Embree – Image Based Lighting

Luís Paulo Santos, May, 2018

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

1. Download the files `VI2_EmbreeT6_IBL.cpp`, `infinite_light.cpp`, `infinite_light.h`, `,` `hdrloader.cpp` e `hdrloader.h` 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 the files above instead of `viewer_device.cpp`

NOTE: Steps 3 and 4 below do not apply if you did this during Tutorial4. If not the files `materials.h` and `obj_loader.cpp` can be found together with Tutorial 4 in the web site!

---

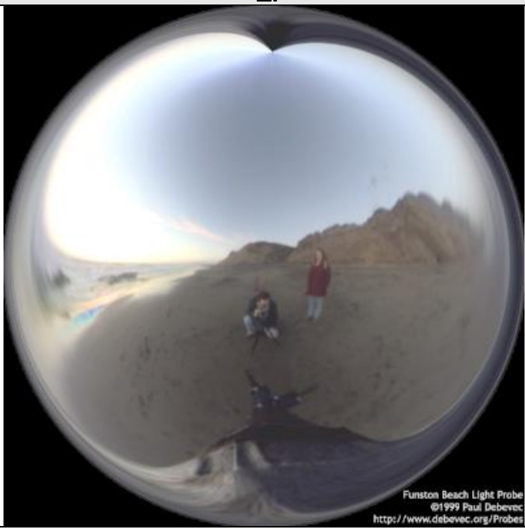
3. Go to your `$EMBREE_SOURCES$/tutorials/common/scenegraph/` folder and rename the `materials.h` and `obj_loader.cpp` files (e.g., to `materials_original.h` and `obj_loader_original.cpp`).
4. Download the new files `materials.h` and `obj_loader.cpp` from the web site into folder `$EMBREE_SOURCES$/tutorials/common/scenegraph/;`

5. Build the `viewer` tutorial.

We will also use a new scene:

6. Download `Debevec_IBL.zip` and extract it, making sure that the respective files (containing the `Debevec_IBL` 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. You will also need the light probes. Download `Dall_probes.zip` (either from the course site or from http://www.pauldebevec.com/Probes/all_probes.zip) and extract it, making sure that the respective files (in HDR format, with extension `.hdr`) become available in the `$TUTORIALS_BUILD$` folder, where `$TUTORIALS_BUILD$` is the pathname of the folder where the `viewer` executable file is stored.
   - HDR images can only be viewed with purpose-made viewers. Even though these are not required for completing this tutorial, if you want to open these files you can install viewers such as IrfanView (https://www.irfanview.com/) or Luminance HDR (http://qtpfsgui.sourceforge.net/). Below you can see two such image probes.
8. Verify your installation by opening a *shell* and from the `$TUTORIALS_BUILD$` folder executing

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

beach_probe	**kitchen_probe**
	

The code structure

The image you have just rendered is not using *Image Based Lighting* (IBL). It is lighten by a classic area light, which within `Embree` tutorials code structure is an instance of the `QuadLight` class.

If you move around in the viewer by pressing "`Shift`" + `Mouse Left Button` you will probably be able to see the `QuadLight` reflected on the platform holding the spheres.

Have a look at the `device_init()` method in `VI2_EmbreeT6_IBL.cpp`; this method is called once at the beginning to allow us to initialize whatever is appropriate. A part of this code is reproduced below. You can see that if the Boolean global variable `InfiniteLightExists` is set to false then a Quad_Light is added to the scene, instead of an Infinite_Light (this is the one we will be using for IBL).

```
/* called by the C++ code for initialization */
extern "C" void device_init(char* cfg)
{
        int l;

        /* do some stuff …
                . . .
         */

        for (l = 0; l < NUM_QUAD_LIGHTS; l++) QuadLightID[l] = -1;
        if (InfiniteLightExists) {
                // add Infinite Light
                add_InfiniteLight();
        }
        else {
                // add area lights
                add_QuadLight();
        }
        rtcCommit(g_scene);
}
```

Search for the line of code below at the beginning of your code file and change it to true:

```
static bool InfiniteLightExists = false;
```

Rebuild the viewer and navigate around a bit. Do you see how the scene is immersed in the environment map? Do you see how the environment map acts as a light source? For instance, the shadows projected by the spheres on the platforms are opposite to the sun's location in the environment map; that's because the sun has the largest radiance values in the whole map.

