Interactive ray tracing with the **NVIDIA®OptiX™** engine

Steven Parker, OptiX development mananger

OptiX ray tracing engine

OVERVIEW

OptiX ray tracing engine - Overview

- A General Purpose Ray Tracing API
  - 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
- 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

Programmable Operations

Rasterization	Ray Tracing
Fragment	Closest Hit Any Hit
Vertex Geometry	Intersection Selector
	Ray Generation Miss Exception

The ensemble of programs defines the rendering algorithm

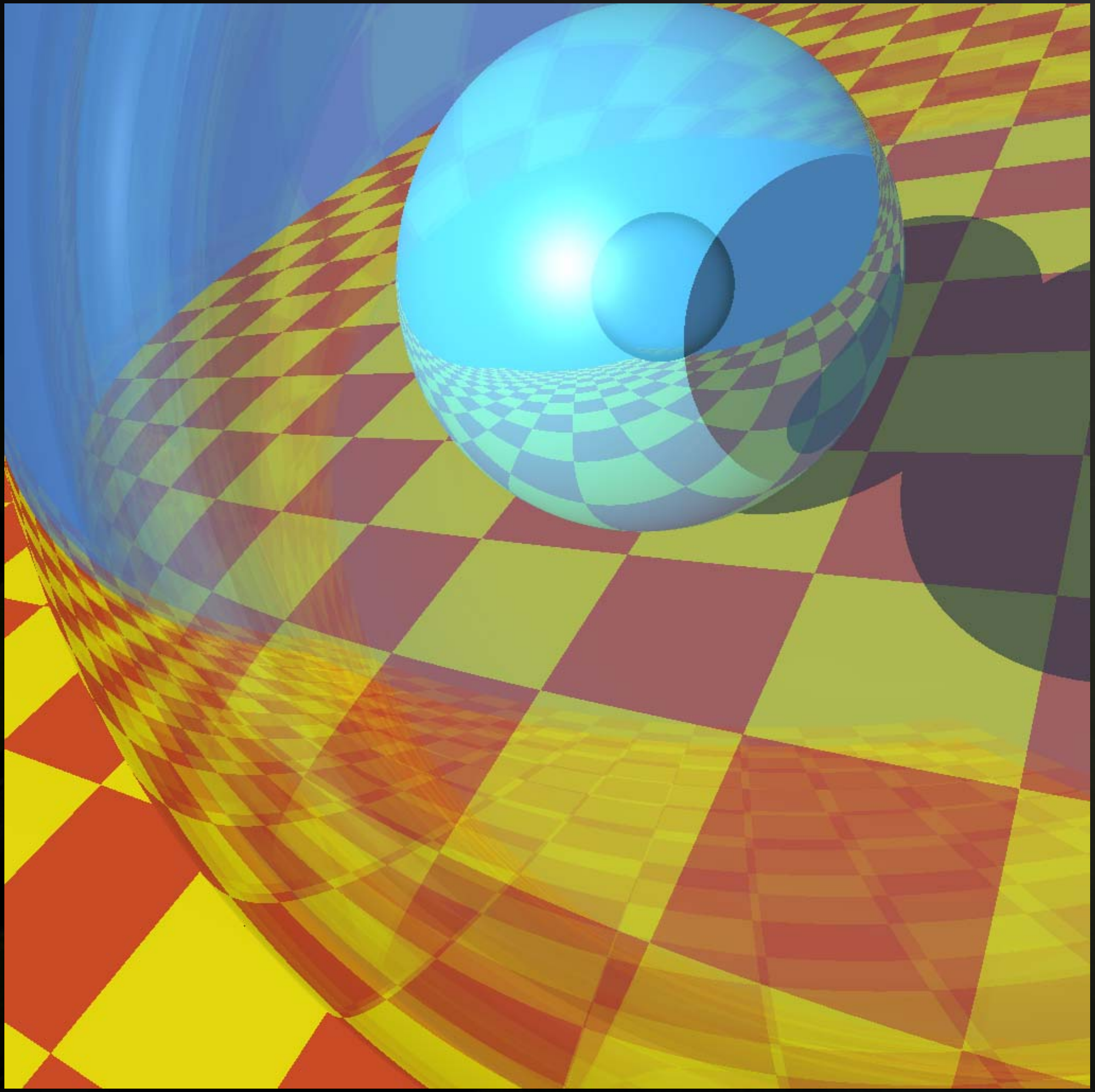
(or collision detection algorithm, or sound propagation algorithm, etc.)

- 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

Today's example – Whitted style ray tracing



Shading in OptiX

- Interconnection of shaders defines the outcome
  - Whitted ray tracing, cook, path tracing, photon mapping
  - Or collision detection, sound propagation, …
- Shading “language” is based on C/C++ for CUDA
  - No new language to learn
  - Powerful language features available immediately
    - Pointers
    - Templates
    - Overloading
    - Default arguments
    - Classes (no virtual functions)
- Adds a powerful object model designed for ray tracing
- Caveat: still need to use responsibly for performance

Anatomy of a shader

includes

declarations

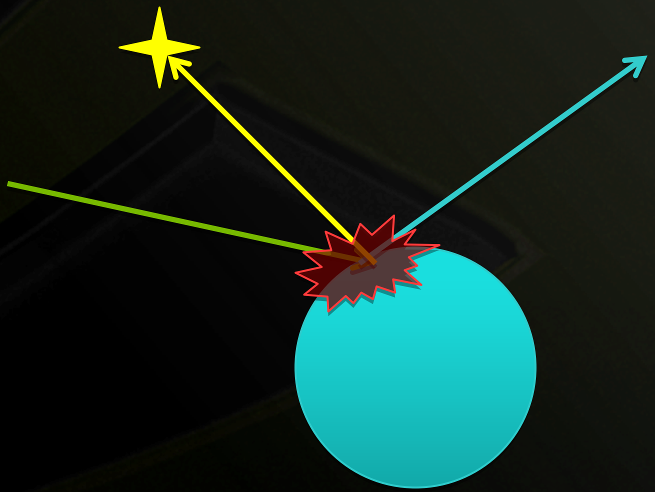
- variables - shader state (read only)
- textures – 1,2,3D (read only)
- buffers – 1,2,3D (read/write)

