NVIDIA® OptiX™ Ray Tracing Engine

Quickstart Guide

Version 4.0

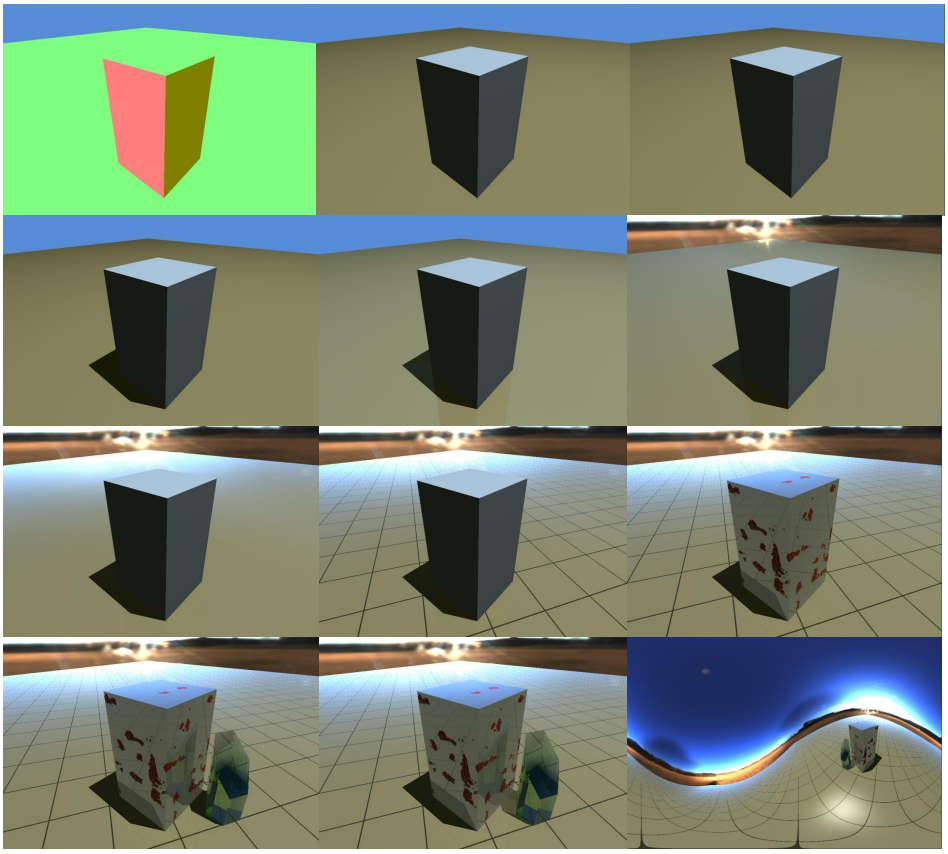
7/18/2016

Table of Contents

CHAPTER 1. TUTORIAL SAMPLE **1**

1.1. TUTORIAL 0 - NORMAL SHADER 2
1.2. TUTORIAL 1 - DIFFUSE SHADING 6
1.3. TUTORIAL 2 - PHONG HIGHLIGHT 8
1.4. TUTORIAL 3 - SHADOWS 9
1.5. TUTORIAL 4 - REFLECTIONS 11
1.6. TUTORIAL 5 - ENVIRONMENT MAPPING 13
1.7. TUTORIAL 6 - FRESNEL REFLECTANCE 14
1.8. TUTORIAL 7 - SIMPLE PROCEDURAL TEXTURE 15
1.9. TUTORIAL 8 - COMPLEX PROCEDURAL TEXTURE 16
1.10. TUTORIAL 9 - PROCEDURAL GEOMETRY 19
1.11. TUTORIAL 10 - SHADOWING TRANSPARENT OBJECTS 22
1.12. TUTORIAL 11 - ENVIRONMENT MAP CAMERA 23
1.13. NEXT STEPS 25

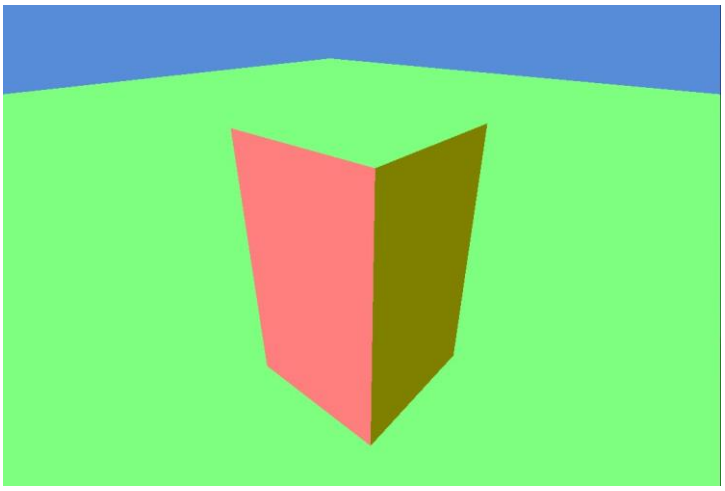
Chapter 1. Tutorial Sample



The OptiX SDK provides a source code sample, *tutorial*, that demonstrates how to implement several basic ray tracing effects, from trivially simple to moderately complex. The sample consists of eleven stages, each stage adding a new effect. In this section, we discuss each of these stages and show programs for both shading and intersection.