Let's have a quick look at the method, a part of which is reproduced below:

```
static void add_InfiniteLight() {
        ...
        const int light_map = 0;
        const char* HDRfilename[9] = { "beach_probe.hdr", "building_probe.hdr",
"campus_probe.hdr",
                "galileo_probe.hdr", "grace_probe.hdr", "kitchen_probe.hdr",
                "rnl_probe.hdr", "stpeters_probe.hdr", "uffizi0_probe.hdr" };

        ...
        void* myInfiniteLight;
        Light** newLights = new Light*[totalPreviousLights + totalNewLights];

        g_ispc_scene->lights = newLights;
```

```
            // add infinite light to the set of lights
            myInfiniteLight = InfiniteLight_create();
            InfiniteLight_set(myInfiniteLight, HDRfilename[light_map]);

            g_ispc_scene->lights[totalPreviousLights] = (Light *)myInfiniteLight;
            g_ispc_scene->numLights += totalNewLights;
        }
```

Without going into many details you can see that the method `InfiniteLight_create()` returns a pointer to the new light source and then some of its parameters are set using `InfiniteLight_set()`. The main parameter is the path to the file where the HDR environment map is stored.

The InfiniteLight

Open the file `infinite_light.cpp`.

We will see that the method `InfiniteLight_create()` creates the `InfiniteLight` and an instance of its super class `Light` and then calls `InfiniteLight_set()` in order to set it with some default parameters. `InfiniteLight` is defined as

```
struct InfiniteLight
{
  Light super;             //!< inherited light fields
  HDRLoaderResult hdr_map; // lightMap
  float spherePdf;          //!< pdf of sampling the sphere
};
```

Note that `InfiniteLight_create()` sets the super class pointers to the methods `sample()` and `eval()`. These two methods have to be implemented for each new type of light source we want to define:

```
  self->super.sample = InfiniteLight_sample;
  self->super.eval = InfiniteLight_eval;
```

`InfiniteLight_set()` is implemented as

```
extern "C" void InfiniteLight_set(void* super, const char *HDRfilename)
{

       InfiniteLight* self = (InfiniteLight*)super;
       self->hdr_map.cols = NULL;
       self->hdr_map.height = self->hdr_map.width = 0;
       self->spherePdf = .25f / (float)pi;   // this is 1 / (4* PI) steradians

       if (HDRfilename) {  // read HDR file
             if (!HDRLoader::load(HDRfilename, self->hdr_map)) {
                    self->hdr_map.cols = NULL;
                    self->hdr_map.height = self->hdr_map.width = 0;
             }
             // else { NormalizeToT(self, 8.f); }
       }
}
```

When `InfiniteLight_set()` is called with the name of an existing and valid HDR file, then it loads the pixels (three floating point numbers per pixel: R, G, B) to the `hdr_map.cols` buffer and initializes `hdr_map.height` and `hdr_map.width` with the map dimensions. If something goes wrong then these values are set to `NULL` and 0, respectively.

The `sample()` method

```
Light_SampleRes InfiniteLight_sample(const Light* super,
                                     const DifferentialGeometry& dg,
                                     const Vec2f& s)
```

This method receives as a parameter a pointer to the light source itself (`super`), a reference to the geometry of the intersection point being shaded (dg) and two floating point numbers in the interval [0, 1[(`s`, these are usually uniformly distributed within this interval, but in fact this depends on the renderer implementation).

It returns:

```
struct Light_SampleRes
{
  Vec3fa weight;  //!< radiance that arrives at the given point divided by pdf
  Vec3fa dir;     //!< direction towards the light source
  float dist;    //!< largest valid t_far value for a shadow ray
  float pdf;     //!< probability density that this sample was taken
};
```

Our implementation first checks whether a valid map is loaded and then generates a direction uniformly distributed over the sphere with probability 1 / (4 * PI):

```
// uniform sample the sphere to get the direction
const float phi = float(two_pi) * s.x;
const float cosTheta = 1.f - 2.f * s.y;
const float sinTheta = sqrt(1.0f - cosTheta*cosTheta);
```

This direction is then converted from polar coordinates to Cartesian `(Dx,Dy,Dz)` coordinates:

```
res.dir = cartesian(phi, sinTheta, cosTheta);
```

Finally, the value of the radiance map (remember this is a 2 dimensional matrix of floating point RGB values) has to be read:

```
map_coord = dir2map_coord(res.dir);
radiance = map_lookup(self, map_coord);
```

Conversion from a direction `(Dx,Dy,Dz)` to image coordinates `(x,y)` is according to:

$$d = \sqrt{D_x^2 + D_y^2}\,;\; r = (d > 0\,?\,0.159 * \cos^{-1} D_z : 0)\,;\,(x,y) = 0.5 + r * (D_x, D_y)$$

The `eval()` method

```
Light_EvalRes InfiniteLight_eval(const Light* super,
                                 const DifferentialGeometry& dg,
                                 const Vec3fa& dir)
```

This method receives as a parameter a pointer to the light source itself (`super`), a reference to the geometry of the intersection point being shaded (dg) and a direction in Cartesian coordinates (`dir = (Dx,Dy,Dz)`).

