Nvidia OptiX

What is OptiX?

- General Purpose Ray Tracing API
  - Rendering, baking, collision detection, etc.
  - Shader-centric, stateless and bindless
  - Not a renderer itself, used to make renderers
- Highly Programmable
  - Shade as you like
  - Ray generation/framebuffer operations
  - Programmable intersections
- Easy to Program
  - Write code for a single ray
  - Usable on any modern Nvidia GPU

Getting OptiX

Obtain OptiX here:
http://developer.nvidia.com/object/optix-home.html

CUDA here:
http://developer.nvidia.com/object/cuda_3_2_downloads.html

Programmable parts of OptiX

- Closest Hit
- Any hit
- Intersection and Selection
- Ray Generation
- Misses
- Exceptions

Shaders

Programs for closest hit/any hit and everything else is called a shader

- Written in CUDA C/C++
  - Templates
  - Pointers
  - Overloading
  - Classes (no virtual yet)
- Shaders chained together define the overall operation of the raytracer

Hit Shaders

Closest Hit

- Called after the closest hit has been found
- Used for traditional surface shading
- Can also be used for deferred shading

Any Hit

- Called during traversal for each potentially closest hit
- Can be used for things such as discarding hits on transparent parts of an object through texturing
- Can be used to end shadow rays with any hit

Parts of a Shader

- Includes for utilities and definitions
- Variable declarations
  - Textures
  - Buffers
  - Read-only shader variables
- Multiple Shader programs

Example: Closest Hit

- Defines what happens when a ray hits an object
- OptiX runs it for the closest intersection along a ray
- Can automatically perform deferred shading
- Recursive Ray Generation
  - Shadow Rays
  - Reflections
  - Ambient Occlusion
- Most common shader

Example: Closest Hit - Show Normals

```
struct PerRayData_radiance {
    float3 result;
}

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

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

Example: Miss Shader

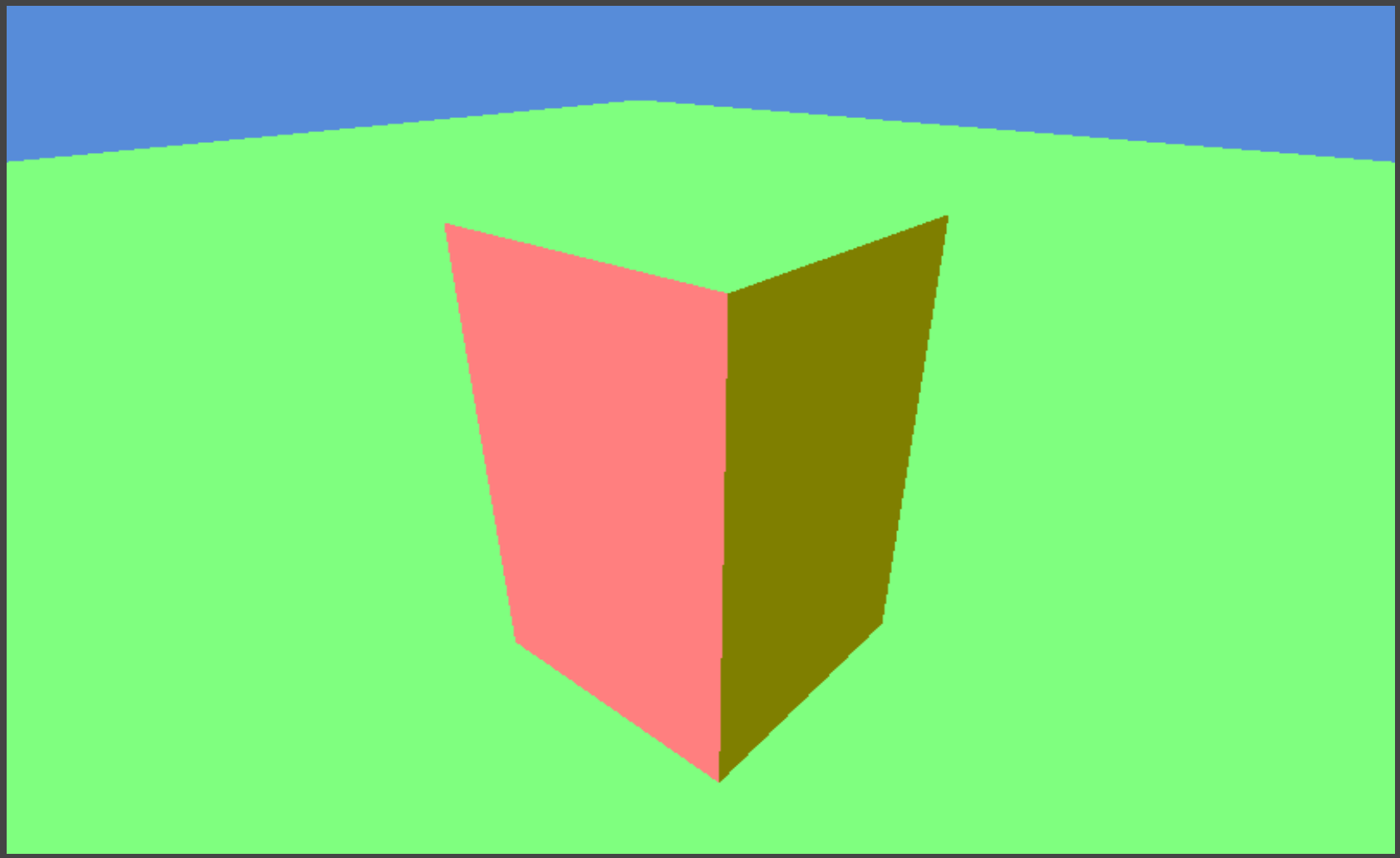
- If the ray doesn't hit anything, what do you do?
- Can be used for environment mapping
- For now, let's just show a background color

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

Generating the Rays

