Rendering assignment two

Alexander Birke
study nr: 124044 email: alex@alexanderbirke.dk

December 19, 2012

Introduction

This is my handin for the third week for the course "Rendering".

Part 1 - Hard Shadows

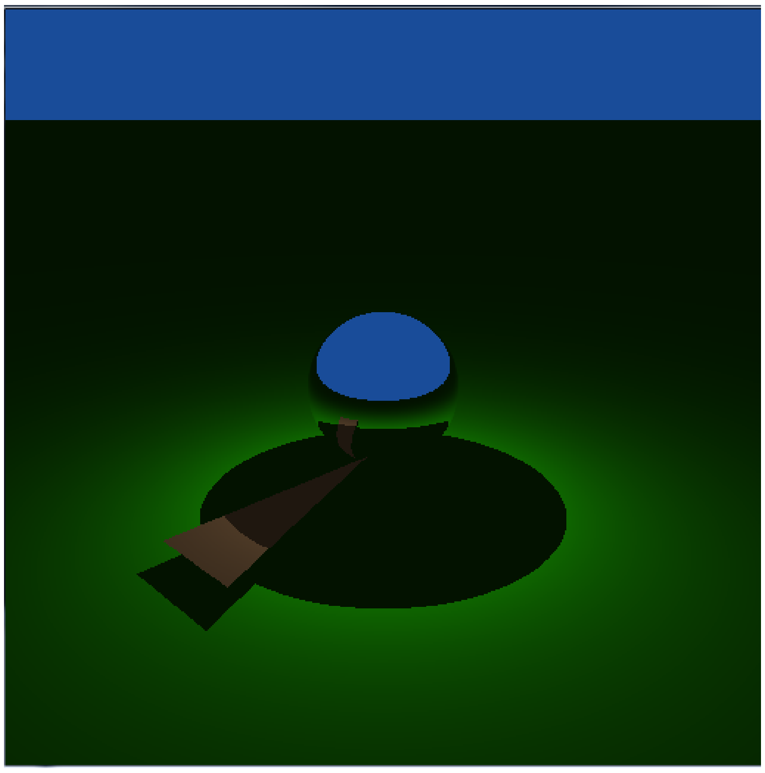
I covered this in the last handin so I have chosen not to explain it again.

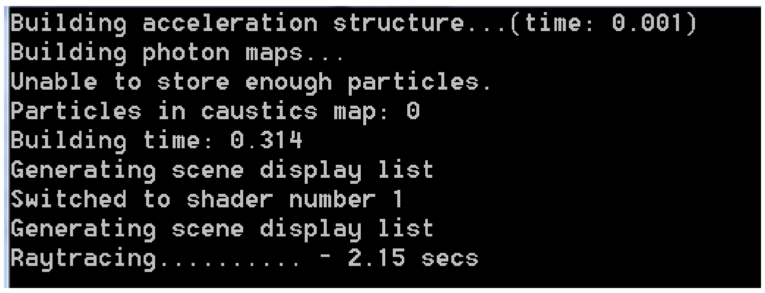
Part 2 - Mirror Reflection

The tracing of reflected rays is implemented in the following way:

```
bool RayTracer::trace_reflected(const Ray& in, const HitInfo& in_hit, Ray&
    out, HitInfo& out_hit) const
{
  float3 reflectedDir = optix::reflect(in.direction, in_hit.
      geometric_normal);

  out = Ray(in_hit.position, reflectedDir, 0, 0.01f);

  if(this->trace_to_closest(out, out_hit))
  {
    out_hit.trace_depth = in_hit.trace_depth + 1;

    out_hit.ray_ior = in_hit.ray_ior;

    return true;
  }
  return false;
```

First the reflected vector is calculated with the use of the reflect function in the optix library. A new ray is then created with start point where the last ray hit and in the direction of the reflection. The ray is then traced. If it hits anything, the outgoing ray's trace depth is set to the incoming ray's plus one. This is done to make sure that the tracing will terminate when a certain trace depth has been reached. The outgoing ray's index of refraction is then set and since it is in the same medium as the incoming ray it is set to have the same index of refraction as the first ray. The function then returns true if the outgoing ray hit anything. The end result look like this:



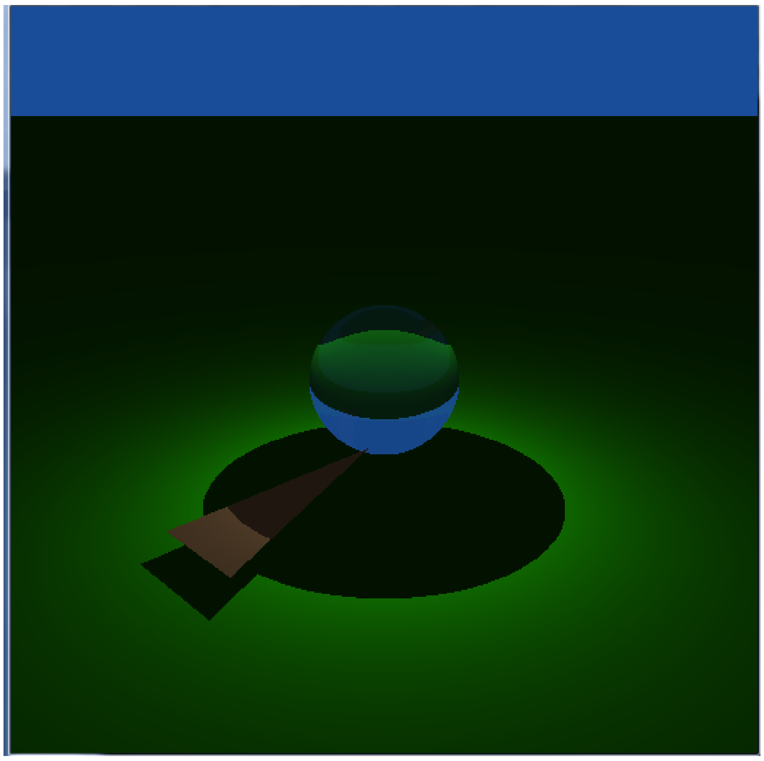


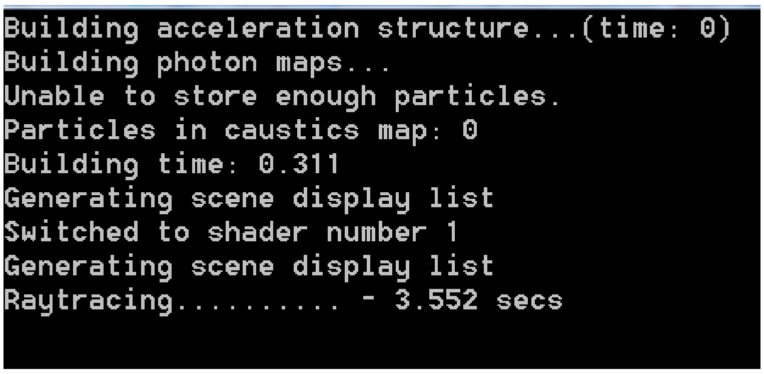
```
Building acceleration structure...(time: 0.001)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.314
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing......... - 2.15 secs
```

Part 3 - Refraction

```
bool RayTracer::trace_refracted(const Ray& in, const HitInfo& in_hit, Ray&
    out, HitInfo& out_hit) const
{
  float3 outDirection;
  float3 outNormal;
  float inTheta;

  out_hit.ray_ior = get_ior_out(in, in_hit, outDirection, outNormal,
      inTheta);

  float3 refractedDirection;

  if(optix::refract(refractedDirection, -outDirection, outNormal, out_hit.
      ray_ior / in_hit.ray_ior))
  {
    out.ray_type = 0;
    out.direction = refractedDirection;
    out.origin = in_hit.position;
    out.tmin = 0.001f;
    out.tmax = 99999;

    if(this->trace_to_closest(out, out_hit))
    {
      out_hit.trace_depth = in_hit.trace_depth + 1;

      return true;
    }
  }
  return false;
}
```

We first declare two variables of the type float3 to hold the direction and normal of the outgoing ray along with one to hold the theta value. The theta value is not used for anything in this piece of code but is needed in order to call the index of refraction helper function. This is the next thing to be called. This function checks if the ray will go out of the object it is currently in, if that is the case the normal of the incoming hit is reverted along with theta and an index of refraction of 1 is returned which means the ray is traveling in air. Otherwise the function finds the index of refraction based on the incoming ray's hit's material reference. Next we check if the outgoing ray will refract. This is done by calling the optix refract function that also calculates the refraction vector. optix functions in general assume that the ray direction they get in still is the original direction so we have to invert the direction that was inverted in the helper function. If no internal refraction occurs, we initialize the outgoing ray with direction of the refraction, with start in the incoming hit, and tmin and tmax set to sensible values. We then trace the ray and if we hit anything, add one to the trace depth of the outgoing hit and return true. The end result look like this:



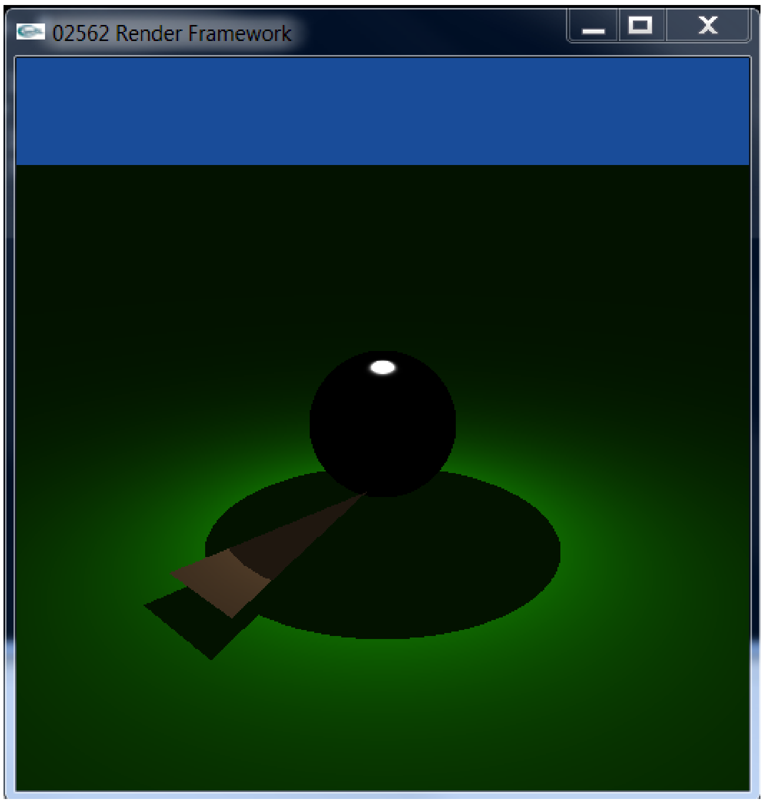


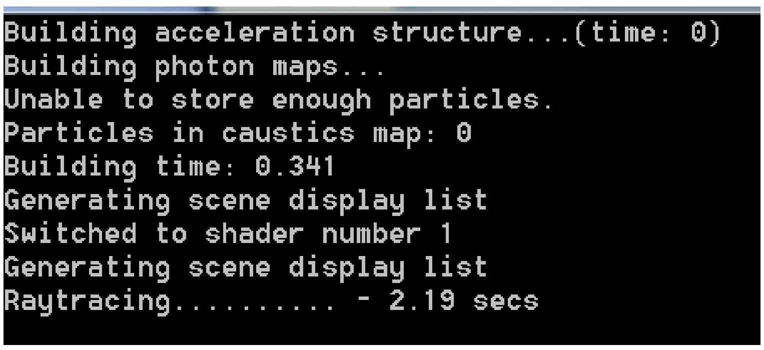
```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.311
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing......... - 3.552 secs
```

Part 4 - Phong Illumination

```
float3 Phong::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  float3 rho_d = get_diffuse(hit);
  float3 rho_s = get_specular(hit);
  float s = get_shininess(hit);

  float3 result = make_float3(0);

  for(int i = 0 ; i < lights.size() ; i++)
  {
    float3 L;
    float3 dir;
    if( lights[i]->sample(hit.position, dir, L) )
    {
      float3 R = optix::reflect(-dir,hit.shading_normal);
      unsigned int sInt = (int)s;

      result += rho_d*L* optix::fmaxf(optix::dot(hit.shading_normal, dir),
         0) + rho_s*L* int_pow(optix::fmaxf( optix::dot(R, -r.direction),
         0), sInt);
    }
  }
  return result;
}
```

We start by getting the material specific parameters of the phong equation and declare a variable to store the result of the shading. Then we loop over the lights in the scene. If the light hits the surface we use the returned direction to the light source and its intensity in the phong equation. We also calculate the reflected light ray in order to implemented specular reflection. To make the calculation of the specular reflection faster we also convert the shininess of the material to an int so we can use the int pow function. The end result look like this:



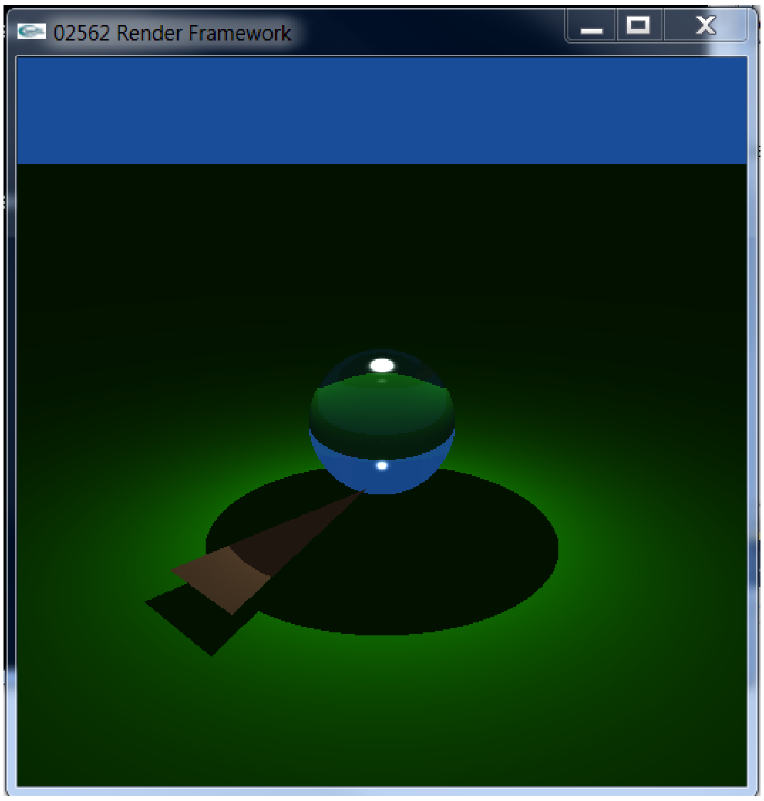


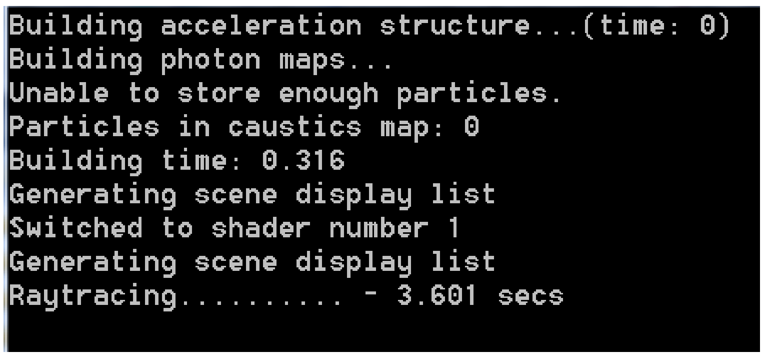
```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.341
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing.......... - 2.19 secs
```

Part 5 - Combining it all

```
float3 Glossy::shade(const Ray& r, HitInfo& hit, bool emit) const
{
  if(hit.trace_depth >= max_depth)
    return make_float3(0.0f);

  Ray reflected, refracted;
  HitInfo hit_reflected, hit_refracted;
  tracer->trace_reflected(r, hit, reflected, hit_reflected);
  tracer->trace_refracted(r, hit, refracted, hit_refracted);

  return 0.1f * Phong::shade(r, hit, emit) +
        0.1f*shade_new_ray(reflected, hit_reflected) +
      0.9f* shade_new_ray(refracted, hit_refracted);
}
```

The first part is to check if the hit given to the function has reached the maximum trace depth, if that is the case the shader returns black. Since the shader implements both refraction and reflection we need to declare a ray and hit info for each of these phenomena. Each of these rays is then calculated. And used to shade the surface with the shade new ray method. These can then again create a new call to the glossy shader as the light bounces around inside the sphere. It is therefore we have the trace depth. The end result is also added together with the phong reflection of the material. Since the ball is made of glass it only has a specular component which gives the highlights on the ball. This highlight is multiplied with the reflectance since it produces the reflection of the lightsource. The end result look like this:





```
Building acceleration structure...(time: 0)
Building photon maps...
Unable to store enough particles.
Particles in caustics map: 0
Building time: 0.316
Generating scene display list
Switched to shader number 1
Generating scene display list
Raytracing.......... - 3.601 secs
```

As we can see, the more rays a method generates the longer it takes to calculate.