shader programs

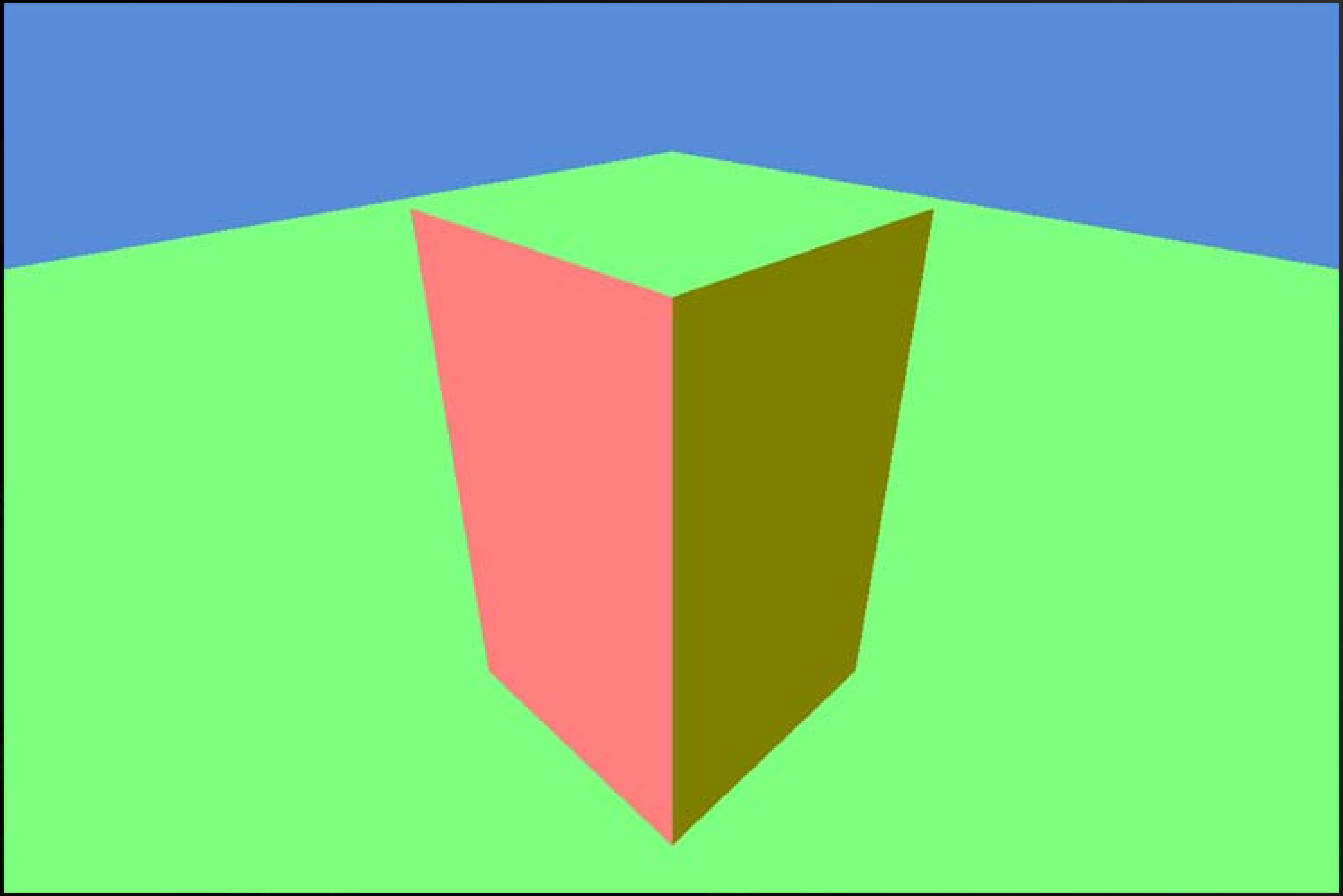
- multiple allowed

Closest hit program (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

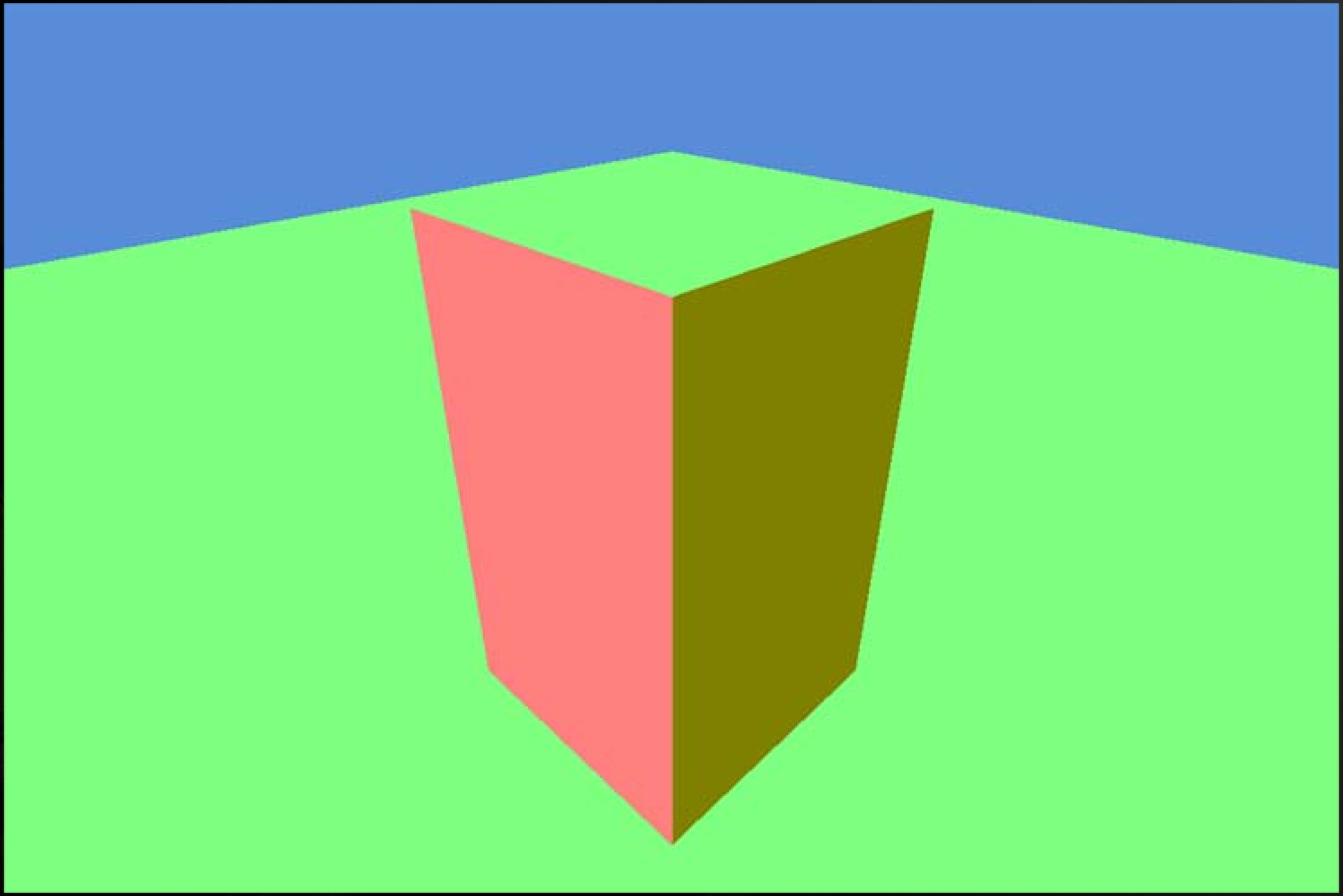


Normal shader - goal

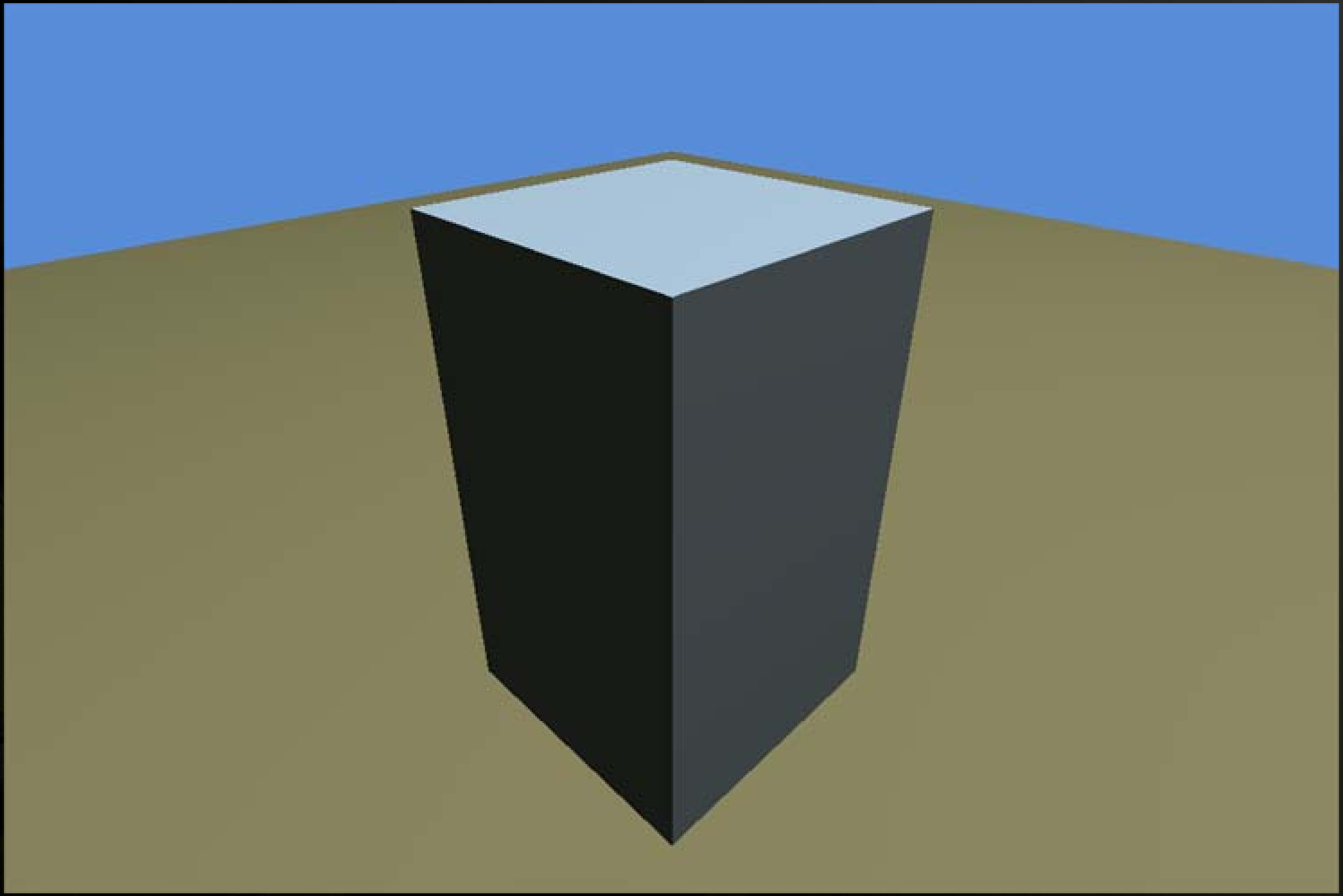


```
struct PerRayData_radiance
{
  float3 result;
};

rtDeclareRayData(PerRayData_radiance, prd_radiance);
rtDeclareAttribute(float3, shading_normal);

RT_PROGRAM void closest_hit_radiance()
{
  PerRayData_radiance& prd = prd_radiance.reference();
  float3 worldnormal = normalize(rtTransformNormal(RT_OBJECT_TO_WORLD,
                                 shading_normal));
  prd.result = worldnormal * 0.5f + 0.5f;
}
```

Normal shader - result

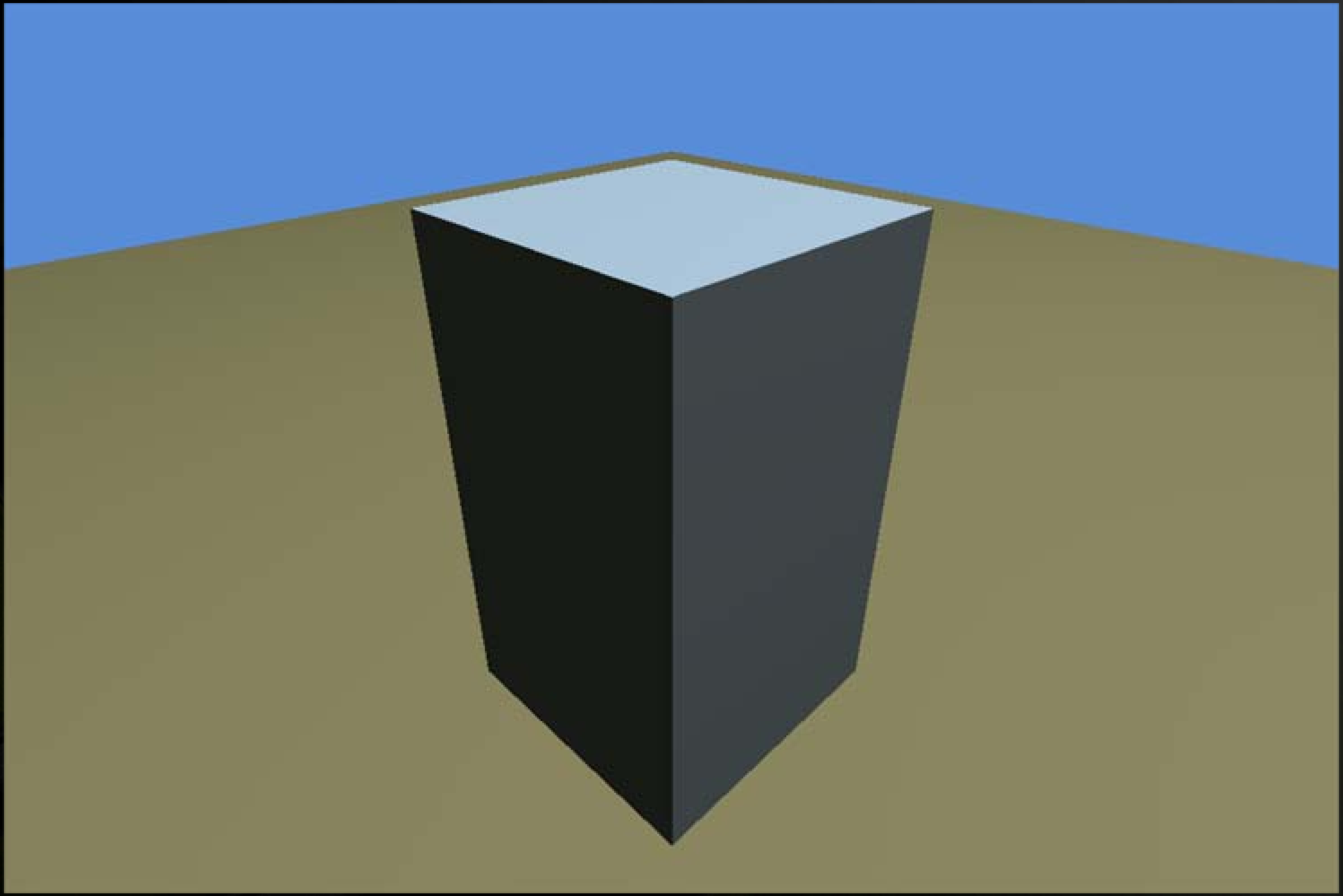


Lambertian shader - goal

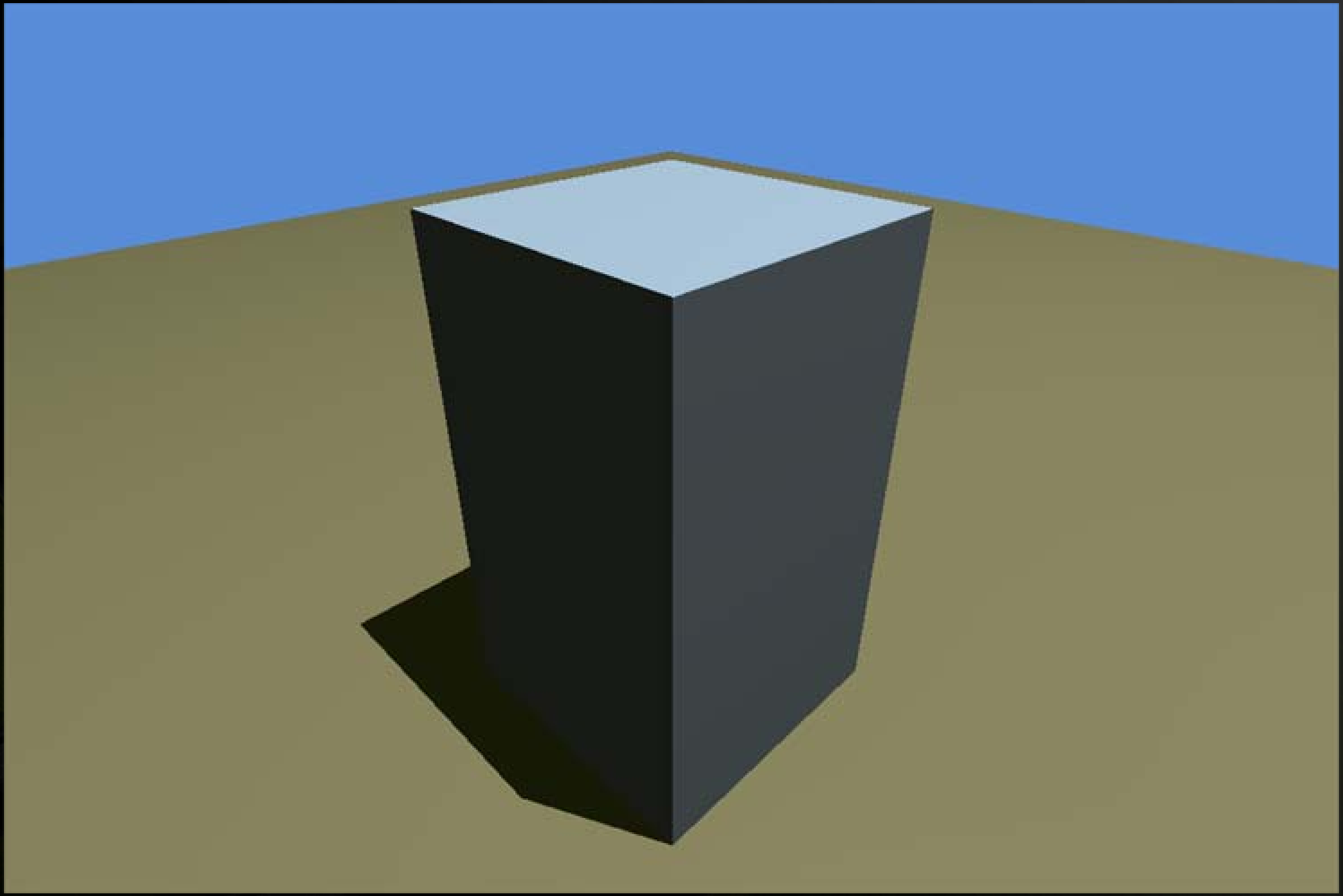


```
rtDeclareVariable(float3, Ka);
rtDeclareVariable(float3, Kd);
rtDeclareVariable(float3, ambient_light_color);
rtBuffer<BasicLight> lights;

RT_PROGRAM void 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 );
  float3 color = Ka * ambient_light_color;

  float t_hit = incoming_ray_t.get();
  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.result = color;
}
```