This tutorial focuses on the CUDA C programming mechanism and does not describe how the host API is used to set up the objects. The complete source code for both the CUDA C and host API portions is included in the SDK. This tutorial is intended only to get you started with OptiX; advanced features such as visit programs and acceleration structures can be found in other SDK samples. More advanced rendering techniques and scientific computing with OptiX will also not be covered here.

1.1. Tutorial 0 - Normal shader



The most common program type in OptiX is the *closest hit program*. It is executed whenever OptiX finds the closest intersection between a ray and an object. Typically the purpose of the closest hit program is to determine the color of the intersection point. The user can create multiple closest hit programs and bind each to objects in the scene, so that different objects may have different appearances. In this tutorial, we bind one simple closest hit program to each object in the scene. This program is a *normal shader* -- it transforms the object normal into world space and scales it so that each component lies between 0 and 1.. The resulting (x,y,z) vector is interpreted directly as a color and deposited into the payload associated with the ray.

```
RT_PROGRAM void closest_hit_radiance0()
{
  prd_radiance.result = normalize(rtTransformNormal(
                                  RT_OBJECT_TO_WORLD,
                                  shading_normal))
                                  *0.5f+0.5f;
}
```

The program above refers to a variable named `shading_normal`. The intersection programs for the box and for the floor (not shown here; refer to the SDK for their code) will both compute this variable when an intersection is found. Because this variable will be shared between multiple programs, it must be declared in the following special way:

```
rtDeclareVariable(float3,
                  shading_normal,
                  attribute shading_normal, );
```

The resulting color is written to another variable called `prd_radiance`. This is an instance of a user-defined structure that carries data associated with each ray. In this

case, we will write a `float3` color to a portion of that structure called `result`. More information on the two other elements of this structure will be shown later.

```
struct PerRayData_radiance
{
  float3 result;
  float  importance;
  int depth;
};
rtDeclareVariable(PerRayData_radiance,
                  prd_radiance, rtPayload, );
```

There is nothing special about the variable names `shading_normal` and `prd_radiance`. The third argument to the `rtDeclareVariable` macro, called the *semantic name*, is used to bind these variables to the right places in the system. Here, using `rtPayload` as the semantic name lets OptiX know that this data structure should be associated with each individual ray. The `result` portion of the ray payload will be copied to the output of the raytracer in a separate program below.

In addition to the closest hit program, we must specify a *miss program*. Miss programs are run when a ray does not intersect any object. In this case, we just set the resulting color to a user-specified value, `bg_color`.

```
rtDeclareVariable(float3, bg_color, , );

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

The value for `bg_color` is set by the host and can be modified between different invocations of the raytracer. This is the most common mechanism for communication between the host and OptiX programs[1]. OptiX also provides an inheritance model for these variables, but those details are not discussed in this tutorial.

To create the rays themselves, we will use a pinhole camera model. The *ray generation program* is responsible for creating a ray, shooting it into the scene, and copying the resulting color into an *output buffer*. Output buffers are subsequently used by the host for further analysis or by OpenGL for rendering. OptiX can write to an arbitrary number of output buffers, and those buffers can have arbitrary types. In this tutorial, the single output buffer is a two-dimensional RGBA8 image that is designed for efficient transfer to an OpenGL texture. The helper function `make_color` (not shown here) will convert a floating-point RGB color to the appropriate RGBA8 integer value, scaling and clamping as necessary.

---

[1] OpenGL programmers may be familiar with the concept of a uniform variable, which is a similar concept.

```
RT_PROGRAM void pinhole_camera()
{
  size_t2 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 );
}
```

The most important portion of this program is the call to `rtTrace`. There are three arguments to this function:

1) The root of an object hierarchy representing the scene. This hierarchy was created by the host prior to launching the raytracer.
2) A ray, computed above using vector math to simulate the viewing frustum of a pinhole camera.
3) A reference to a local variable that holds the data structure attached to each ray. Because the `prd_radiance` variable (described above) was declared with the `rtPayload` semantic, this local variable will be bound to `prd_radiance` in all other OptiX programs.

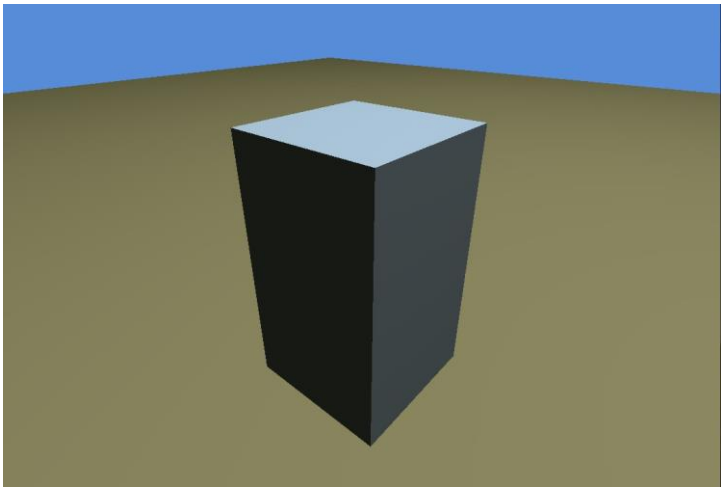
If the ray hits an object, the closest hit program will set the `result` member to the normal color, and if it does not hit any object, the miss program will set `result` to the background color. Once the ray is fully traced, control is returned to the camera program, where we deposit the color into the output buffer.

One final note: Since OptiX supports recursion both in traversal and in shading, a local stack is used to maintain state. If that stack is not large enough then it can overflow. If this occurs, all processing for the current ray generation program is aborted and an *exception program* is executed. In this tutorial, we just set the output buffer to a special color (also set by the host) to alert the user that this occurred.

