Agenda

1. A “ray tracing pipeline”

2. OptiX crash course

3. Adding ray tracing to a rasterization pipeline

4. A few use cases
Raster pipelines as a software framework

- Raster pipeline evolved from HW design
  - Not really a pipeline anymore
  - Execution model not specified
- Frameworks have 4 features (wikipedia)
  - Inversion of control
  - Default behavior
  - Extensibility
  - Non-modifiable framework code
- Provides separation of concerns
Alternative pipelines

- Freepipe (Liu et al 2010)
- CUDA Raster (Laine/Karras 2011)
- Voxelpipe (Pantaleoni 2011)
- GRAMMPS (Sugerman et al 2009)
Life of a ray

1. Ray Generation
2. Intersection
3. Shading

Pinhole Camera
Ray-Sphere Intersection
Lambertian Shading

Payload float3 color
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);
}

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);
        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);
            }
        }
    }

    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 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;
}

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 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);
    }
}

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 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);
    }
}
The ensemble of programs defines the rendering algorithm (or collision detection algorithm, or sound propagation algorithm, etc.)
Programming in OptiX

- Interconnection of programs defines the outcome
  - Whitted ray tracing, cook, path tracing, photon mapping
  - Or collision detection, sound propagation, ...

- Input “language” is based on CUDA C
  - No new language to learn
  - Powerful language features available immediately
    - Pointers
    - Templates
    - Overloading
    - Default arguments
    - Classes (no virtual functions)
  - Can also take raw PTX as input

- Caveat: still need to use it responsibly to get performance
All objects can have multiple parents (instancing)
Geometry data
Materials
Acceleration structures
Tree may have multiple roots
Separate objects for shadow rays
Not shown: Selector
Programmable (LOD, switch, etc.)
Acceleration Structure Choices

- **Sbvh**: Splits sliver triangles  
  - Fastest ray tracing
- **Bvh**: Standard CPU build
- **MedianBvh**: Fast CPU build
- **Lbvh**: Built on GPU  
  - Fastest build time

- Fairy Forest (174K triangles) – 4.8ms
- Turbine Blade (2M triangles) – 10.5ms
- Power Plant (12M triangles) – 62ms
OptiX Functional Overview

- Ray Generation
- Material Shading
- Object Intersection

CUDA C shaders from user program

JIT Compiler

OptiX API

Acceleration Structures

Scheduling

GPU Execution via CUDA
Execution models

OptiX pipeline does not completely specify an execution model

- Sequential consistency within a thread (usually pixel/sample)
- No consistency between threads
- No ordering guarantees for intersection tests
- Side-effects limited to output buffers and ray payloads

Many possible execution models

- Allows for HW/SW innovation

1. Map launchIndex $\rightarrow$ CUDA thread

2. Sequential execution within each thread

3. Map launchIndex $\rightarrow$ CUDA thread

4. Explicitly manage divergence in a MIMD state machine

5. Defer work when data not present

- Initiate rays in large-ish batches
- Sort or scan for state and data coherence
- Process batches until done

1. Initiate rays to fill warp

2. Manage on-chip queues to process each state
Material Programs: OptiX workhorse

- **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()`
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.
RT_PROGRAM void
  triangle_intersect(int)
{
  // Intersect ray with triangle
  float3 e0 = p1 - p0;
  float3 e1 = p0 - p2;
  float3 n = cross( e0, e1 );

  float v = dot( n, ray.direction );
  float r = 1.0f / v;

  float3 e2 = p0 - ray.origin;
  float va = dot( n, e2 );
  float t = r*va;

  if(t < ray.tmax && t > ray.tmin) {
    float3 i = cross( e2, ray.direction );
    float v1 = dot( i, e1 );
    float beta = r*v1;
    if(beta >= 0.0f){
      float v2 = dot( i, e0 );
      float gamma = r*v2;
      if( (v1+v2)*v <= v*v && gamma >= 0.0f )
      {
        if( rtPotentialIntersection( t ) ) {
          shading_normal = geometric_normal = -n;
          rtReportIntersection( 0 );
        }
      }
    }
  }
}
RT_PROGRAM void mesh_intersect(int primIdx)
{
    uint3 v_idx = index_buffer[primIdx];
    float3 p0 = vertex_buffer[v_idx].p0;
    float3 p1 = vertex_buffer[v_idx].p1;
    float3 p2 = vertex_buffer[v_idx].p2;

    // Intersect ray with triangle
    float3 e0 = p1 - p0;
    float3 e1 = p0 - p2;
    float3 n = cross( e0, e1 );

    float v = dot( n, ray.direction );
    float r = 1.0f / v;
    ...
}
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 (shade once per ray)
- Can recursively shoot more rays
  - Shadows
  - Reflections
  - Ambient occlusion
- Most common
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
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
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 color = Ka * ambient_light_color;