Lambertian shader - result

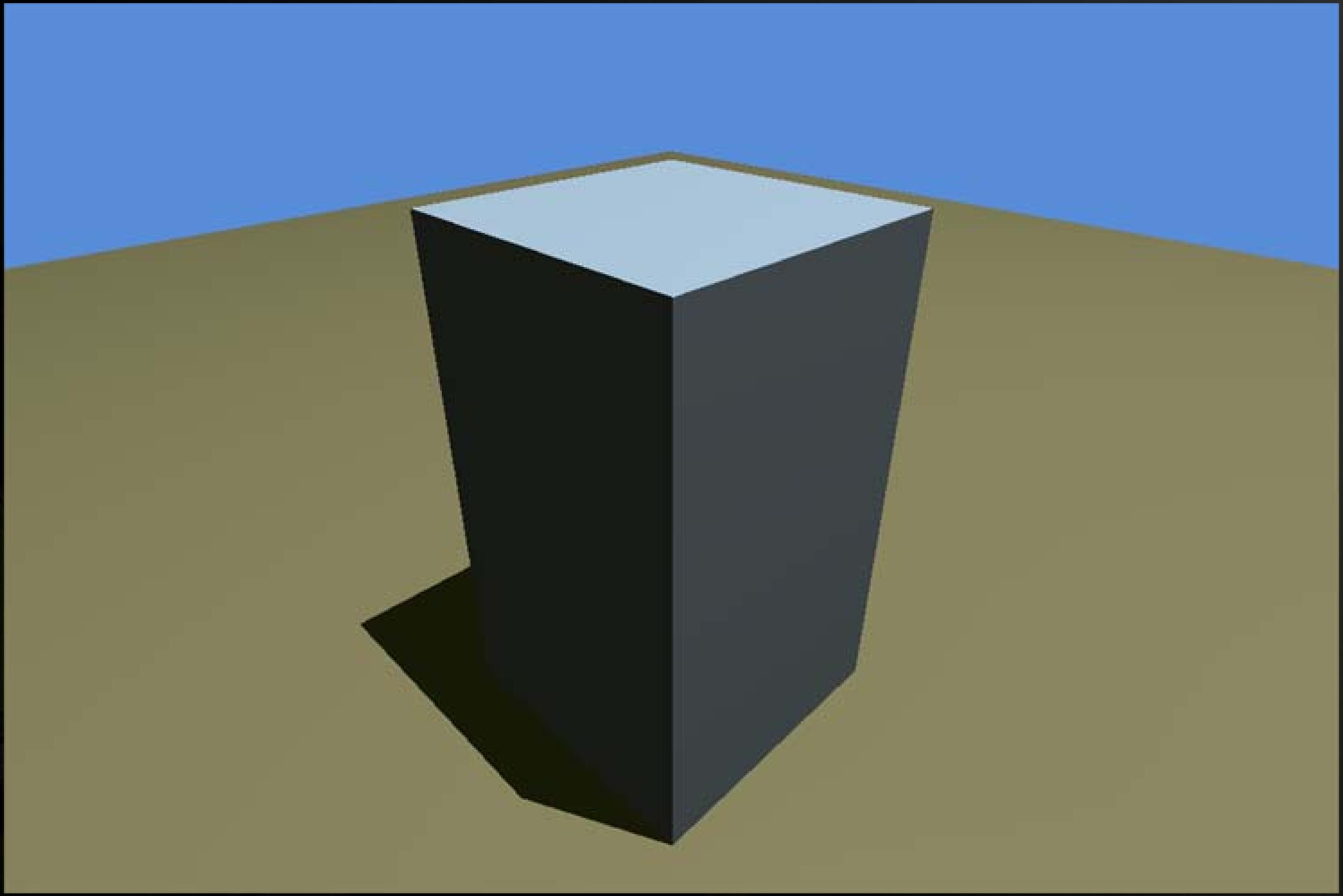


Adding shadows - goal

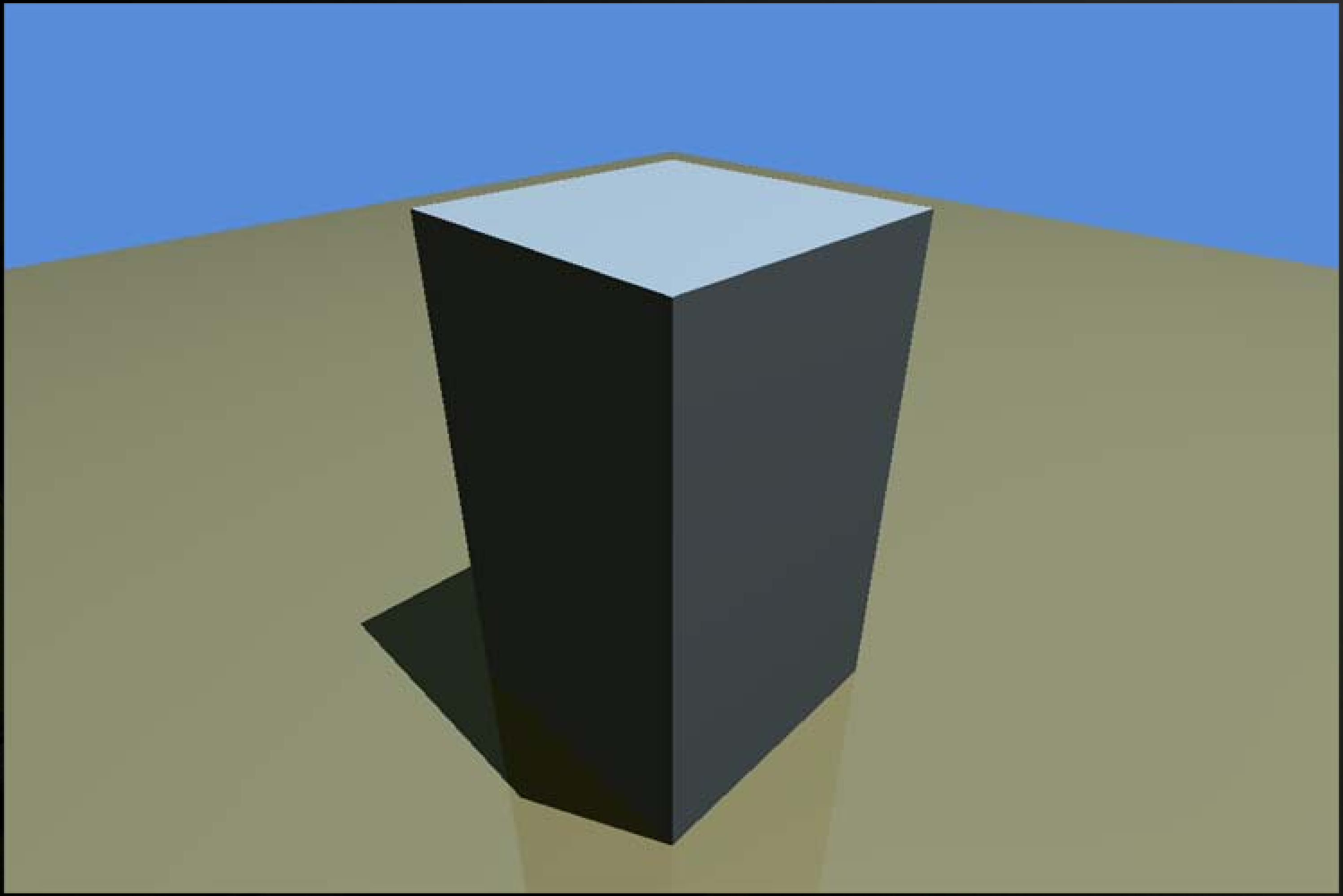


```
for(int i = 0; i < lights.size(); ++i) {
  BasicLight light = lights[i];
  float3 L = normalize(light.pos - hit_point);
  float nDl = dot( ffnormal, L);

  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;

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

  }
```

Adding shadows -result



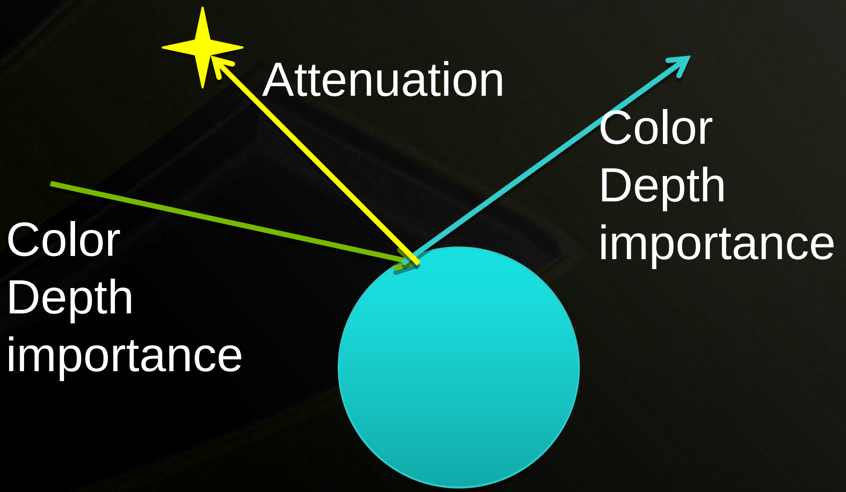
Adding reflections - goal



```
 …
 // reflection ray
PerRayData_radiance refl_prd;
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;
```

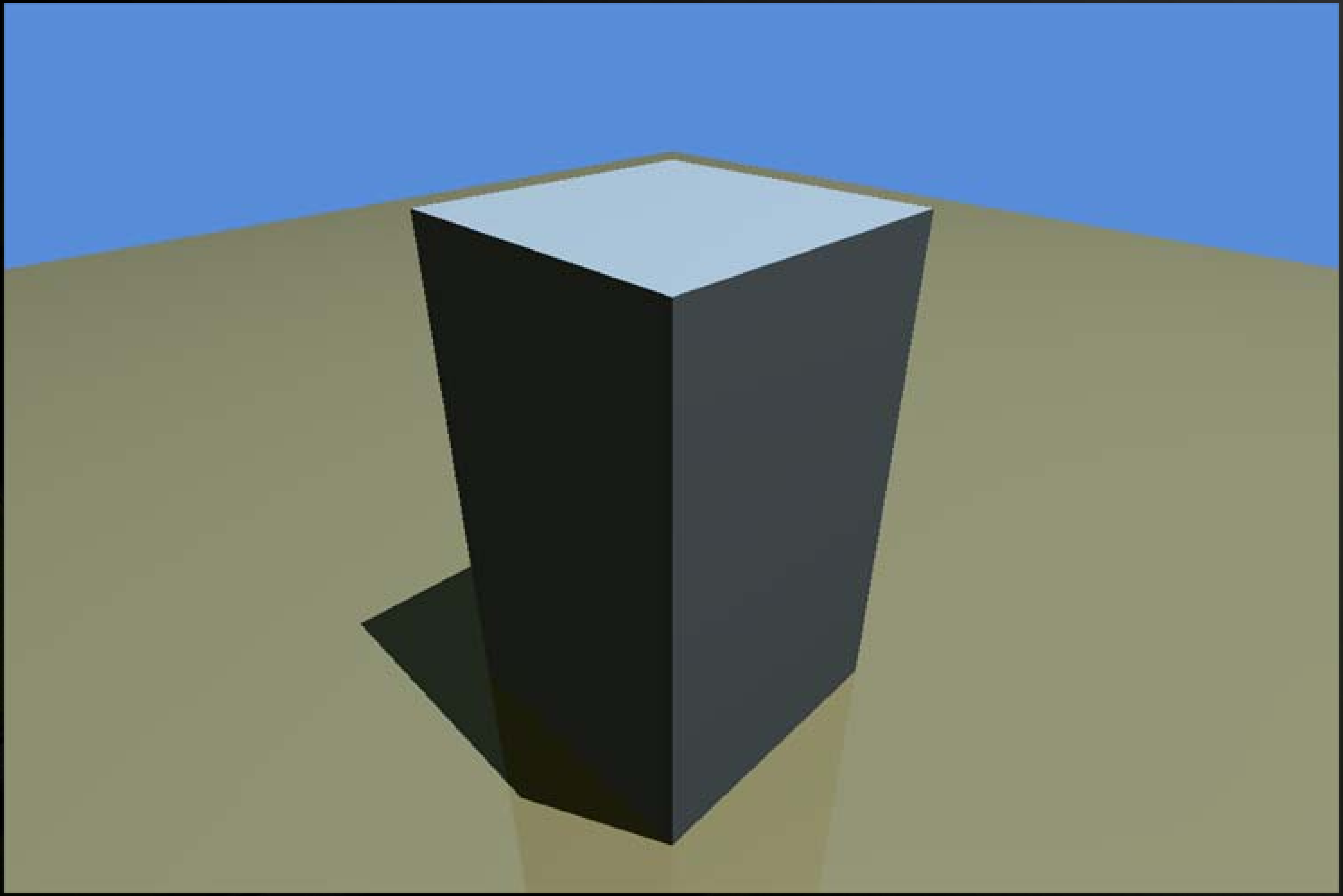
Per ray data

- Can define arbitrary data with the ray
- Sometimes called the "payload of the ray"
- 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