```
rtDeclareVariable(float3,        eye, , );
rtDeclareVariable(float3,        U, , );
rtDeclareVariable(float3,        V, , );
rtDeclareVariable(float3,        W, , );
rtDeclareVariable(float3,        bad_color, , );
rtBuffer<uchar4, 2>              output_buffer;

RT_PROGRAM void pinhole_camera()
{
  uint2 screen = output_buffer.size();

  float2 d = make_float2(launch_index) / make_float2(screen) * 2.f - 1.f;
  float3 ray_origin = eye;
  float3 ray_direction = normalize(d.x*U + d.y*V + W);

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

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

  rtTrace(top_object, ray, prd);

  output_buffer[launch_index] = make_color( prd.result );
}
```

Result from Normal Shader



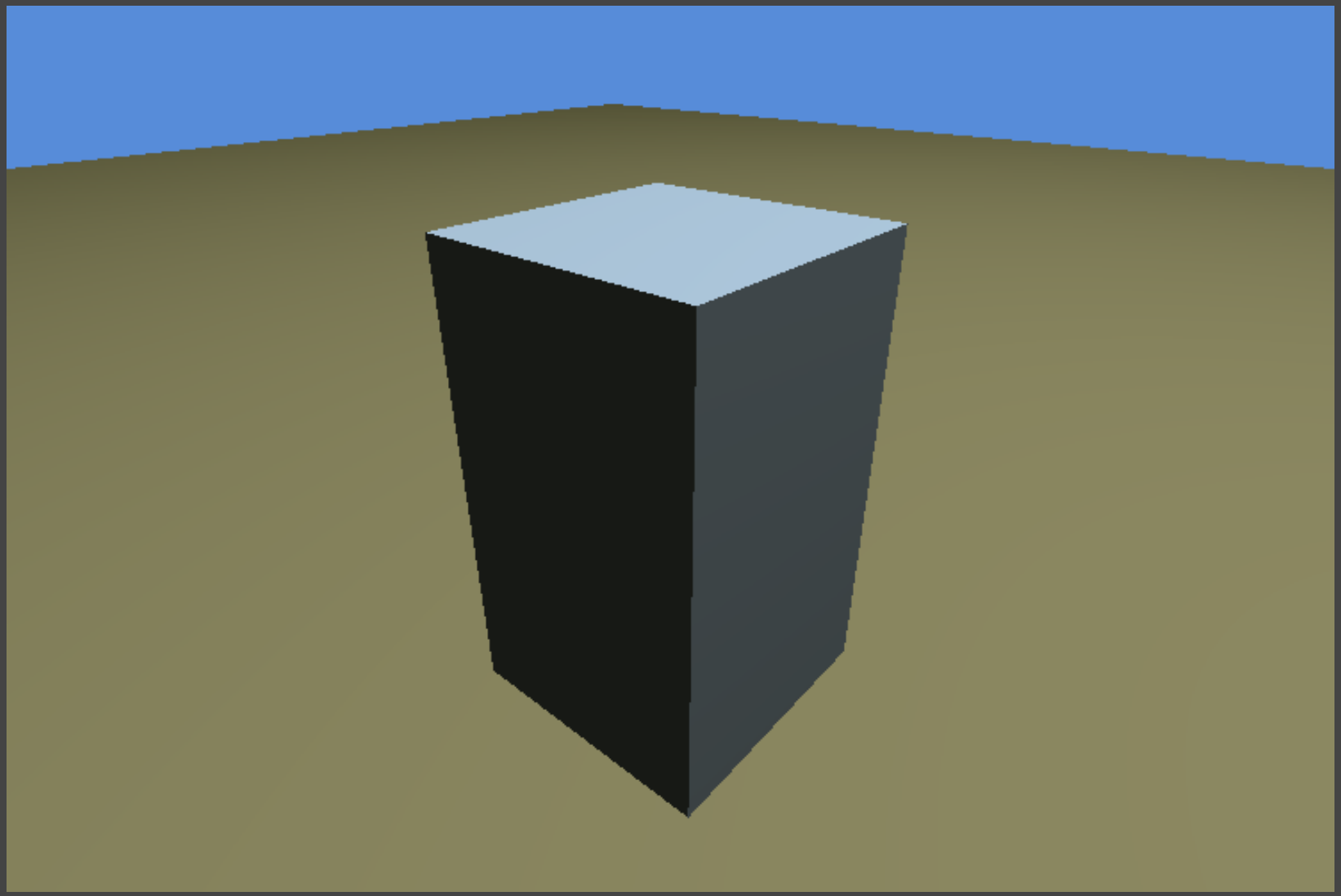
Diffuse Shading

- Uses our normals from the previous normal shader
- Iterate through all the lights
  - Take the cosine of the angle between the normal and the light direction
  - Use this to weight the contribution from the light
- Accumulate the contributions for the final color

Diffuse Shading

```
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) {
    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;
}
```

Result from Diffuse Shader



Shadows

- Modify the loop in which we do diffuse shading
- Before adding the contribution of the light, we need to make sure the light is not occluded
- Generate a shadow ray, and check to see if anything is in its way to the light
- If something is in the way, attenuate the contribution from this light

Shadows

```
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);
      }

    }
```

Per Ray Data

- Add arbitrary data with the ray
- Also called the "payload"
- Can be passed up and down the ray tree
- Simple user defined struct
- Different Per ray type

For shadows, defines the total amount of attenuation:

```
struct PerRayData_shadow
{
  float3 attenuation;
};
```

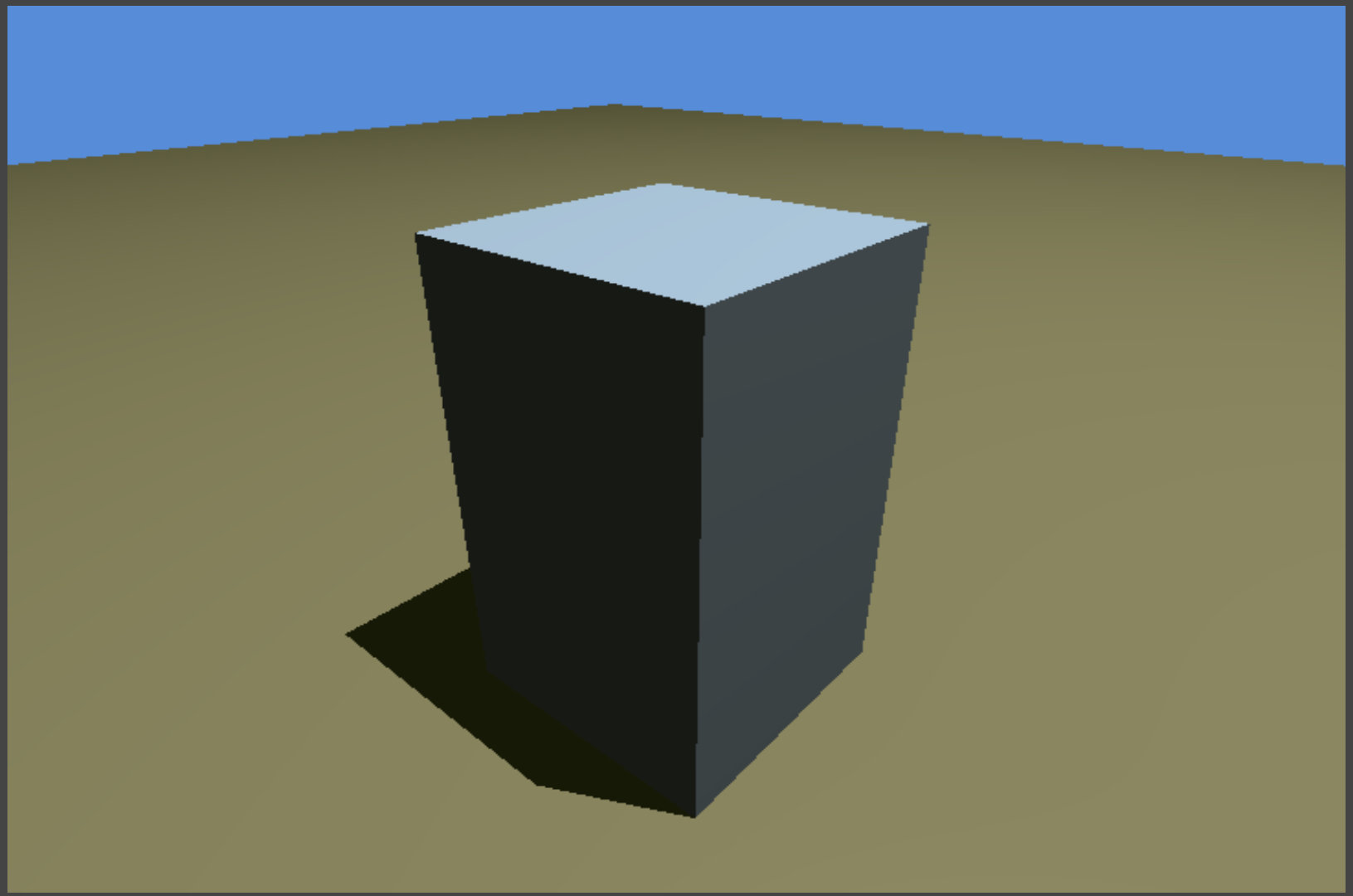
Any Hit Shaders

- Shadows do not need to use the closest hit
- Instead, shadows can use an "any hit" shader

Simple "any hit" shader for shadows:

```
RT_PROGRAM void any_hit_shadow()
{
  // this material is opaque
  // so it fully attenuates all shadow rays
  prd_shadow.attenuation = make_float3(0);