```
rtDeclareVariable(float3, bad_color, , );

RT_PROGRAM void exception()
{
  output_buffer[launch_index] = make_color( bad_color );
}
```

With all these programs in place, we can run the tutorial program and produce the image shown above. Each of these programs will be executed by OptiX millions of times per second to produce an interactive image. Now we will build on these basics to render a more realistic and complex image.

1.2. Tutorial 1 - Diffuse Shading



The next step is to add simple shading to the objects in the scene. To do this, we just rewrite the closest hit program to do a lighting calculation at each hit point.

```
RT_PROGRAM void closest_hit_radiance1()
{
  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;
}
```

This program has three basic steps:

First, it computes an accurate normal in world space. To do so, it needs to take both the shading normal and the geometric normal, transform them into world space, and then use the `faceforward` function to ensure that the normal is oriented to point back toward the origin of the ray. Most rendering systems distinguish between a shading normal and a geometric normal, e.g. for vertex normal interpolation or bump-mapped surfaces. Although this tutorial does not include such effects, the shader accounts for them anyway to allow this program to be reused immediately in a more complex scene.

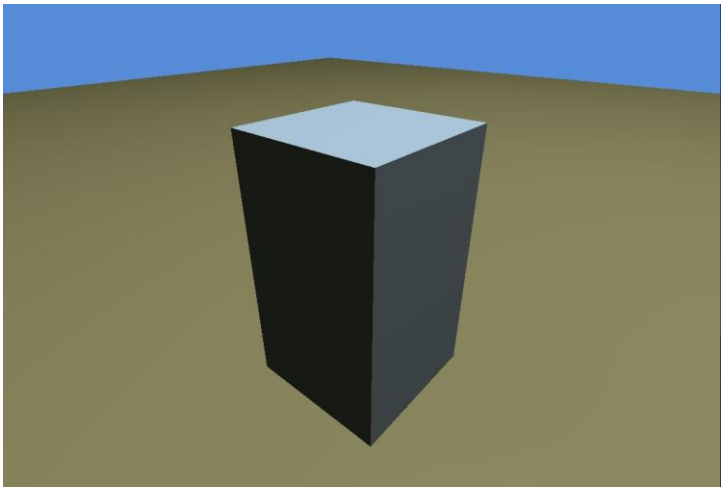
Second, this program computes an *ambient color* for the surface. Ambient lighting is an approximation to the average total illumination falling on the object. In this case, we use two variables to get these parameters from the host. The first, `Ka`, is a property of the object and is bound to either a material object or a geometry object by the tutorial's host code. The other, `ambient_light_color`, is a global property bound to the OptiX context. Note that the OptiX inheritance mechanism is a powerful mechanism for specifying these variables; by attaching them at different points in the scene hierarchy on the host, they can affect any subset of the objects being rendered. Variable inheritance is not discussed in detail in this tutorial; see the OptiX Programming Guide for more details.

Finally, we loop over each light source in the scene and compute the contribution from that light. In this example, each light source is described in a user-declared structure called `BasicLight`, and the set of lights is stored in a one-dimensional input buffer called `lights`. The host code will allocate and populate the `lights` buffer before launching the raytracer. Following is the corresponding structures in the device:

```
struct BasicLight
{
  float3 pos;
  float3 color;
  int    casts_shadow;
  int    padding;     // make the structure more efficient
};
rtBuffer<BasicLight> lights;
```

In the lighting loop, we use a simple Lambertian shading model based on the cosine of the angle between the surface normal and the direction to the light source. The surface color is specified in another host-initialized variable, `Kd`. Subsequent tutorials will procedurally compute this color to simulate more complex visual appearance.

1.3. Tutorial 2 - Phong Highlight

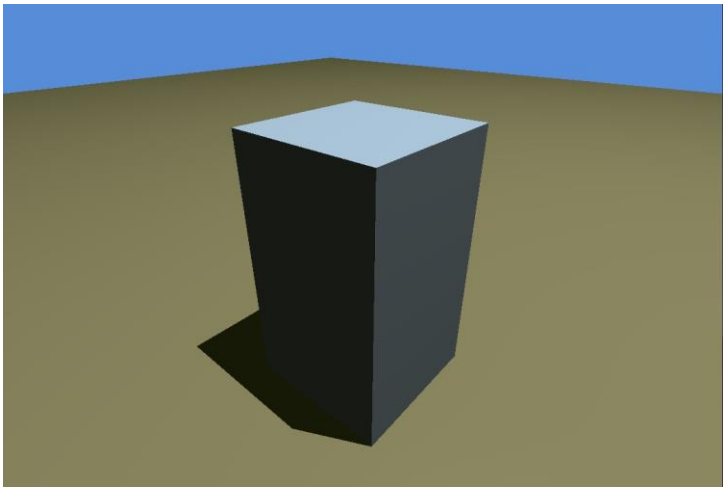


One simple modification to the basic Lambertian shading model is to add a *Phong highlight*: a bright highlight that can be seen on many real-world plastic or metallic objects. We use Jim Blinn's approach that computes the *halfway vector*: the vector that lies halfway between L and the negated ray direction. The cosine of the angle between the halfway vector H and the normal vector N is raised to a user-specified power `phong_exp`, which controls the sharpness of the highlight. In this example, we use the built-in function `pow(x,y)`, which computes x^y. This function leverages special purpose hardware in the GPU to perform this computation efficiently.

The code below shows the small modification that must be made to the diffuse shader from the previous tutorial.

```
...
if( nDl > 0 ){
  float3 Lc = light.color;
  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);

}
...
```

1.4. Tutorial 3 - Shadows



So far, we have not made any image that could not be easily created using OpenGL. However, one of the powerful features of ray tracing is that we can add complex lighting effects (e.g., shadows and reflections) with very little effort. To modify the previous tutorial to support shadows, we add a few lines of code to trace another ray. In this case, the new ray (called a shadow ray) will start on the surface at the shading point, and point towards the light source.

```
...
if( nDl > 0.0f ){
  // cast shadow ray
  PerRayData_shadow shadow_prd;
  shadow_prd.attenuation = 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);
  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);
  }
}
...
```

This code constructs a new ray just like the pinhole camera. Notice that the third parameter to the ray constructor, `shadow_ray_type`, is different from the corresponding argument in the pinhole camera. These ray types are just integer variables supplied by the host code that allows OptiX to handle different ray types separately. Also note that shadow rays have a different kind of ray payload, `PerRayData_shadow`, than camera rays, because shadow rays do not need to carry any data other than occlusion information, represented here as an attenuation factor that ranges from 0 to 1.

We initialize the ray to have an attenuation of 1.0, and invoke `rtTrace` as in the camera code. Notice that OptiX programs are effectively recursive; this call to `rtTrace` will happen deep inside the camera function's invocation of `rtTrace`. For now, we will limit ourselves to opaque objects, so shadow rays that hit objects will be blocked entirely.

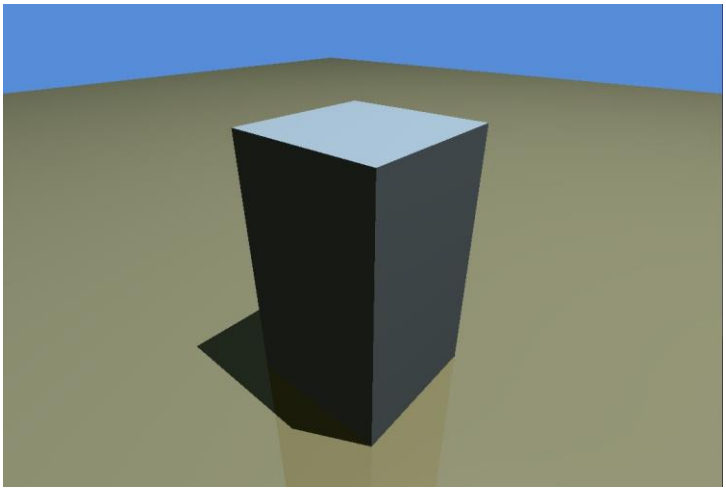
Shadow rays do not require the closest intersection, since we don't care what object the ray hits. Therefore, instead of using a closest hit program, we use an *any hit program* for these rays. Any hit programs are invoked by OptiX at any ray-object intersection. If there are multiple intersections along the ray, the order in which they will invoke the any hit program is unspecified.

```
RT_PROGRAM void any_hit_shadow()
{
  // this material is opaque, so it fully attenuates all
  // shadow rays
  prd_shadow.attenuation = 0.0f;

  rtTerminateRay();
}
```

If a shadow ray intersection is found, we set `attenuation` in the ray payload to zero, indicating that none of the light reaches the object. In addition, we do not need to search for further intersections, so we call `rtTerminateRay`, which returns control immediately to the function that most recently called `rtTrace` (here, the closest hit program shown above).

The closest hit program will add the contribution of the light source only if the shadow feeler was not blocked on the way to the light source. Furthermore, the light's contribution is multiplied by the resulting attenuation, which will allow us to support colored shadows from transparent objects later in this tutorial. Otherwise, this section uses the same Phong shading model from before.

1.5. Tutorial 4 - Reflections



Adding a perfect mirror reflection to a ray tracing system is very simple; these types of effects are what make ray tracing such a flexible image synthesis method. In this case, we will focus only on the material that is bound to the floor. Recall that each object's material can be controlled separately by binding a new closest hit program to its geometry in the host code.

To make the floor's material reflective, we construct a new ray that originates on the surface that we are shading and goes in the direction of perfect mirror reflection. A standard ray tracing text can show you how to derive this reflected direction, but here we use a built-in OptiX function called `reflect` to hide these details. We create a new radiance ray (again, note the third parameter to the ray constructor) and trace it. The resulting color is multiplied by a reflectivity parameter (also supplied by the host code) and added to the surface color that we computed previously.