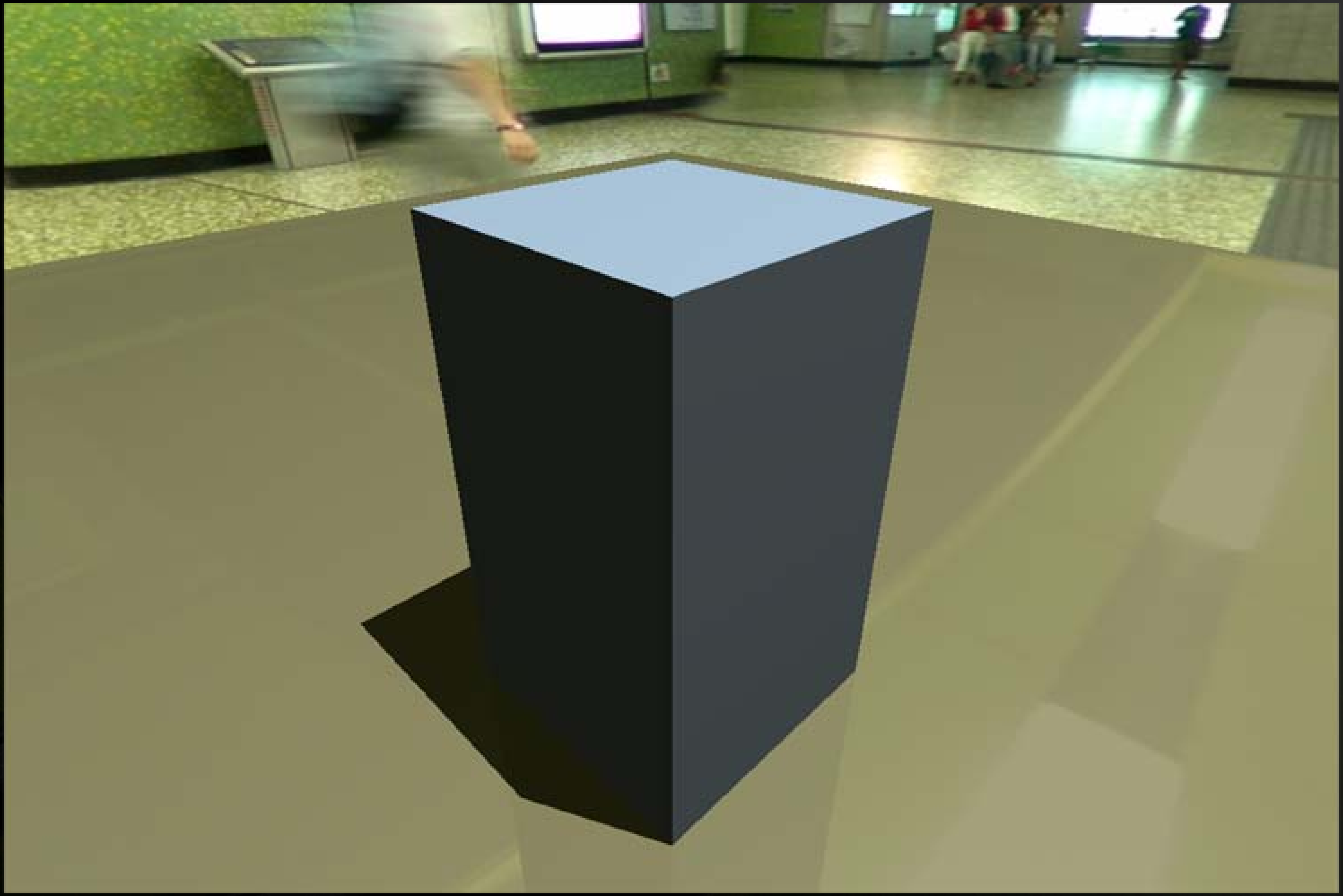


```
  struct PerRayData_radiance
{
  float3 result;
  float  importance;
  int depth;
};

…
  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 - result

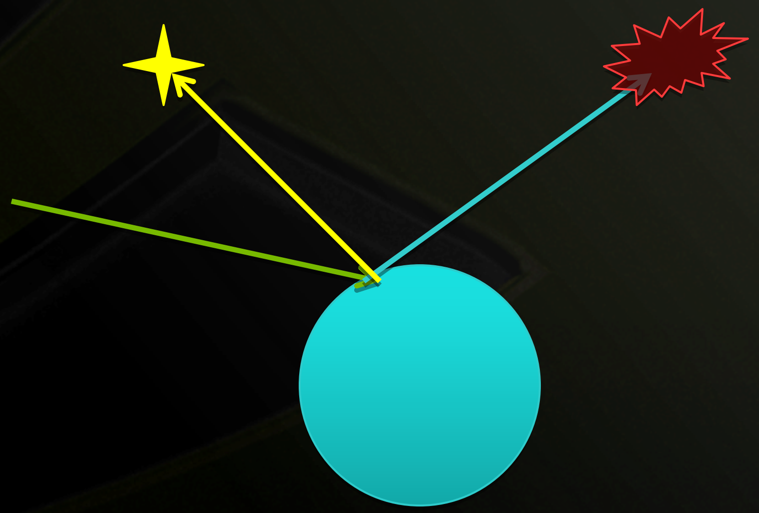


Environment map - goal



Miss program

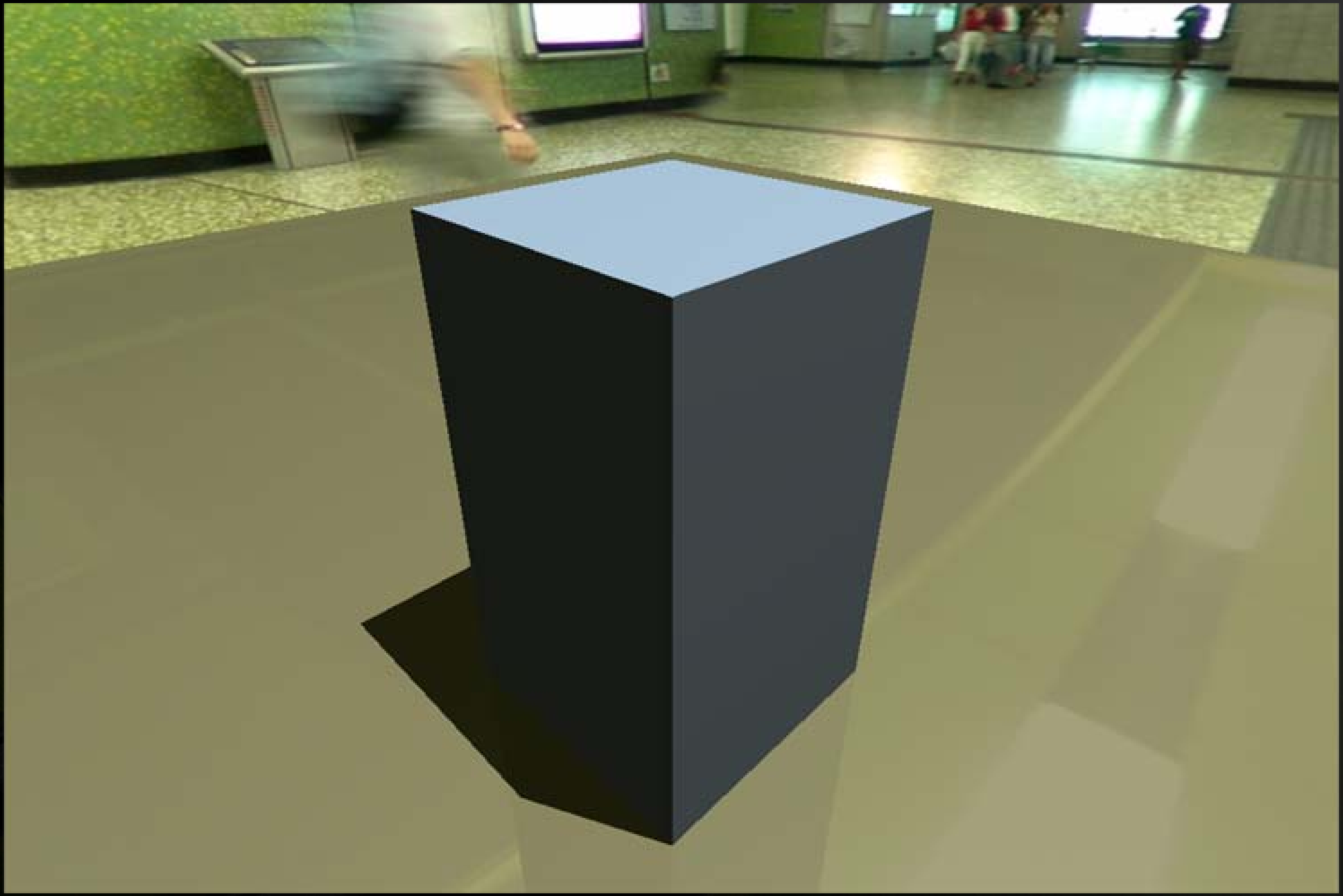
- Defines what happens when a ray misses all objects
- Accesses per-ray data
- Usually – background color



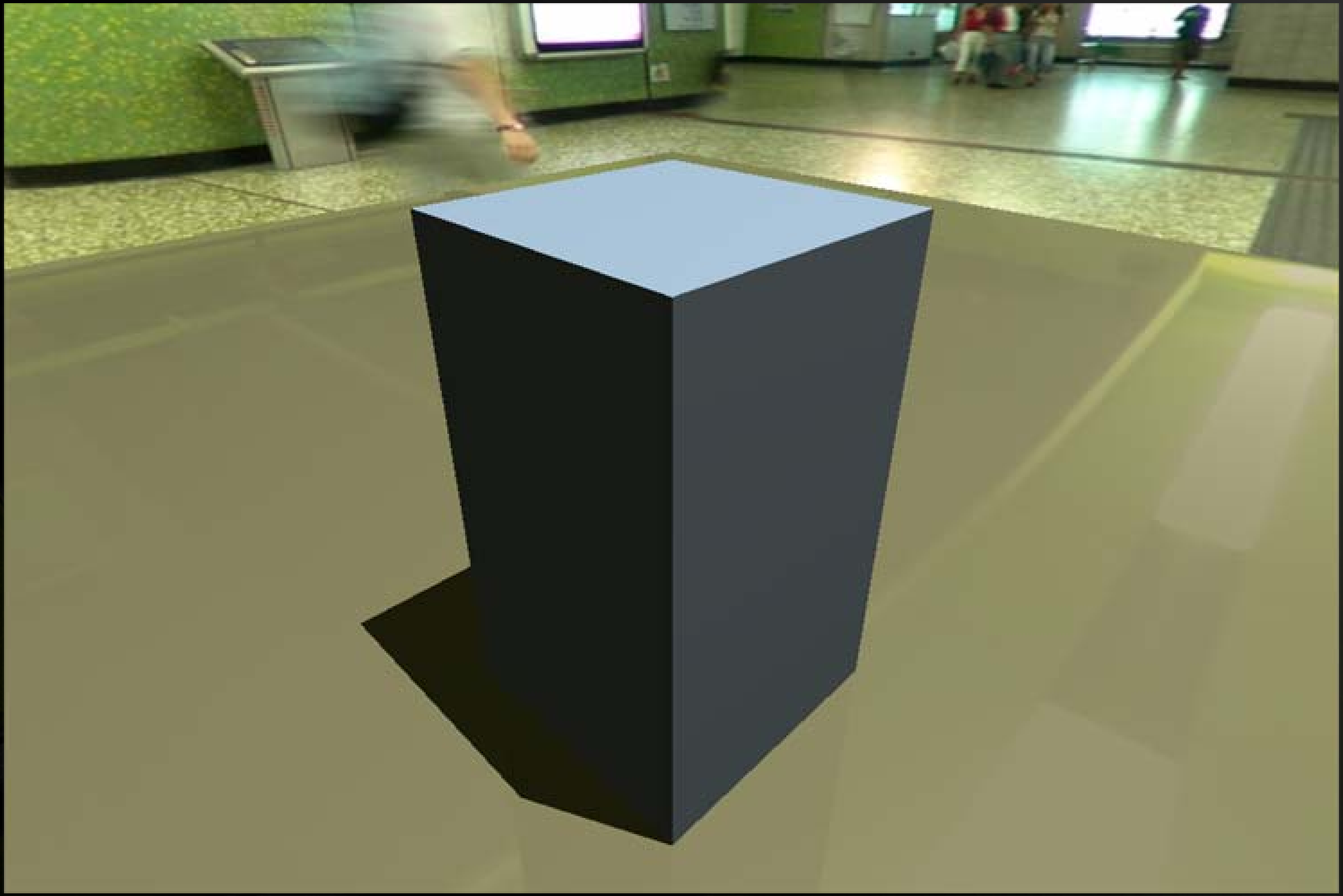
```
rtDeclareVariable(float3, bg_color);
RT_PROGRAM void miss()
{
  PerRayData_radiance& prd = prd_radiance.reference();
  prd.result = bg_color;
}
```

```
rtTextureSampler<uchar4, 2, cudaReadModeNormalizedFloat> envmap;