  rtTerminateRay();
}
```

Shadows Result



Reflections

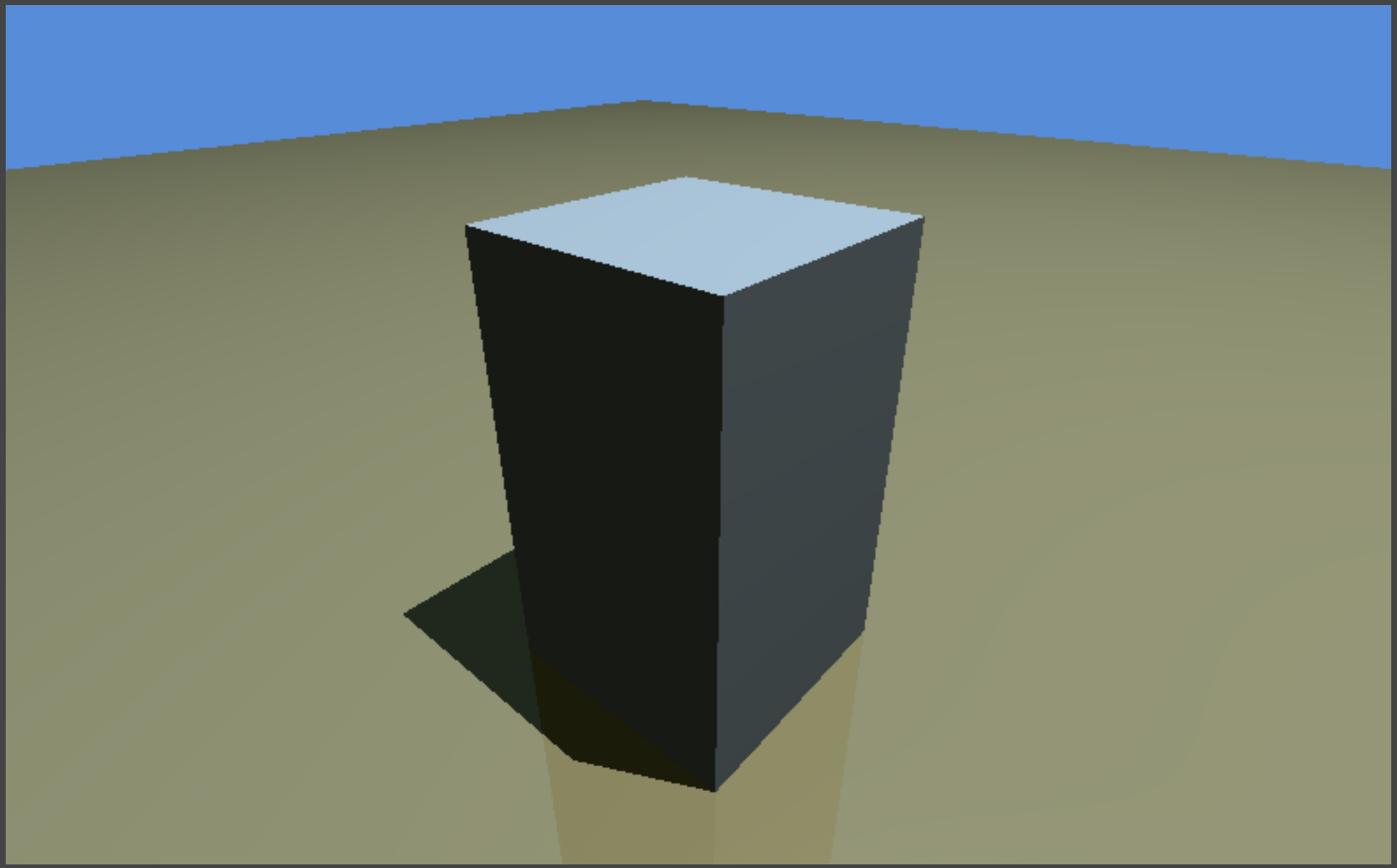
- Along the the shadow ray, generate a ray to calculate the color of the reflections
- Reflected rays are the same as the original rays, with less importance
- Need to change the importance of each recursive ray, or set a maximum number of bounces
- Change the radiance ray payload data:

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

Reflections

