Embree – Whitted Ray Tracing Basics

Installation and verification

Luís Paulo Santos, March, 2018

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

1. Download the file `VI2_EmbreeT2_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_EmbreeT2_device.cpp` instead of `viewer_device.cpp`
3. Build the `viewer` tutorial

We will also use a modified version of the `Cornell Box` distributed with `Embree`:

4. Download `cornell_box_VI2.zip` and extract it, making sure that the respective files (containing the `cornell_box_VI2` 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.
5. Verify your installation by opening a *shell* and from the `$TUTORIALS_BUILD$` folder executing

```
viewer –c models/cornell_box_VI2.ecs
```

Viewer Tutorial Code Walkthrough

Let's have a quick look at the viewer tutorial rendering code, in order to get a feeling of how a ray tracer rendering method can be organized.

NOTE: the code description done here is very high level and some of the modifications suggested are not generalizable for more complex shaders.

Open the `VI2_EmbreeT2_device.cpp` file.

There are 3 methods which are accessed by the remaining of the tutorials' code:

- `extern "C" void device_init (char* cfg)` – sets a number of event handling methods, such as handling errors, handling keyboard events and, most importantly, handling the rendering event; the latter will be handled by the `renderTileStandard()` method.
- `extern "C" void device_render (int* pixels, const unsigned int width, const unsigned int height, const float time, const ISPCCamera& camera)` – is responsible for handling render events. Note that it receives as input the image width and height, the current time, the camera parameters and a pointer to the frame buffer.
  this method can eventually decompose the rendering job into tiles (as a way of exploiting thread level parallelism by means of screen space decomposition) and then calls `renderTileTask()`, which is basically `renderTileStandard()` as set by the `device_init()` method described in the previous item.
- `extern "C" void device_cleanup ()` - does the final clean up tasks.

The remaining methods are internal to the viewer tutorial.
`renderTileStandard()` loops through the pixels in the tile, computes their `(x,y)` screen space coordinates and calls `renderPixelStandard()` :

```
void renderTileStandard(int taskIndex, int threadIndex, int* pixels, int width,
                        int height, float time, ISPCCamera& camera,
                        int numTilesX, const int numTilesY)
{
  [ … ]
  for (unsigned int y=y0; y<y1; y++) for (unsigned int x=x0; x<x1; x++)
  {
    color = renderPixelStandard((float)x,(float)y,camera,…);

    /* write color to framebuffer */
    unsigned int r = (unsigned int) (255.0f * clamp(color.x,0.0f,1.0f));
    unsigned int g = (unsigned int) (255.0f * clamp(color.y,0.0f,1.0f));
    unsigned int b = (unsigned int) (255.0f * clamp(color.z,0.0f,1.0f));
    pixels[y*width+x] = (b << 16) + (g << 8) + r;
  }
}
```

`renderPixelStandard()` is the method responsible for shading one pixel:

```
 Vec3fa renderPixelStandard(float x, float y, ISPCCamera& camera, RayStats& stats)
```

Note that a primary ray is created that originates in the camera position and goes through the current pixel `(x,y)`:

```
 /* initialize ray */
   RTCRay ray;
   ray.org = Vec3fa(camera.xfm.p);
   ray.dir = Vec3fa(normalize(x*camera.xfm.l.vx + y*camera.xfm.l.vy +
 camera.xfm.l.vz));
   ray.tnear = 0.0f;
   ray.tfar = inf;
```

The ray is traced through the scene:

```
 /* intersect ray with scene */
   rtcIntersect(g_scene, ray);
```

The shading method is then called. By default, the tutorial computes a pseudo-color based on depth, supported by the `depth_shade()` method.

If the ray doesn't intersect any geometry then its color is set to `(0., 0., 0.)`:

```
 /* shade background black */
   if (ray.geomID == RTC_INVALID_GEOMETRY_ID) {
     return Vec3fa(0.0f);
   }
```

Otherwise the ray length is returned:

```
              return (Vec3fa(ray.tfar*((float)1.e-4)));
```

1. Diffuse BRDF component Shading

Let's change the code such that the primary ray color is identical to the BRDF `Kd` component at the intersection point. Comment and uncomment the appropriate code lines at `renderPixelStandard()`, such that `diffuse_shade()` method is called.

Analyze the code carefully such that you can understand the rendered image.

2. Direct Lighting – no shadows

Knowing that the Cornell Box has one point light source (see file `cornell_box.ecs`) and that you can access the attributes `g_ispc_scene->numLights` and `g_ispc_scene->lights[i]`, comment and uncomment the appropriate code lines at `renderPixelStandard()`, such that `direct_no_shadows_shade()` method is called.

Analyze the code carefully such that you can understand the rendered image.

3. Direct Lighting

Now add shadows by shooting shadow rays! Comment and uncomment the appropriate code lines at `renderPixelStandard()`, such that `direct_with_shadows_shade()` method is called.

Analyze the code carefully such that you can understand the rendered image.

4. Whitted style ray tracing (specular reflections)

Specular reflection rays are now added to the tree of rays. Comment and uncomment the appropriate code lines at `renderPixelStandard()`, such that `Whitted_shade()` method is called.

Note that there is a `depth` parameter in this method, which allows stopping the recursion after a number of reflections (`MAX_DEPTH`). This is a physically and mathematically very inaccurate process of stopping the light transport simulation, but that will be discussed further along in the course.

Specular transmission rays could be added following the same reasoning. Unfortunately, the material description model within `Embree's` tutorials' code does not maintain a correct value for the materials' transparency parameter, `Kt`, and we will thus skip it. Nevertheless, the reasoning is exactly the same as the one used for reflections with a bit of added complexity to account for refraction.

5. Scene description

The scene is described using 3 files: `.mtl, .obj, .ecs`.

The `mtl` file describes the materials present in the scene, including light scattering coefficients such as `Ka`, `Kd` and `Ks` (ambient, diffuse and specular reflection, respectively).

The `obj` file describes the geometric primitives, associating them with the materials described in the `mtl` file.

The `ecs` file provides a few rendering parameters to the renderer and viewer associated with `Embree's` tutorials. You can easily recognize them by their names.

Suggestion: play around with the light sources by adding a few additional and well positioned point light sources.