RT_PROGRAM void miss()
{
  const Ray ray = incoming_ray.get();
  PerRayData_radiance& prd = prd_radiance.reference();

  float theta = atan2f(ray.direction.x, ray.direction.z);
  theta = (theta + M_PIf) * (0.5f * M_1_PIf);
  float phi = ray.direction.y * 0.5f + 0.5f;
  prd.result = make_float3(tex2D(envmap, theta, phi));
}
```

Environment map - result



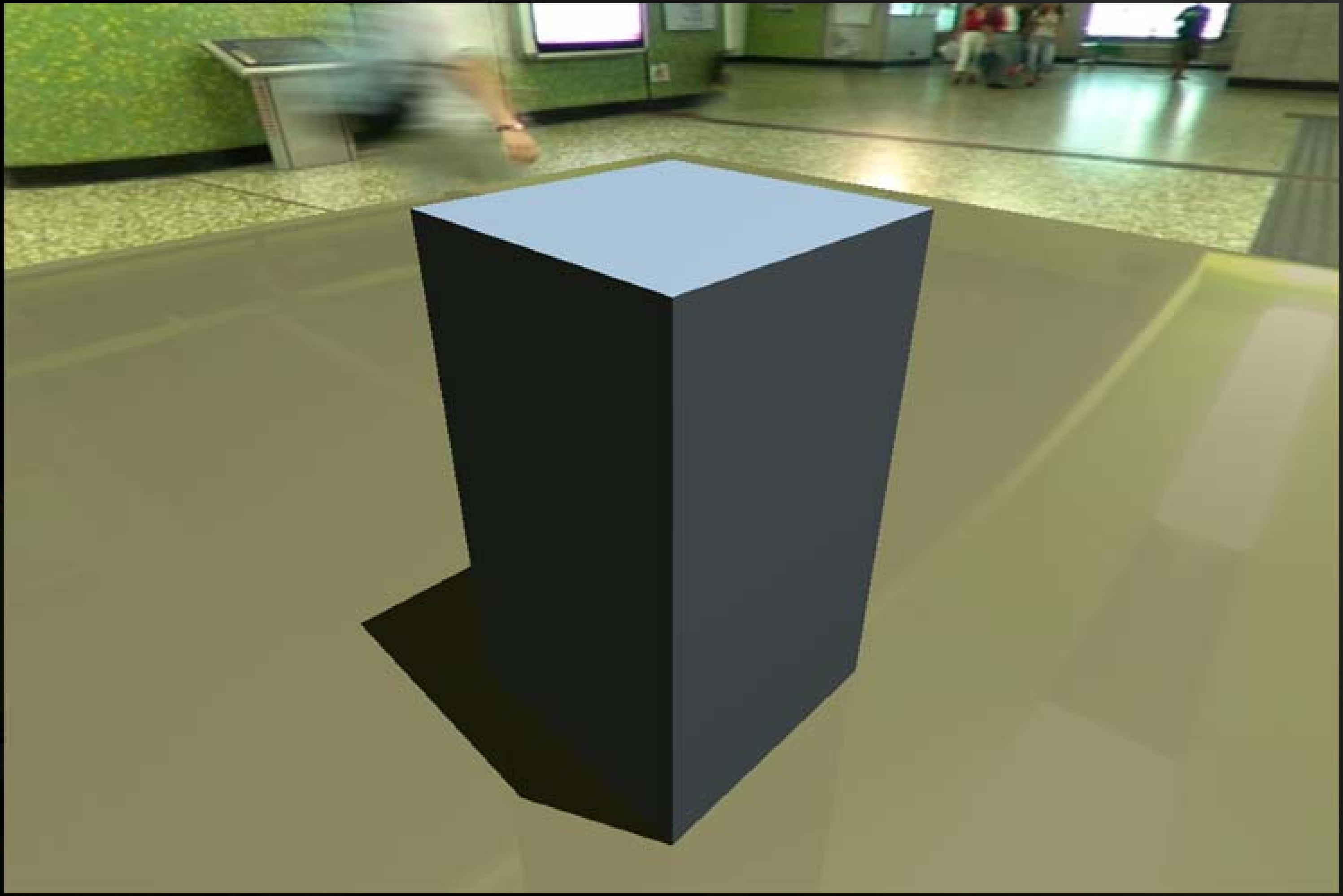
Schlick approximation - goal



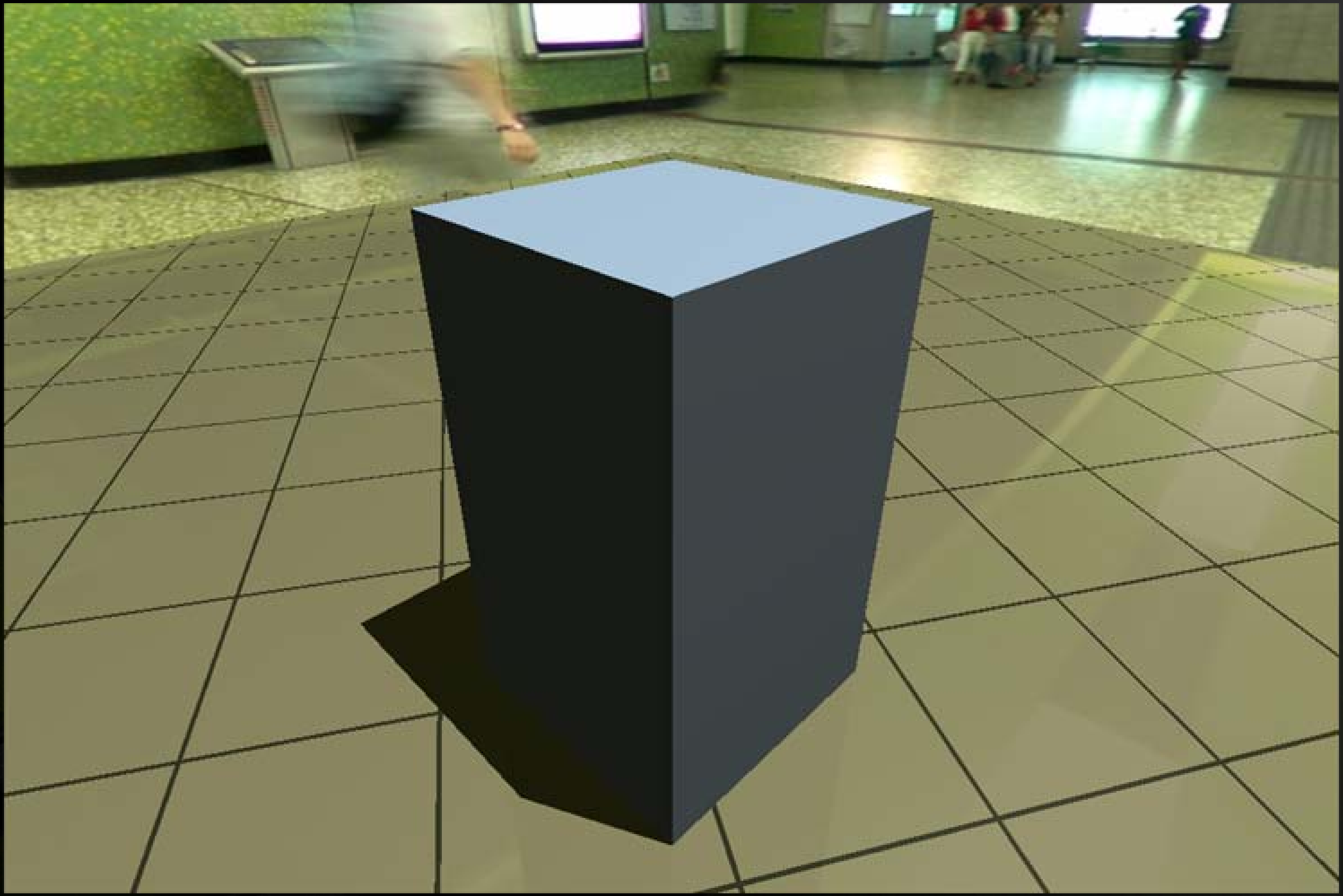
```
float3 r = schlick(-dot(ffnormal, ray.direction), reflectivity_n);
float importance = prd.importance * luminance( r );
```

```
// 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 += reflectivity * refl_prd.result;
}
```

Schlick approximation - result

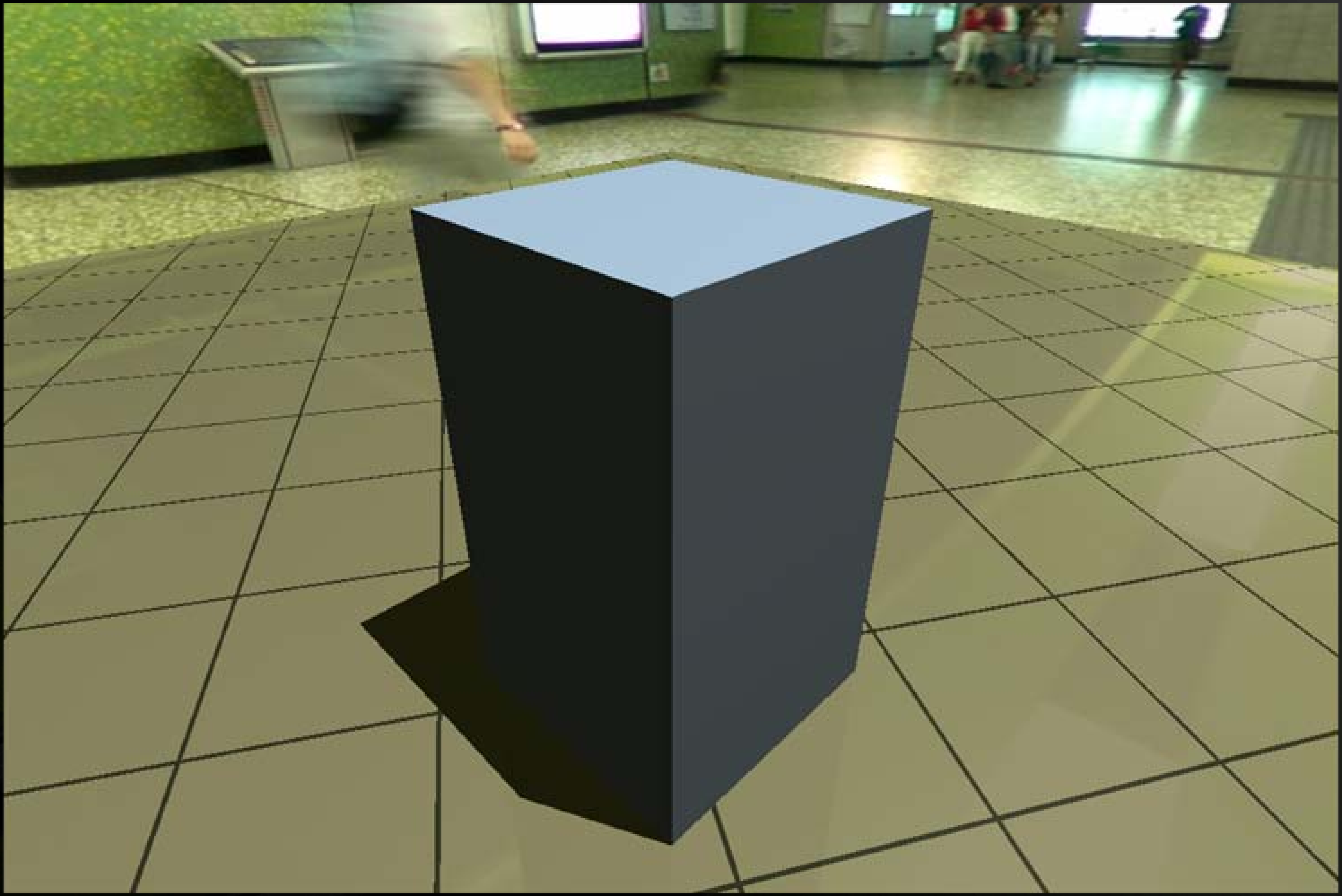


Procedurally tiled floor - goal

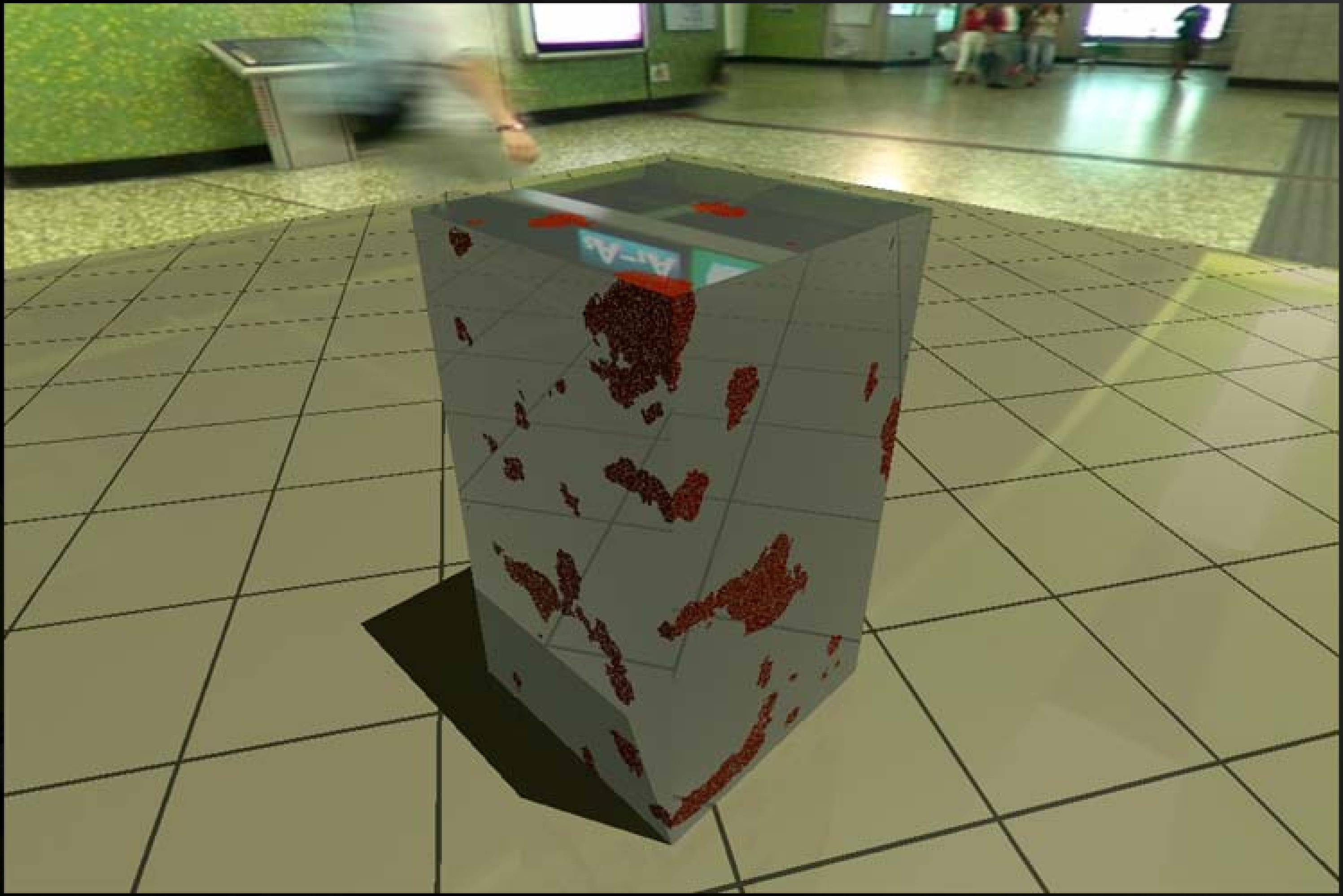