```
RT_PROGRAM void floor_closest_hit_radiance4()
{
  // Calculate direct lighting using Phong shading, as
  // shown above
  ...
  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;
}
```

Just as with the shadow rays, this function is recursive – computing a color for a reflection ray might send other reflection rays, shadow rays, and so forth. OptiX

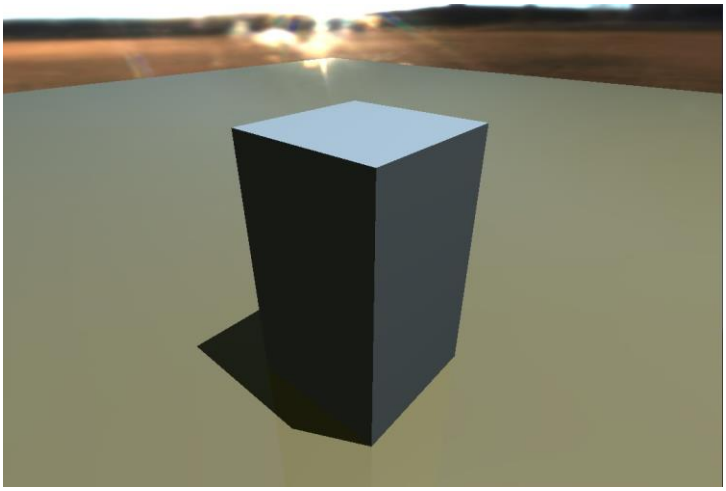
will use a small function call stack to compute all of those results before returning control to this point. This highlights a potential termination issue that would cause OptiX to overflow the call stack (imagine a hall of mirrors where a ray bounces around indefinitely), resulting in the invocation of the exception program shown above.

To address this concern, we need to modify the reflection code to stop sending rays after a certain number of bounces. We will use the `depth` variable in the ray payload to track the recursion depth. If that depth exceeds a user-specified threshold, we do not send a reflection ray. The tutorial example will send up to 100 bounces, which should be adequate for nearly any scene.

```
RT_PROGRAM void floor_closest_hit_radiance4()
{
  // Calculate direct lighting using Phong shading, as
  // shown above
  ...
  if(prd_radiance.depth < max_depth) {
    PerRayData_radiance refl_prd;
    refl_prd.depth = prd_radiance.depth+1;
    float3 R = reflect( ray.direction, ffnormal );
    optix::Ray refl_ray( hit_point, R, 0, scene_epsilon );
    rtTrace(top_object, refl_ray, refl_prd);
    color += reflectivity * refl_prd.result;
  }
  prd_radiance.result = color;

}
```

One more simple modification can improve performance substantially in many scenes. In addition to tracking the depth of the ray, we will track its "importance". Importance tracks how much of the energy in the color will get added to the final color. To track this, we use another variable in the ray payload, initialized to 1.0, and multiplying it by the reflectivity at every bounce. We then add a final condition to the reflection code that avoids sending reflections when the estimated contribution is too dim. Another function, `luminance`, is used to compute a brightness value for the color to compute this importance.

1.6. Tutorial 5 - Environment Mapping



Now that we have enabled reflections, we can make the scene a lot more interesting by adding an environment map to the scene. In this case, we use a third type of declaration:

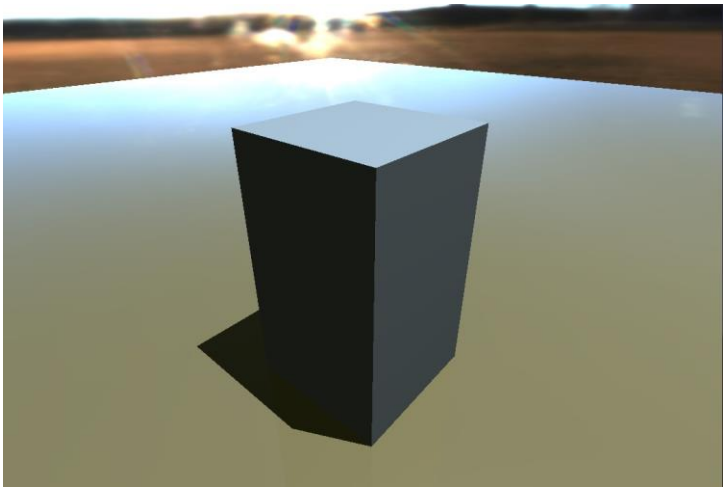
```
rtTextureSampler<float4, 2> envmap;
```

On the host, this texture is bound to an image read from a file. Then we modify the miss program to compute the latitude and longitude of the ray's direction and lookup the color from an environment map that was created by the host.

