Embree – Path Tracing

Luís Paulo Santos, April, 2018

This tutorial with Embree will modify a copy of the viewer tutorial's code:

1. Download the file VI2_EmbreeT4_device.cpp made available on the web site and copy it to $EMBREE_SOURCES$/tutorials/viewer/
2. Modify your Visual Studio solution or Makefile or even CMake file, such that the viewer project (included in the tutorials) compiles VI2_EmbreeT4_device.cpp instead of viewer_device.cpp
3. Go to your $EMBREE_SOURCES$/tutorials/common/scenegraph/ folder and rename the materials.h and obj_loader.cpp files (e.g., to materials_original.h and obj_loader_original.cpp).
4. Download the new files materials.h and obj_loader.cpp from the web site into folder $EMBREE_SOURCES$/tutorials/common/scenegraph/;
5. Build the viewer tutorial.

We will also use a modified version of the Cornell Box distributed with Embree. This is different from the previous Cornell Box therefore:

6. Download cornell_box_VI2_Specular.zip and extract it, making sure that the respective files (containing the cornell_box_VI2_Specular model, with extensions .obj, .mtl and .ecs) become available in the $TUTORIALS_BUILD$/models folder, where $TUTORIALS_BUILD$ is the pathname of the folder where the viewer executable file is stored.
7. Verify your installation by opening a *shell* and from the $TUTORIALS_BUILD$ folder executing

```
viewer –c models/cornell_box_VI2_Specular.ecs
```

Note as this progressive **path tracing** renderer converges to an image with interesting light transport phenomena. In particular, it includes **diffuse indirect light transport**, clearly visible in the ceiling, which is no longer rendered as black and contains **color bleeding** effects.

1. Analysing the path tracer code

Open the VI2_EmbreeT4_device.cpp file, locate the pathTrace() method and carefully go through it. You will see that it has the following basic structure:

```
pathTrace (RTCRay& ray, int depth) {

       dg, material = intersected_geometry_and_material_respectively;

       color = 0. ;
       if (material_is_diffuse) color += direct_illumination();

       if (depth < MAX_DEPTH) {
         if (material_is_diffuse) {
              sec_ray = cosineSampleHemisphere(&pdfDirection);
              trace (sec_ray);
              L = pathTrace (sec_ray, depth++);
              color += π * material_Kd * L;  }  // see NOTE1 below

         if (material_has_specular_reflection) {
              sec_ray = SpecularReflectionRay();
```

```
            trace (sec_ray);
            L = pathTrace (sec_ray, depth++);
            color += material_Ks * L;  }      // see NOTE2 below

        if (material_has_specular_transmission) {
            sec_ray = SpecularTransmissionRay();
            trace (sec_ray);
            L = pathTrace (sec_ray, depth++);
            color += material_Kt * L;  }      // see NOTE2 below
      }
      return (color);
}
```

NOTE1: the multiplication by the cosine and the division by the pdfDirection do not appear because cosine weighted hemisphere sampling has been used, therefore pdfDirection == cosine!

NOTE2: since these are specular phenomena there is one and only one relevant direction, therefore its probability is 1 and there is no division!

A few comments on this code are required:

1. It uses a maximum depth criterion to stop the recursion. We know this is a biased solution (i.e., it does not converge to the correct value). Russian Roulette will be used instead below.
2. It directly samples the light sources (direct illumination). So it is not following a single path, but branching (creating new paths) in the direction of the light sources. This is a common approach that greatly increases the renderer convergence rate. Within the Monte Carlo slang this is referred to as next event estimation.
3. It can create up to three additional rays (diffuse, specular reflection and specular transmission). So this is not exactly tracing a single path: we are allowing multiple paths to be created at each intersection point!

Run the renderer again, change it to full screen mode (press ‘f’) and after allowing some time for Embree to tune its internal parameters write down the frame rate.