```
…
float t_hit = incoming_ray_t.get();
float3 hit_point = ray.origin + t_hit * ray.direction;

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;
}
…
```

Procedurally tiled floor - result



Rusty metal procedural - goal

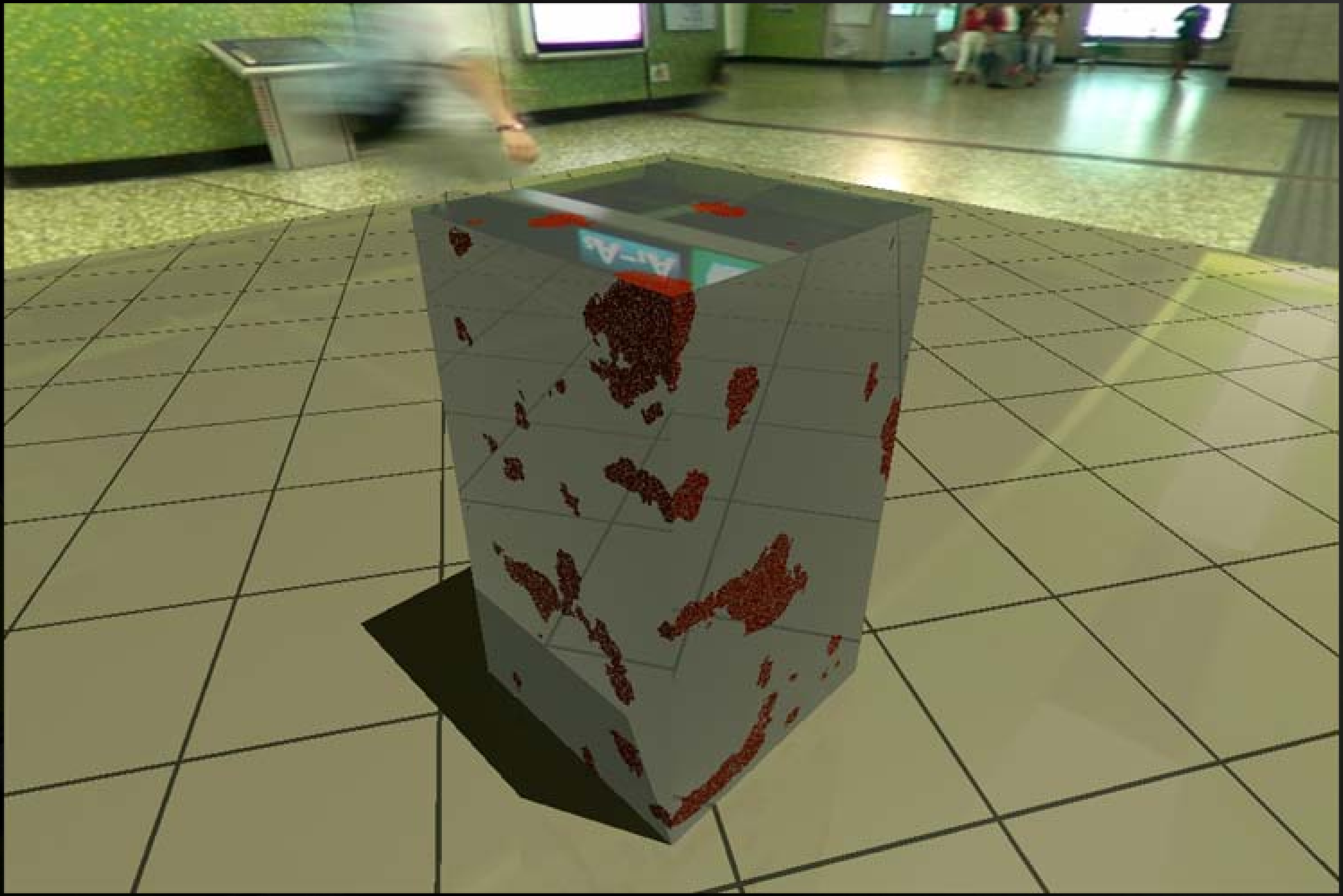


```
rtDeclareVariable(float, metalKa) = 1;
rtDeclareVariable(float, metalKs) = 1;
rtDeclareVariable(float, metalroughness) = .1;
rtDeclareVariable(float, rustKa) = 1;
rtDeclareVariable(float, rustKd) = 1;
rtDeclareVariable(float3, rustcolor) = {.437, .084, 0};
rtDeclareVariable(float3, metalcolor) = {.7, .7, .7};
rtDeclareVariable(float, txtscale) = .02;
rtDeclareVariable(float, rusty) = 0.2;
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;

  /* Sum several octaves of abs(snoise), i.e. turbulence.  Limit the
   * 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) / 0.08f;
  rustiness *= rustiness;

  /* If we have any rust, calculate the color of the rust, taking into
   * account the perturbed normal and shading like matte.
   */
  float3 Nrust = ffnormal;
  if (rustiness > 0) {
    /* If it's rusty, also add a high frequency bumpiness to the normal */
    Nrust = normalize(ffnormal + rustbump * snoise(PP));
    Nrust = faceforward (Nrust, -ray.direction, world_geo_normal);
```

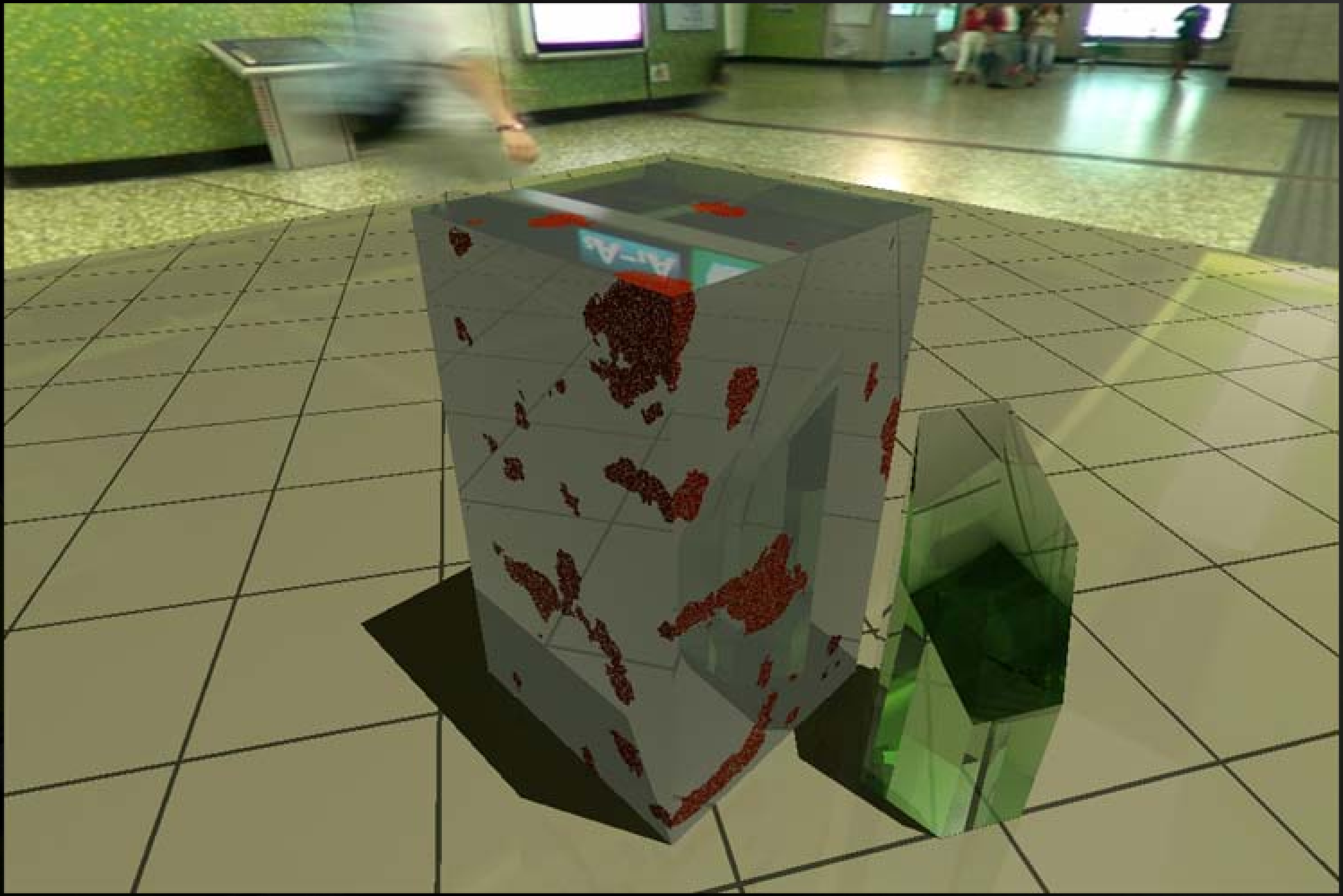
```
  float3 color = mix(metalcolor * metalKa, rustcolor * rustKa, rustiness) *
       ambient_light_color;
  for(int i = 0; i < lights.size(); ++i) {
    BasicLight light = lights[i];
    float3 L = normalize(light.pos - hit_point);
    float nmDl = dot( ffnormal, L);
    float nrDl = dot( Nrust, L);

    if( nmDl > 0.0f || nrDl > 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;
        nrDl = max(nrDl * rustiness, 0.0f);
        color += rustKd * rustcolor * nrDl * Lc;

        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 * metalKs * Lc * pow(nmDh, 1.f/metalroughness);
        }
      }

