PerRayData_radiance prd;
  prd.importance = 1.f;
  prd.depth = 0;

  rtTrace(top_object, ray, prd);

  output_buffer[index] = make_color( prd.result );
}
CUDA Interoperability

- Sharing Contexts
- Sharing Pointers
- Multi-GPU
Sharing CUDA Contexts

- There is a CUcontext on each device
- CUDA runtime silently manages these
- OptiX used to create its own CUcontexts
- Now we share with CUDA:
  - If CUDA runtime runs first we find its CUcontexts
  - If OptiX runs first we make new CUcontexts
Sharing Pointers with CUDA

- `rtBufferSetDevicePointer()` - CUDA owns the buffer

```c
const float* d_out_probe_buf;
cudaSetDevice(0);
cudaMalloc(&d_output_probe_buffer, moving_obj_count * sizeof(float));
rtBufferSetDevicePointer(buf, optixDevice0, d_out_probe_buffer);
rtContextLaunch1D(..., moving_obj_count);

LOS_reduction<<<moving_obj_count, 1>>>(d_moving_objs, d_out_probe_buf);
```
Sharing Pointers with CUDA

- `rtBufferGetDevicePointer()` - OptiX owns the buffer

```c
{
    rtContextLaunch1D(..., moving_obj_count);
    const float* d_out_probe_buf;
    rtBufferGetDevicePointer(buf, optixDevice0, &d_out_probe_buf);
    
    cudaSetDevice(0);
    LOS_reduction<<<moving_obj_count, 1>>>(d_moving_objs, d_out_probe_buf);
}
```
Collision Sample

- OptiX OUTPUT buffer used by CUDA
- OptiX, CUDA, and OpenGL
Ocean cuFFT Sample

- Tessendorf FFT-based ocean surface algorithm
  - Uses cuFFT
  - 1024x512 simulation
  - 1024x1024 height field primitive

- Water with Fresnel dielectric shading model
  - 6 bounce reflection; 6 bounce refraction

- Preetham physically-based sky model miss program

- CUDA owns the buffer; OptiX uses it as RT_INPUT buffer

- Reinhard tone mapping on RT_OUTPUT buffer
Ray Tracing in the Abstract

- Given a ray \((O, D)\) and a geometric dataset find
  - any hit
  - closest hit
  - all hits

- Current datasets \(~1M \rightarrow 100M\) primitives, usually triangles
- Use a spatial data structure optimized for these operations
- Datasets can also include GB of other data like textures
Scene 1: Column Fracture

Fracture

GPU Rigid Bodies, Max ray depth = 12, ~350,000 triangles
Vertex Light Baking

- Working with Bungie
  - But available publicly. *Just ask us.*

- Compared to textures...
  - Less memory & bandwidth
  - No $u,v$ parameterization
  - Good for low-frequency effects
Typical Scene

- Linear interpolation
- Static mesh
- Coarse mesh
Piecewise Linear Approximation

- Sample illumination on surface
- Each sample is a hemisphere of rays
- Reconstruct values at vertices
Weighted Averaging
Least Squares Vertex Baking
Radiation Heat Transfer
OptiX 3.0 SDK

- Available for free: Windows, Linux, Mac