2. Stochastically select the BRDF component

Remember that for each pixel multiple (many) primary rays will be shot. Therefore it is possible to avoid sampling all the 3 BRDF components diffuse, specular reflection and specular transmission). We can stochastically select one of these 3 components, render it (avoiding branching into multiple paths) and then, very important, divide the returned radiance by the probability with which you selected that BRDF component.

Uniform distribution

Here I am proposing that you use a uniform distribution, i.e., the same probability is assigned to each of the BRDF components, independently on their values. Here is the high level algorithm:

```
if (depth < MAX_DEPTH) {
      r = RandomNumber();
      if (r<=0.333f) {
            do diffuse reflection …    }
      else if (r<=0.667f) {
            do specular reflection …   }
```

```
        else {
                do specular transmission … }
        color += L / 0.333f;                // see NOTE3 below
}
```

NOTE3: we are dividing by 0.333 (or multiplying by 3) because only 1 out of 3 BRDF components was evaluated!

Change the method code to include this enhancement and build the tutorial. Run the renderer again, change it to full screen mode (press 'f') and after allowing some time for Embree to tune its internal parameters write down the frame rate. Also assess the noise level compared to the previous version.

Importance sample

For each BRDF component use probability proportional to its magnitude relatively to the other components. Here is the high level algorithm:

```
if (depth < MAX_DEPTH) {
        r = RandomNumber();
        // create BRDF CDF
        float Kd_max = reduce_max(material->Kd);
        float Ks_max = reduce_max(material->Ks);
        float Kt_max = reduce_max(material->Kt);
        const float K_normalizer = Kd_max + Ks_max + Kt_max;
        if (K_normalizer < 1e-2f) return (color);
        Kd_max /= K_normalizer;
        Ks_max /= K_normalizer;
        Kt_max /= K_normalizer;
        if (r<= Kd_max) {
                BRDF_weight = Kd_max;
                do diffuse reflection …     }
        else if (r<= Kd_max+ Ks_max) {
                BRDF _weight = Ks_max;
                do specular reflection …    }
        else {
                BRDF_weight = Kt_max;
                do specular transmission … }
        color += L / BRDF_weight;           // see NOTE4 below
}
```

NOTE4: we are dividing by the probability with which this given BRDF component was selected, which is proportional to its magnitude relatively to the other components!

Change the method code to include this enhancement and build the tutorial. Run the renderer again, change it to full screen mode (press 'f') and after allowing some time for Embree to tune its internal parameters write down the frame rate. Also assess the noise level compared to the previous version.

3. Russian roulette

Our path tracer is following paths of fixed length (MAX_DEPTH == 3, in the code). This is a biased approach that never converges to the true solution. Let's change the code such that Russian roulette is used instead. Below you have a pseudo-code suggestion, which you should use instead of the current "`if (depth < MAX_DEPTH) {`" conditional construct:

```
#define continueProb .1f
const float RR_weight = 1.f / continueProb;

// continue the random walk ?
bool continueWalk = (RandomSampler_get1D(sampler) < continueProb);
if (continueWalk) { …
```

The code above definitively decides on the recursion termination (and consequently on the path length) depending on a random value. However, if used as is the result will be very wrong. The %fact that we decide that only `continueProb*100%` of the paths continue and the others terminate, has to be compensated by multiplying the contributions of the paths that were continued by the number of those that potential paths that were not continued; the number of such potential paths that never happen is `1.f/continueProb` .

Therefore, whenever you add a path segment's contribution (`L`) to `color`, you have to multiply by `RR_weight` (or divide by `continueProb`, as you which):

```
color += RR_weight * L / BRDF_weight;
```

Change the method code to include this enhancement and build the tutorial. Run the renderer again, change it to full screen mode (press 'f') and after allowing some time for Embree to tune its internal parameters write down the frame rate. Also assess the noise level compared to the previous version. In particular, play with different values of the continuation probability and assess its impact on both performance and image quality.