    }
  }

  float3 r = schlick(-dot(ffnormal, ray.direction), reflectivity_n * (1-rustiness));
  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;
  }

  prd.result = color;
}
```

Direct port of rusty metal BMRT shader by Larry Gritz

Rusty metal procedural - result

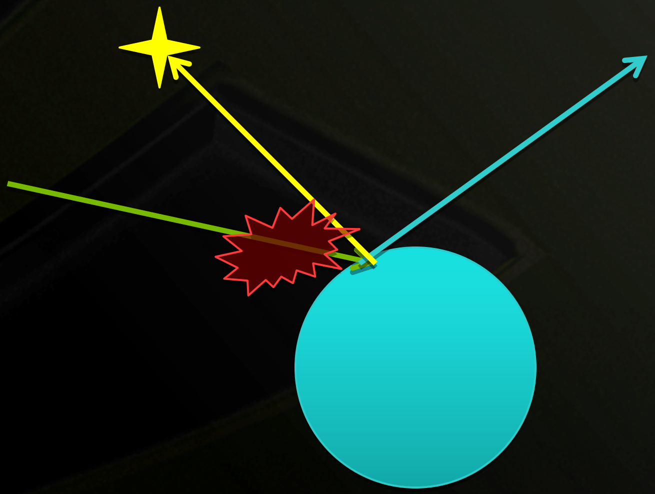


Adding procedural primitives - goal



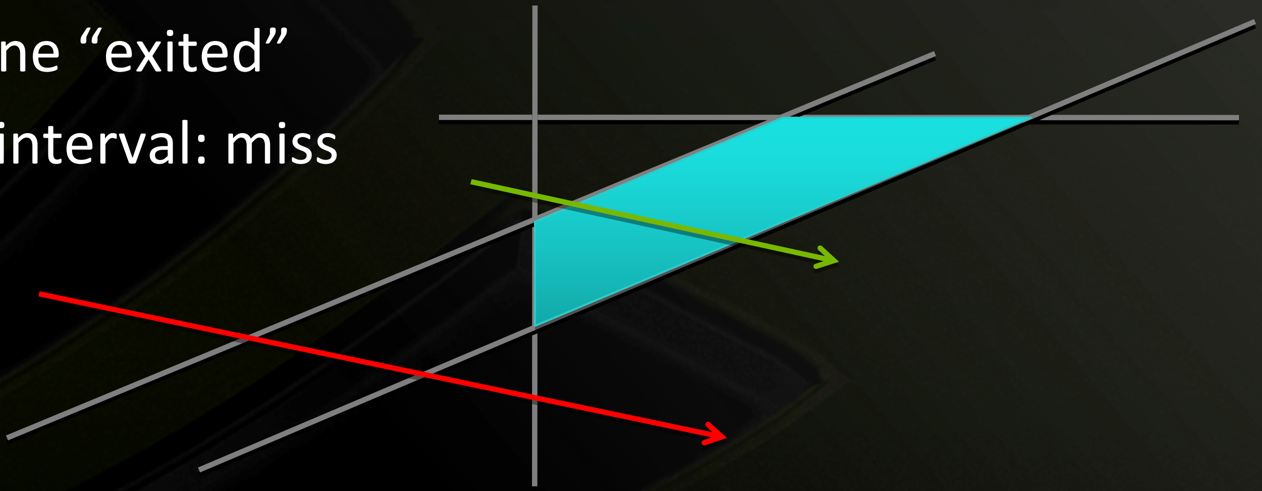
Intersection program

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



Convex hull object

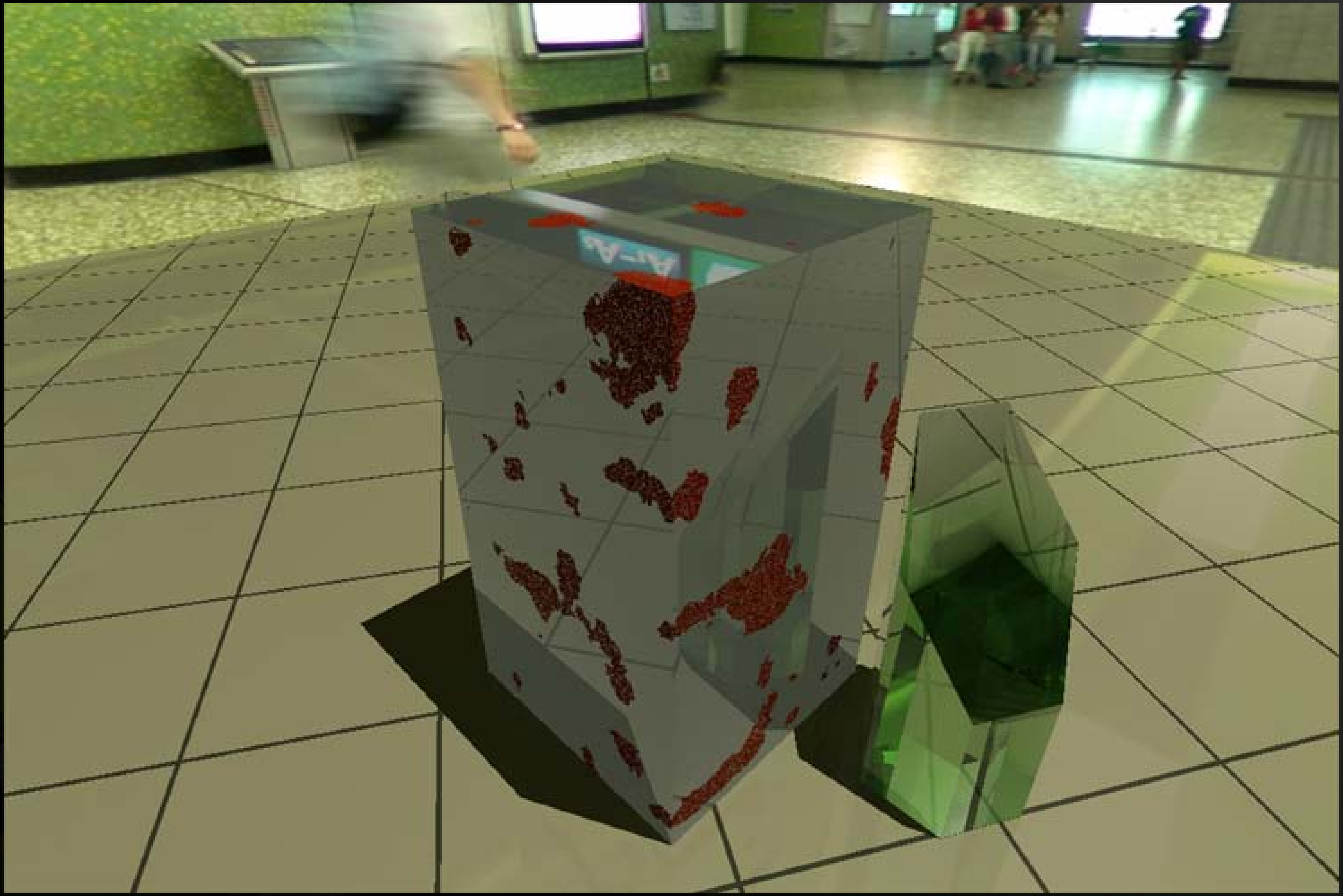
- Defined by a set of planes
- Created by the host
- Simple algorithm can handle any number of planes
    - Find last plane “entered”
    - Find first plane “exited”
    - Degenerate interval: miss



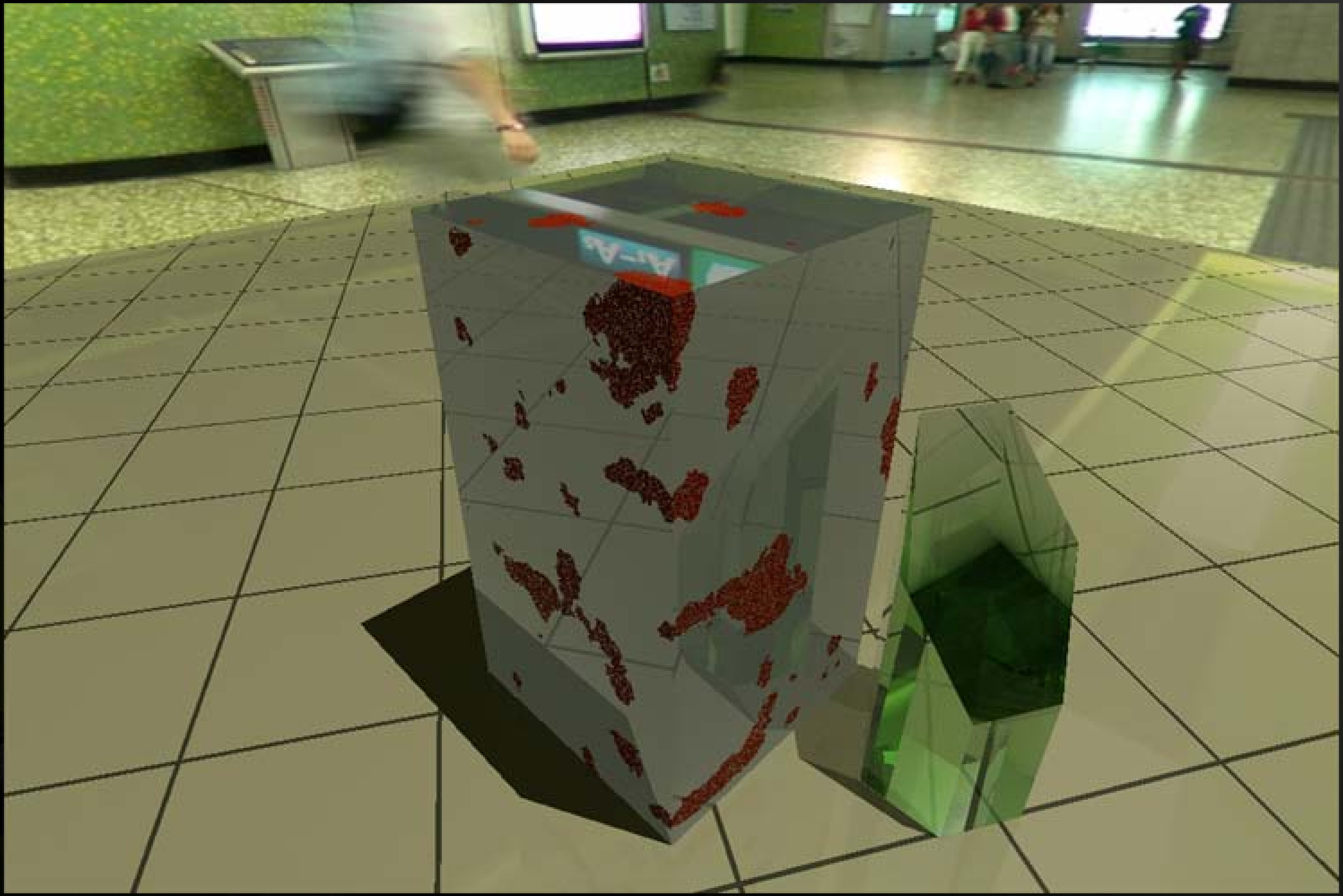
```
rtBuffer<float4> planes;
RT_PROGRAM void chull_intersect(int primIdx)
{
  const Ray ray = incoming_ray.get();