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

    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
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, shadow_ray_type, 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;
        }
    }
}
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();
}
Adding Shadows
rtTextureSampler<float, 3> noise_texture;
RT_FUNCTION float snoise(float3 p)
{
    return tex3D(noise_texture, p.x, p.y, p.z) * 2 - 1;
}

RT_FUNCTION float turbulence(float3 p)
{
    float3 PP = p * .04f;
    float a = 1.f;
    float sum = 0;
    for(int i = 0; i < MAXOCTAVES; ++i) {
        sum += a * fabs(snoise(PP));
        PP *= 2.f;
        a *= .5f;
    }
    return sum;
}

RT_FUNCTION float3 solid_texture(float3 p)
{
    float3 c0 = make_float3(.333, .765, .0);
    float3 c1 = c0 * .25f;
    return lerp(c0, c1, turbulence(p));
}
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, radiance_ray_type,
                           scene_epsilon, RT_DEFAULT_MAX );
    rtTrace(top_object, refl_ray, refl_prd);
    color += reflectivity * refl_prd.result;
}
Adding Reflections
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 Maps
unsigned int seed = rnd_seeds[launch_index];
const float inv_sqrt_samples = 1.f / float(sqrt_diffuse_samples);
for(int i = 0; i < sqrt_diffuse_samples; ++i) {
    for(int j = 0; j < sqrt_diffuse_samples; ++j) {
        float u1 = (float(i) + rnd(seed)) * inv_sqrt_samples;
        float u2 = (float(j) + rnd(seed)) * inv_sqrt_samples;

        float3 dir;
        cosine_sample_hemisphere(u1, u2, dir);
        onb.inverse_transform(dir);

        PerRayData_radiance radiance_prd;
        radiance_prd.importance = prd_radiance.importance * luminance(Kd);
        radiance_prd.depth = prd_radiance.depth + 1;

        optix::Ray radiance_ray = optix::make_Ray(hit_point, dir, radiance_ray_type,
                                                  scene_epsilon, RT_DEFAULT_MAX);
        rtTrace(top_object, radiance_ray, radiance_prd);

        result += radiance_prd.result;
    }
}
result *= Kd/(sqrt_diffuse_samples*sqrt_diffuse_samples);
Indirect Illumination
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);
}
rtDeclareVariable(uint2, launchIndex, rtLaunchIndex,);

RT_PROGRAM void accumulation_camera()
{
    ...

    rtTrace(top_object, ray, prd);

    float4 acc_val = accum_buffer[launch_index];
    if( frame > 0 )
    {
        acc_val += make_float4(prd.result, 0.f);
    }
    else
    {
        acc_val = make_float4(prd.result, 0.f);
    }

    output_buffer[launch_index] = make_color(make_float3(acc_val) * 1.f/(float(frame+1)));
    accum_buffer[launch_index] = acc_val;

}
Accumulation Camera
Adding ray tracing to a rasterization pipeline

**Rasterize G-Buffer**
- Hit positions
- Normals

**Ray trace**
- Shadow rays
- Reflection rays
- A/O

** Deferred shading**
- Filter
- Composite
Hybrid Rendering - Reflections

- Render G-buffer
- Trace requests in OptiX
- Use result in traditional or deferred shading
Hybrid Rendering - Shadows
Hybrid: speed/realism dial for interactive

- Combined as a Scene Effect in Cg
- Or directly with D3D or OpenGL

+ Accurate/Glossy Reflections
+ Accurate/Soft Shadows
+ Ambient Bounce/Occlusion,
+ Photon Mapping etc...
Image Space Photon Mapping
Production Hybrid Example: Works Zebra

- back plate
- dx car
- raytrace
NVIDIA ARC: OptiX ray tracing engine
NVIDIA ARC: mental ray AO

- mental ray 3.9 code & pipeline accelerated w/ OptiX

~21mil faces

+3 minutes 2 CPU

20 seconds 1 GPU

+18X

Rendered with mental ray 3.9

Model courtesy NVIDIA Creative
Workflow acceleration: Baking
Computation: collision detection, path planning
Summary

- Pipelines/frameworks provide separation of concerns
  - Enables fixed-function stages
  - Separates actual execution model from mental model
- Pipelines can be used together

Other features:
- Thread scheduling and divergence management
- Multi-GPU scalability
- Virtualization of texturing resources
- Large dataset paging

Not possible with a fully general program
OptiX SDK

Available to registered developers at http://developer.nvidia.com