```
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) );
}
```

Note that we use a high-dynamic range picture for the background, which does not require any modification of the miss program but results in a much nicer reflection.

1.7. Tutorial 6 - Fresnel Reflectance

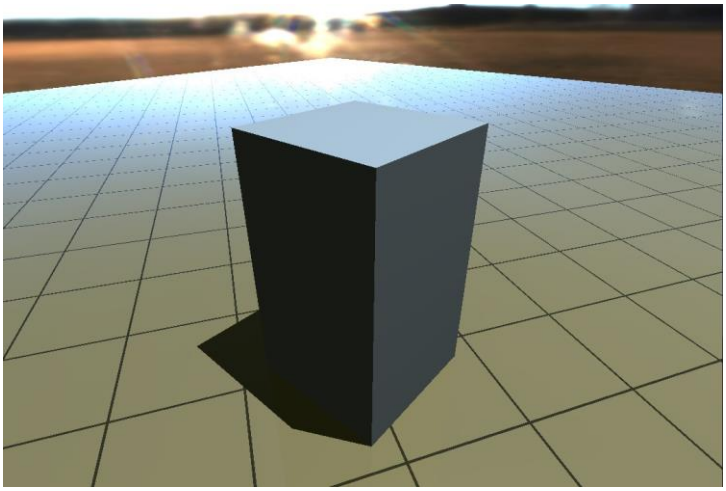


Further richness can be added to the reflections by using an approximation to the Fresnel effect, where rays striking a surface at a grazing angle will be more reflective than rays that are closer to perpendicular to the surface. This effect can be seen in many real-world materials such as a waxed floor, car paint, and glass.

We use the built in `schlick` function to compute this approximation based on the angle between the surface normal and the incoming ray direction. The result is then used for both attenuating the importance and for modulating the reflection computed with the recursive ray. Running the tutorial and looking at the floor from a grazing angle is a good way to see this effect.

```
RT_PROGRAM void floor_closest_hit_radiance5()
{
  ...
  float3 r = schlick(-dot(ffnormal, ray.direction),
                     reflectivity_n);
  float importance = prd_radiance.importance*luminance(r);

  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 += r * refl_prd.result;
  }
  prd_radiance.result = color;
}
```

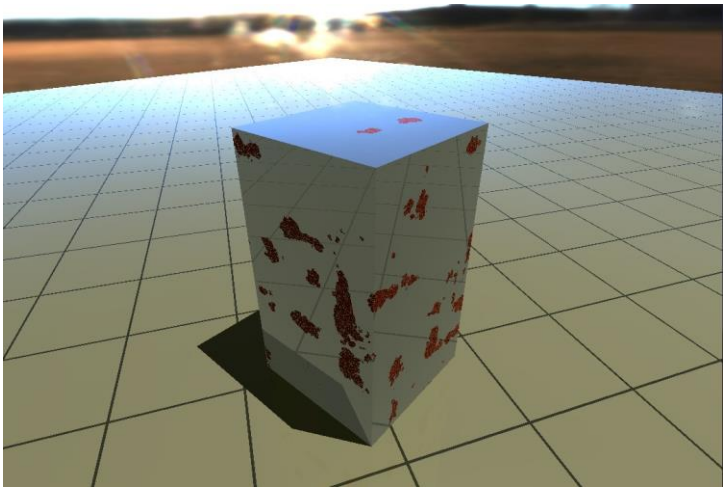
1.8. Tutorial 7 - Simple Procedural Texture



By this point, hopefully it is clear that OptiX programs can perform arbitrary computations. One simple mechanism to add visual detail to an object is to modulate the material properties based on some function. In this section, we use modular arithmetic to simulate a simple tile pattern. The results of this arithmetic are used to choose between a tile color and a crack color. This color is used in place of the single user-specified material reflectivity from the previous examples.