  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;
      }
    } else {
      //exit
      if(t < t1){
        t1 = t;
        t1_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);
  }
}
```

Adding procedural primitives - result

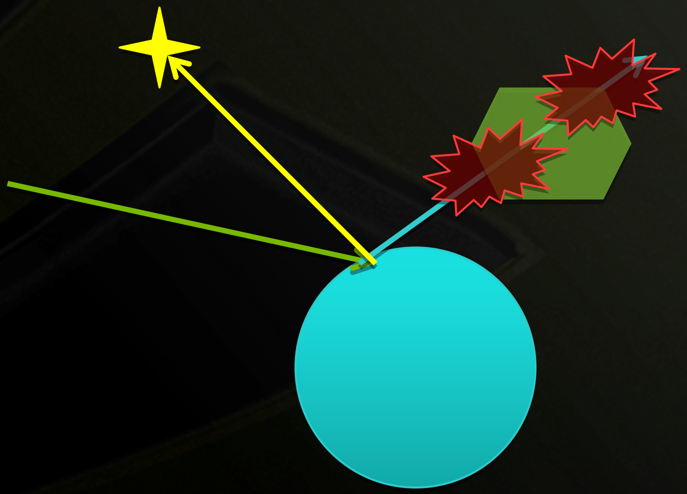


Tweaking the shadow - goal



Any hit program

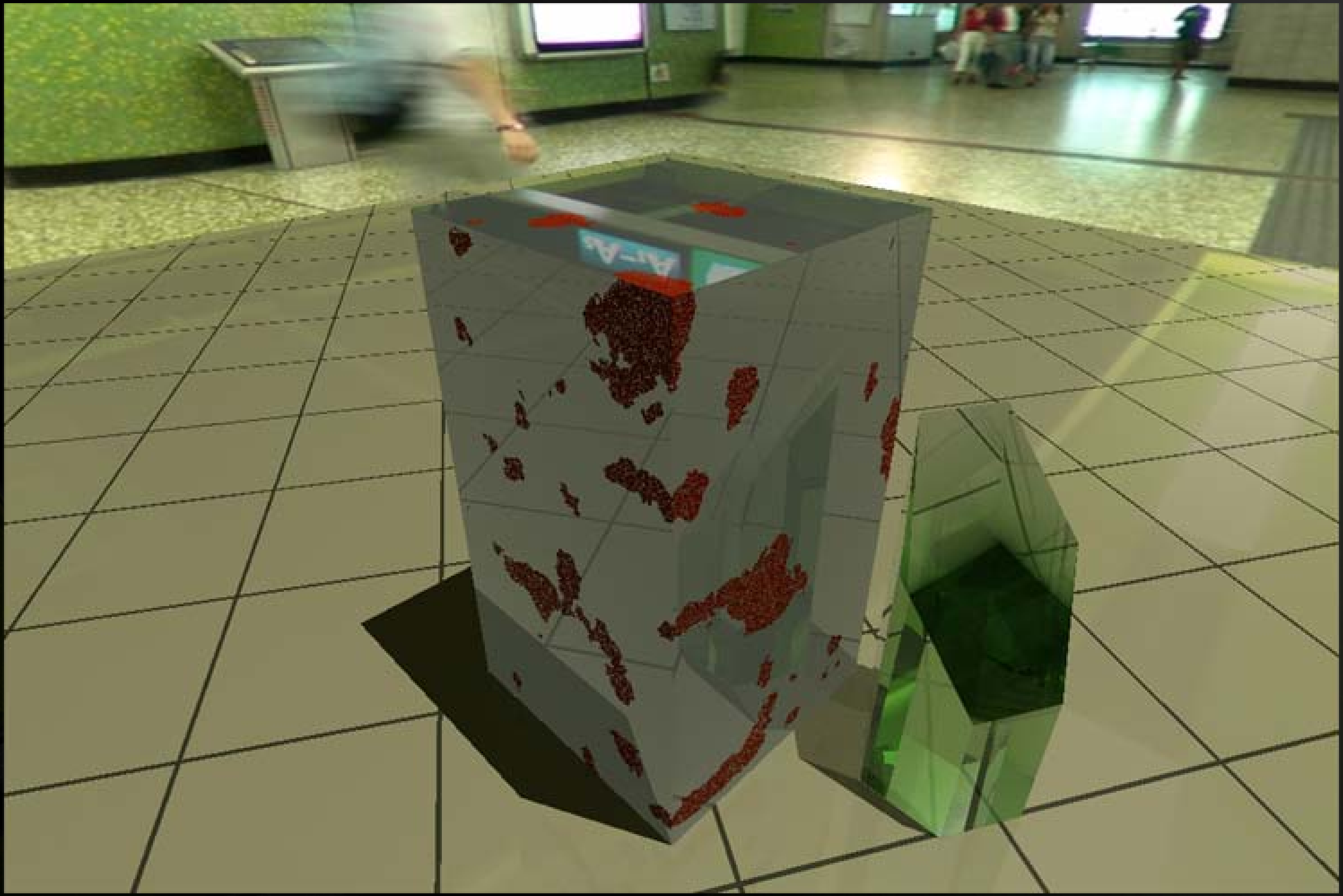
- Defines what happens when a ray attempts to hit an object
- Executed for all intersections along a ray
- Can optionally:
  - Stop the ray immediately (shadow rays)
  - Ignore the intersection and allow ray to continue (alpha transparency)



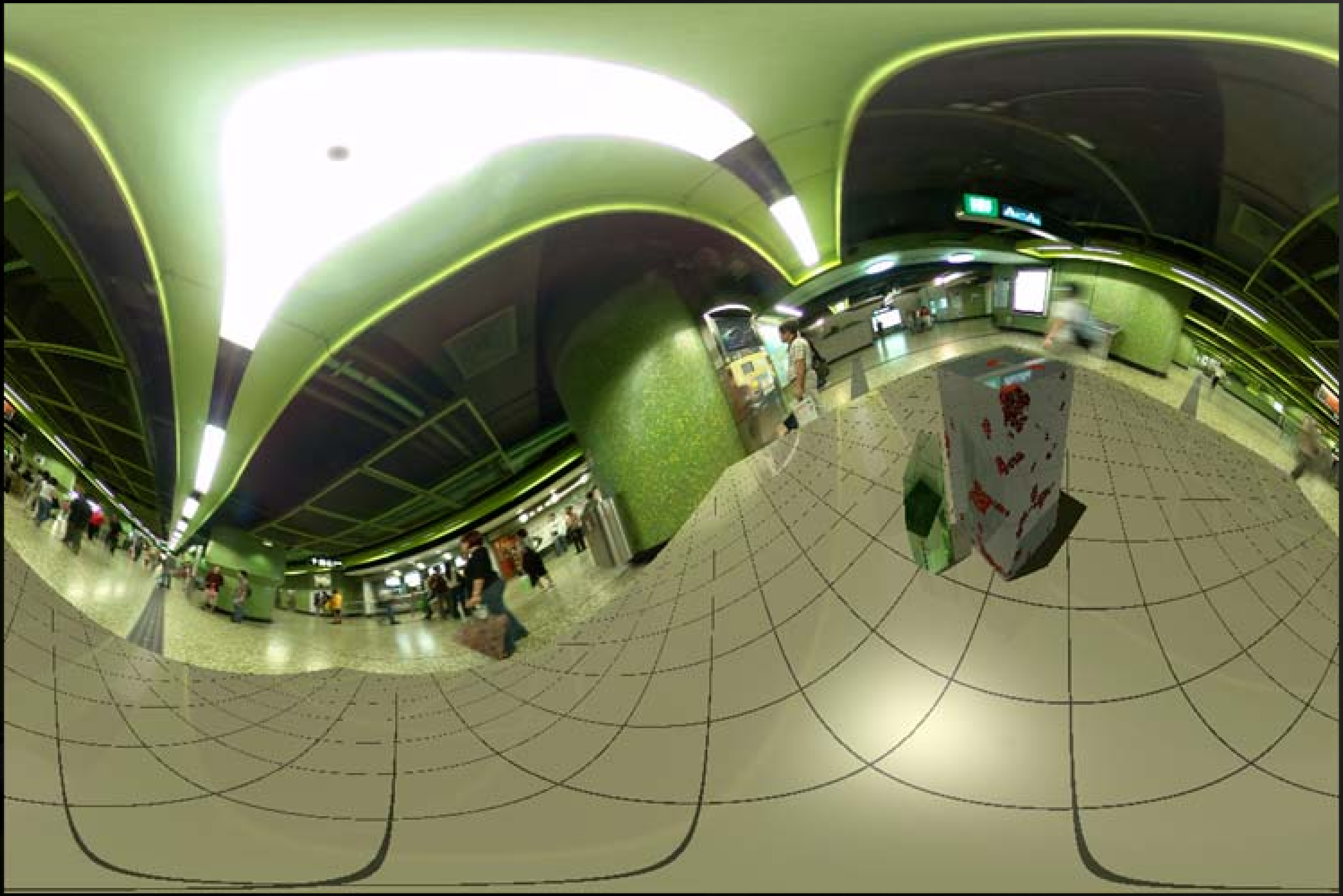
```
RT_PROGRAM void any_hit_shadow()
{
  // this material is opaque, so it fully attenuates all shadow rays
  PerRayData_shadow& prd = prd_shadow.reference();
  prd.attenuation = 0;

  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));

  PerRayData_shadow& prd = prd_shadow.reference();
  prd.attenuation *= 1-fresnel_schlick(nDi, 5, 1-shadow_attenuation, 1);

  rtIgnoreIntersection();
}
```

Tweaking the shadow - result

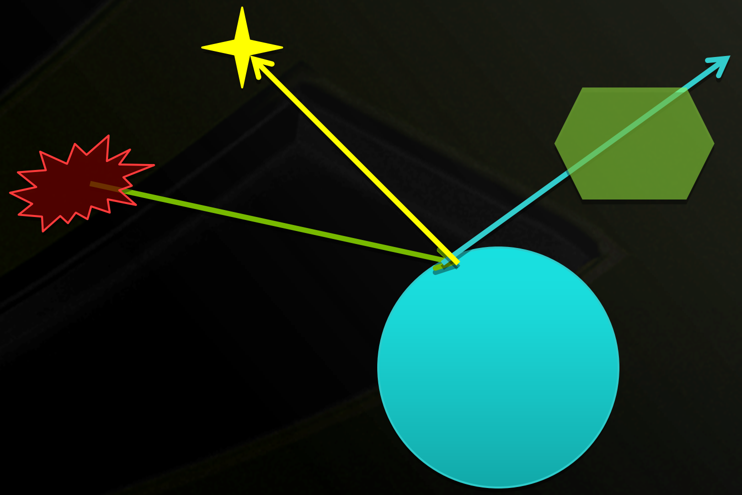


Environment map camera - goal



Ray generation program

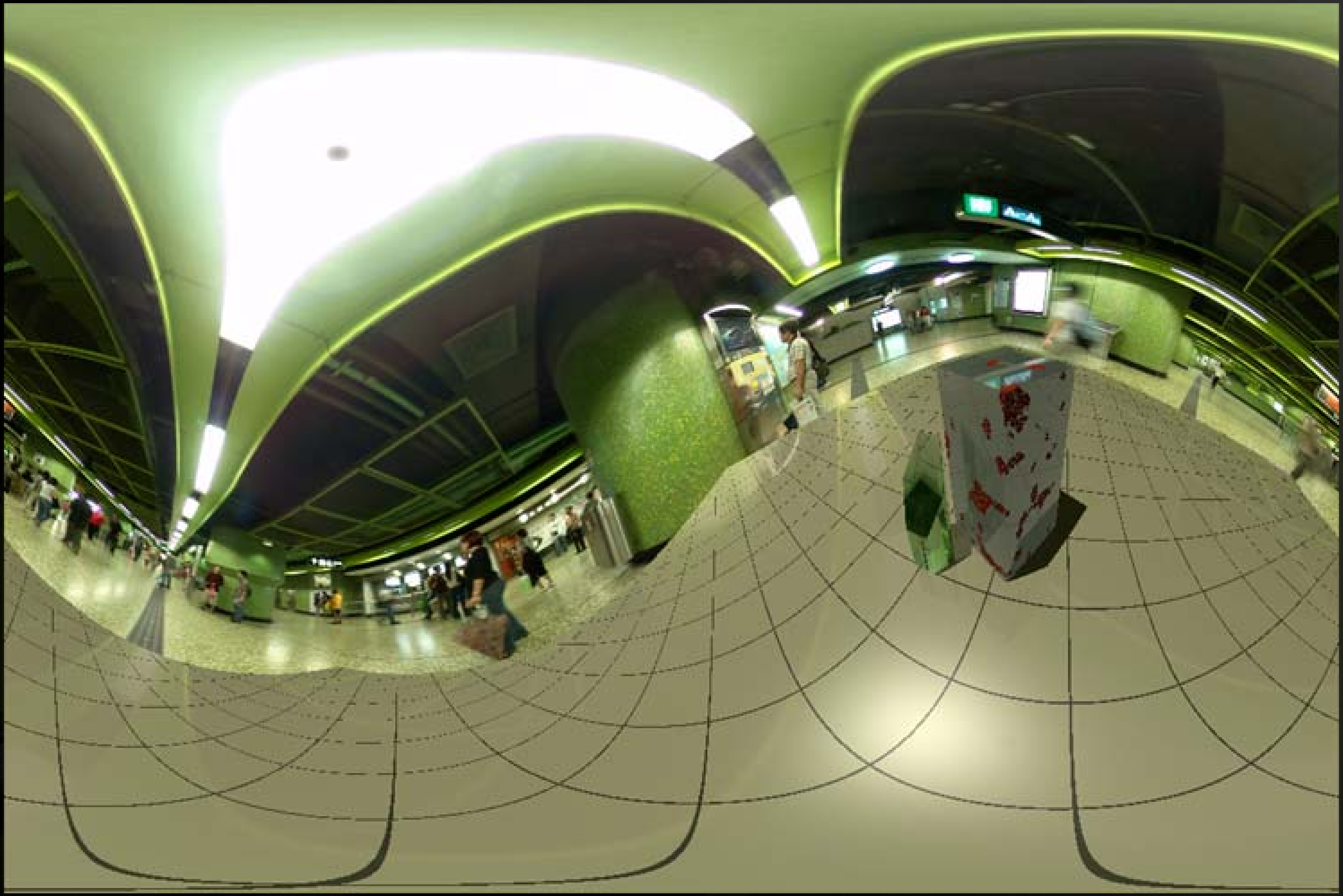
- Starts the ray tracing process
- Used for:
  - Camera model
  - Output buffer writes
- Can trace multiple rays
- Or no rays



```
rtDeclareSemanticVariable(rtRayIndex, rayIndex);

RT_PROGRAM void pinhole_camera()
{
  uint2 screen = output_buffer.size();
  uint2 index = make_uint2(rayIndex.get());

  float2 d = make_float2(index) / make_float2(screen) * 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()
{
  uint2 screen = output_buffer.size();
  uint2 index = make_uint2(rayIndex.get());

  float2 d = make_float2(index) / make_float2(screen);
  d = d * make_float2(2.0f * M_PIf , M_PIf) + make_float2(M_PIf, 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 );
}
```

Environment map camera - result



Next steps one could take

- Multiple rays per pixel (raygen program)
- Image-based lighting (closest hit program)
- Ambient occlusion (closest hit program)
- Path tracer (new shaders, raygen program)
- Interaction with host code

Additional OptiX features

- Powerful object model
  - All objects green except one
  - Different light source list for a single object
- Can use double precision arithmetic
- OptiX node-graph
  - Programmable traversal
  - Dynamic
  - Built-in acceleration structures
    - BVH, SBVH, kd-tree
  - Supports dynamic scenes
- Multiple “entry points”
  - Adapative AA
  - photon pass, gather pass
- Interop with OpenGL
  - Textures, VBOs

OptiX engine - availability

- Freely available to registered developers to both use & deploy
- in early fall, 2009 from http://www.nvidia.com

Questions?

sparker@nvidia.com

http://www.nvidia.com