```
//light loop above

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

  // reflection ray
  if( importance > importance_cutoff
        && prd_radiance.depth < max_depth) {
    PerRayData_radiance refl_prd;
    refl_prd.importance = importance;
    refl_prd.depth = prd_radiance.depth+1;
    float3 R = reflect( ray.direction, ffnormal );
    optix::Ray refl_ray( hit_point, R, radiance_ray_type,
        scene_epsilon );
    rtTrace(top_object, refl_ray, refl_prd);
    color += reflectivity * refl_prd.result;
  }
  prd_radiance.result = color;
```

Reflections Result

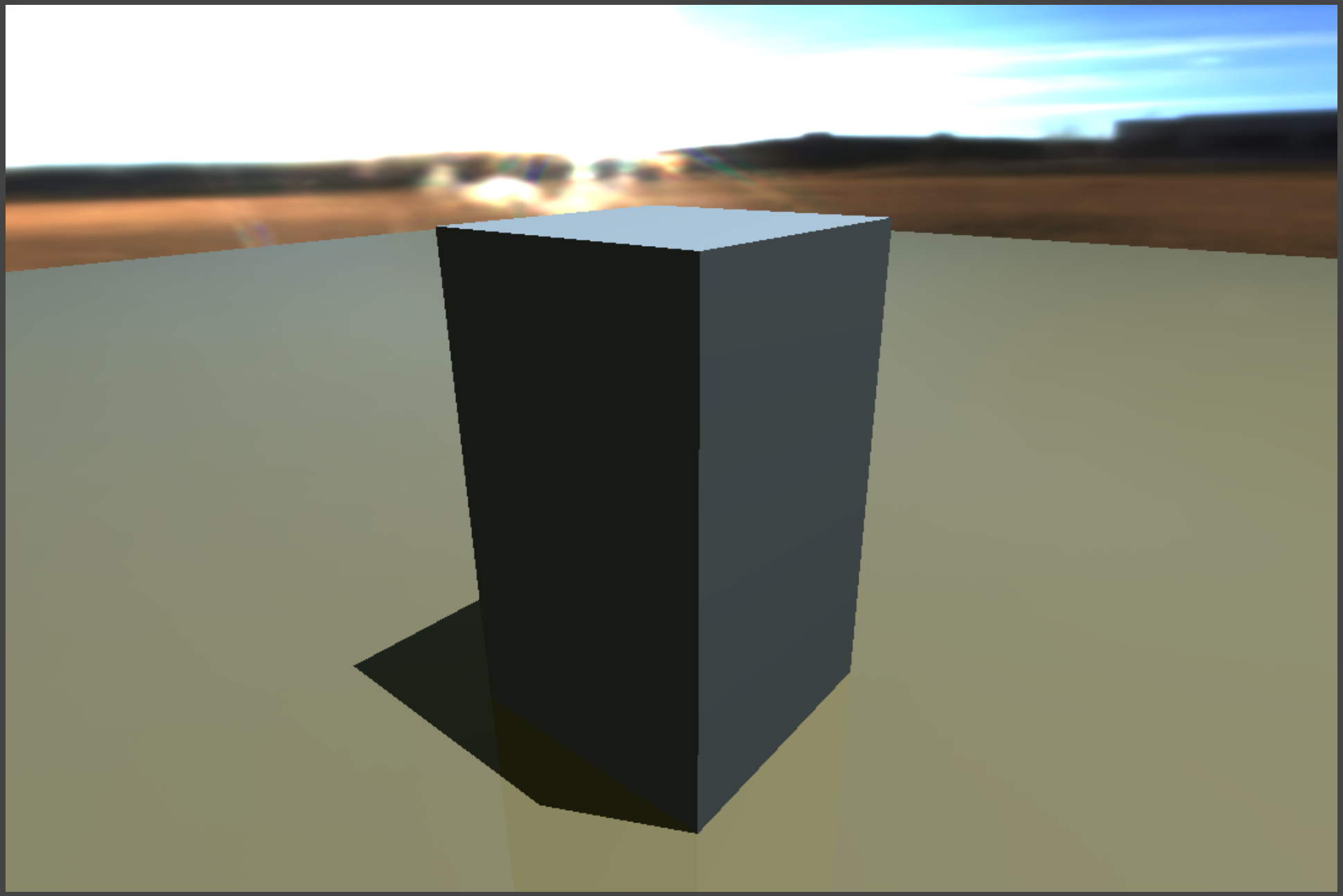


Environment Mapping

- Change our miss shader
- Rather than a background color, use an environment texture
- Use a sphere mapped texture

```
rtTextureSampler<float4, 2> envmap;
RT_PROGRAM void envmap_miss()
{
  float theta = atan2f( ray.direction.x, ray.direction.z );
  float phi   = M_PIf * 0.5f -  acosf( ray.direction.y );
  float u     = (theta + M_PIf) * (0.5f * M_1_PIf);
  float v     = 0.5f * ( 1.0f + sin(phi) );
  prd_radiance.result = make_float3( tex2D(envmap, u, v) );
}
```

Environment Mapping Result



Going Forward

- Modify Ray generation
  - Anti-aliasing
  - Depth of field
- Intersection Shaders
  - Define how to intersect arbitrary objects
  - Also define the bounding box for acceleration structures
- Procedural Content Generation
  - Rust
  - Tiles
  - etc.

Check the SDK tutorials for more

Questions?