```
RT_PROGRAM void floor_closest_hit_radiance()
{
  ...
  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;
  }
  // Shade with calculated Kd
  ...
}
```

1.9. Tutorial 8 - Complex Procedural Texture

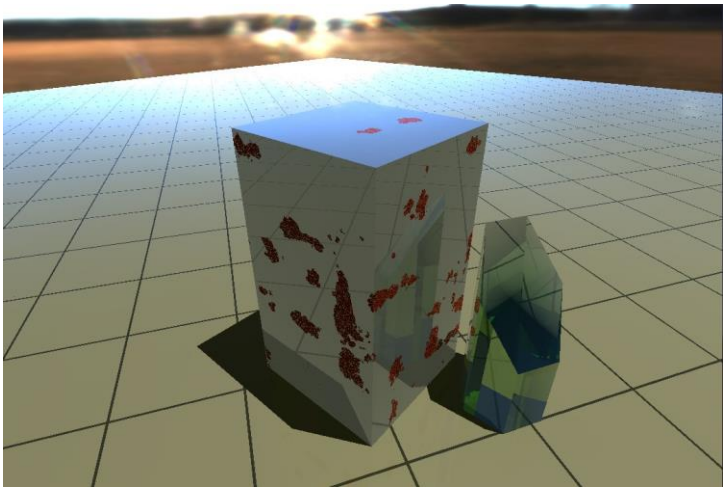


Procedural textures can simulate complex physical phenomena. In this case, we have ported a sophisticated RenderMan shader written by Larry Gritz to OptiX. This shader is fairly complicated, and we will not describe the mathematics here. The interested reader is referred to the RenderMan Repository for the complete source by Larry http://renderman.org/RMR/Shaders/LGShaders/LGRustyMetal.sl.

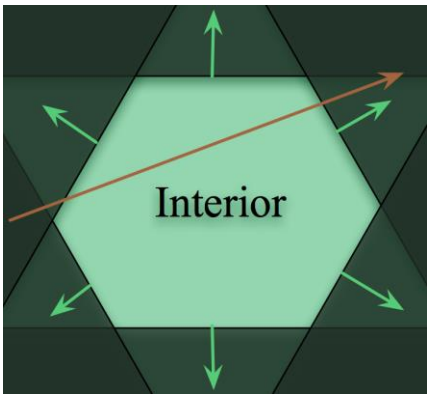
```
RT_PROGRAM void box_closest_hit_radiance()
{
  float3 world_geo_normal   = normalize(rtTransformNormal(
                                        RT_OBJECT_TO_WORLD,
                                        geometric_normal));
  float3 world_shade_normal = normalize(rtTransformNormal(
                                        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;

  // 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 = 1.0f;
  float sum = 0.0f;
  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;

// code continued on next page
```

```
  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);
      optix::Ray shadow 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_radiance.importance*luminance(r);

  // reflection ray
  if(importance > importance_cutoff &&
     rd_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, 0, scene_epsilon );
    rtTrace(top_object, refl_ray, refl_prd);
    color += r * refl prd.result;
  }

  prd_radiance.result = color;
}
```

1.10. Tutorial 9 - Procedural Geometry



One powerful feature of OptiX is the ability to add new geometric primitives. This capability can be used to accommodate triangle vertex formats of almost any kind, making it easier to directly render models stored in any data structure, but it can also be used to add lightweight custom primitives. Here, will use this mechanism to add a "convex hull" primitive. All we need to do is to write an *intersection* program that determines if a ray intersects the object and if so, where along the ray is the first intersection. This program will be executed each time `rtTrace` is called and the acceleration structures determine that the ray is nearby the object.



A convex hull can be defined by a set of oriented planes that bound the area of interest. The ray enters the object when it has crossed the last of all of the planes that the ray "enters" and exits the object when the ray crosses the first of the planes that it "exits". To determine whether the ray is entering or exiting each plane, we use the sign of the dot product between the plane normal and the ray direction. The loop simply tracks the last plane entered and the first plane exited. It also retains the normal that was associated with each plane as it becomes the new enter or exit point.

```
rtBuffer<float4> planes;
RT_PROGRAM void chull_intersect(int primIdx)
{
  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 && t0 < t1; ++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;
  // code continued below
```

If the entry point (t0) is larger than the exit point (t1), then the ray missed the object and this function returns.

Otherwise, we will report the intersection to the OptiX runtime. This happens in two steps. First, `rtPotentialIntersection` determines that an intersection is within the valid t interval for the ray. If it returns `true`, the intersection program computes any attributes associated with the object, which in this case are the shading and geometric normals. Finally, `rtReportIntersection` reports to the OptiX runtime that the attributes are complete. The parameter to this function specifies the material number that is associated with this intersection. This can be used to implement double-sided shading, materials indexed in a triangle mesh, and

so forth. At this point, OptiX will execute the any-hit program associated with this object and material, if any.

```
  // intersection program continued from above
  if(rtPotentialIntersection( t0 )){
    shading_normal = geometric_normal = t0_normal;
    rtReportIntersection(0);
  } else if(rtPotentialIntersection( t1 )){
    shading_normal = geometric_normal = t1_normal;
    rtReportIntersection(0);
  }
}
```

In addition to the intersection program, we must provide a *bounding program* that computes an axis-aligned bounding box for this primitive. Since this program only produces a single object (as opposed to a triangle mesh, for example), we ignore the `primIdx` parameter.

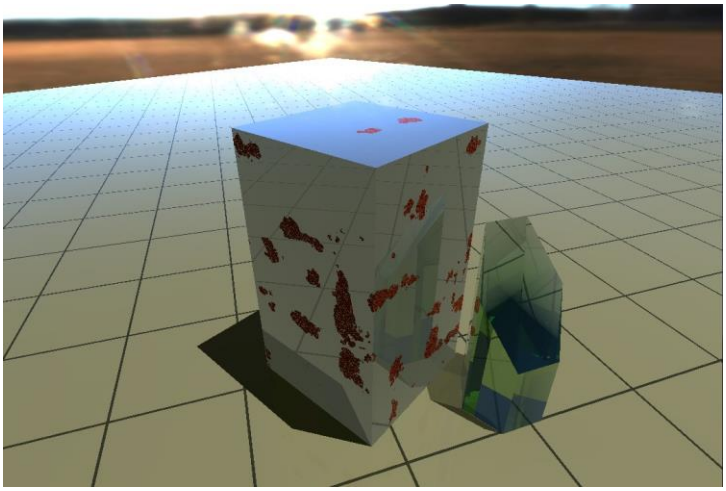
```
RT_PROGRAM void chull_bounds (int primIdx, float result[6])
{
  optix::Aabb* aabb = (optix::Aabb*)result;
  aabb->m_min = chull_bbmin;
  aabb->m_max = chull_bbmax;
}
```

Computing the bounds of the primitive is performed in the host code for this example, so the bounding program just returns the host-provided `chull_bbmin` and `chull_bbmax` variables.

The hexagonal shape shown here is comprised of 8 planes (32 floats total), which are geometrically equivalent to 20 triangles (12 vertices times 3 floats plus 20 indices times 3 integers, or about 3 times as much data); this represents a substantial storage savings. In addition, intersecting a ray with the convex hull will require significantly less computation than its triangle equivalent. The way in which these savings translate to overall running time will depend on a number of factors, such as the quality of the acceleration structures, or the computational expense of any material shaders. Under the right circumstances, programmable object intersection can be a very powerful mechanism for extending the OptiX ray tracing system.

The glass shader for this step is not shown here but is included in the SDK.

1.11. Tutorial 10 - Shadowing Transparent Objects

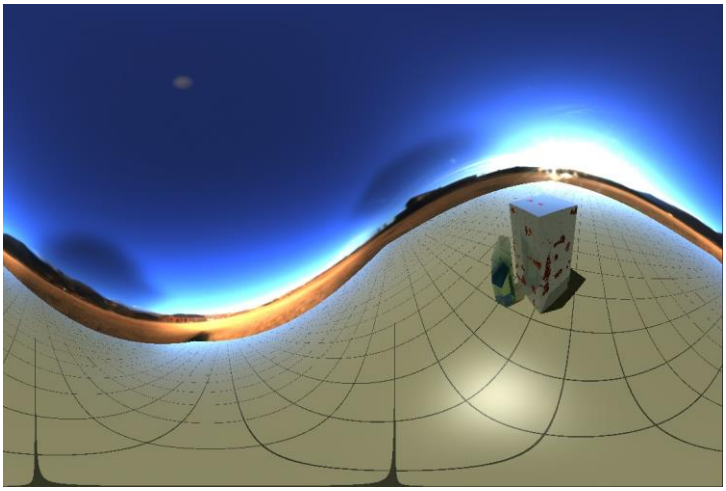


To demonstrate the flexibility of the OptiX any hit program, we will modify the shadow of the glass obelisk to cast a partial shadow. Although this is not a physically-based simulation of glass, it can be computed very quickly and adds substantial perceived realism to the scene. More complex effects, such as caustics, can also be simulated using OptiX, but this is beyond the scope of this tutorial and will usually require tracing large numbers of rays.

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

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

  rtIgnoreIntersection();
}
```

This program is similar to the opaque any-hit program shown in tutorial 3 but has two main differences. First, it computes the ray's fractional attenuation based on the same Schlick Fresnel approximation that we used for providing a realistic reflection. Second, instead of terminating the ray, we use `rtIgnoreIntersection` to allow the ray to continue. In this example, the `glass_any_hit_shadow` program will get executed twice for every ray in the shadow of the glass block–once when it enters the object and once when it exits.

1.12. Tutorial 11 - Environment Map Camera



Our final tutorial demonstrates the flexibility of OptiX (and ray tracing in general) by modifying the pinhole camera ray generation program from the very first example. This new camera shoots rays in a spherical distribution, resulting in an image that can then be used as an environment map in another program, or even as a background image in tutorial step 5.

This program uses the same U, V, W basis and the eye point from the pinhole camera so that the scene can still be manipulated using the mouse controls.

```
RT_PROGRAM void env_camera()
{
  size_t2 screen = output_buffer.size();

  float2 d = make_float2(launch_index) /
             make_float2(screen) *
             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));

  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 );
}
```

1.13. Next steps

This tutorial shows only the beginning of what you can accomplish with OptiX. Further explorations with the tutorial program are suggested. Add a sphere primitive, or use the one provided with the SDK. Modify the shadow payloads to use a color-valued attenuation instead of a single float and use this to make the glass shadow be tinted green. A sampling mechanism can compute ambient occlusion. A modified set of shading and camera programs can perform brute-force path tracing. You can write a program for intersecting a ray with a triangle and import mesh objects.

Examples showing these concepts, plus many more, are included with the OptiX SDK. The SDK also shows how to use random number streams, selector programs, texture maps, meshed objects and acceleration structures. OptiX also supports interoperability between raytracing buffers and OpenGL resources or CUDA buffers, enabling hybrid rendering techniques like efficient use of raytraced images as texture maps. These samples are a valuable source of techniques for building your own high performance ray-tracing based software.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, OptiX, and CUDA are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2010-2013 NVIDIA Corporation. All rights reserved.