It returns:

```
struct Light_EvalRes
{
  Vec3fa value;     //!< radiance that arrives at the given point (not weighted by pdf)
  float dist;       //!< distance of light intersection along that direction
  float pdf;       //!< probability density that the direction would have been sampled
};
```

If a valid map is loaded and using uniformly distributed sampling of the hemisphere then:

```
map_coord = dir2map_coord(dir);
res.value = map_lookup(self, map_coord);
res.pdf = self->spherePdf;
res.dist = inf;
```

HDR, tone mapping and interactive ray tracing

When you rendered your image you might have noted that some pixels are saturated. This is true on the platform and on the golden sphere, for instance. On the `add_InfiniteLight()` select the kitchen probe by changing the following line:

```
const int light_map = 5;
```

Build the viewer and rerun it. Do you agree the rendered scene if fully saturated?

This happens because the HDR light source can have arbitrarily large values (these are floating points, remember), which means that the pixel values can also be arbitrarily large. But since we are in an interactive context the value of each pixel is **clamped to the interval [0,1] and then multiplied by 255!** This means that any pixel whose value is >= 1 will just become white. You can see the respective code:

```
/* renders a single screen tile */
void renderTileStandard(int taskIndex, ...)
{
  ...

  for (unsigned int y = y0; y < y1; y++) for (unsigned int x = x0; x < x1; x++)
  {
    Vec3fa color(0.f);

    Vec2f jitter = RandomSampler_get2D(samplers[taskIndex]) - Vec2f(0.5f);
    color = renderPixelStandard((float)x+jitter.x, (float)y + jitter.y, ...);
    ...

    /* 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;
  }
}
```

On a non-interactive renderer the HDR image is written to a file and later tone mapped for visualization. These tone mapping operations still convert from whatever values to [0, 1], but using clever algorithms, which avoid saturation and preserve visibility and contrast.

The same could be done here, but it would add some latency to our renderer, depending on the tone mapper complexity.

An alternative is to limit the values present on the HDR environment map, and that is why there is a commented line in `InfiniteLight_set()`:

```
if (!HDRLoader::load(HDRfilename, self->hdr_map)) {
       self->hdr_map.cols = NULL;
       self->hdr_map.height = self->hdr_map.width = 0;
}
//else {
//     NormalizeToT(self, 40.f);
//}
```

Uncomment these 3 lines, rebuild the viewer and rerun it. What do you think of the results?

Now try with all different environment maps by changing in `add_InfiniteLight()` the index of the selected map:

```
const int light_map = 0;
```

Note that for different maps you might have to adjust the normalization value in `NormalizeToT(self, 40.f);`

Pixelization

You might have noticed that when the environment map is directly visible (or visible through some specular light interaction) the individual pixels are very noticeable. The HDR environment map has somehow low resolution to be used for direct visualization.

The value of the map for a given direction is given by:

```
static Vec3fa map_lookup(const InfiniteLight* self, Vec2f map_coord) {
	int x, y, ndx;
	float *pix_addr;

	x = (int)(map_coord.x * self->hdr_map.width);
	y = self->hdr_map.height - (int)(map_coord.y * self->hdr_map.height);  // Radiance 
maps are -Y

	ndx = (y * self->hdr_map.width + x ) * 3;
	pix_addr = (float *)&self->hdr_map.cols[ndx];

	return Vec3fa(*(pix_addr+R), *(pix_addr + G), *(pix_addr + B));
}
```

Can you change this such that instead of returning only the value of the `(x,y)` pixel, it return the average of a 3 * 3 neighborhood?

Noise

Run the viewer using the `beach_probe` and the `kitchen_probe` HDR maps (indexes 0 and 5, normalization constants 8 and 40, respectively).

Why does the noise disappear much faster with the `beach_probe` than with the `kitchen_probe`? You can see both probes in the 1st page of this tutorial. What are the differences between them that can justify the slower convergence of the latter?

RunTime Rotation of the Environment Map

Interesting effects can be achieved by animating somehow the environment map. One form of animating it is to rotate the map as a function of time.

Note that when a ray hits the environment map (this just means that it does not hit any geometry and gets lost in infinity!) its direction is mapped into the image coordinates (the image is the HDR map) by the `dir2map_coord()` method (in `infinite_light.cpp`):

```
static Vec2f dir2map_coord(Vec3fa dir) {
	Vec2f res;
	float d, r;

	d = sqrtf(dir.x*dir.x + dir.y*dir.y);

	if (fabsf(d) < 1.e-4f) { r = 0.f; }
	else { r = 0.159154943f * acosf(dir.z); }

	res.x = .5f + dir.x * r;
	res.y = .5f + dir.y * r;

	return res;
}
```

According to this parameterization it is the `dir.z` coordinate that encodes the orientation angle (`-z` is forward, `+z` is backward). Suppose you add a phase parameter to this method, which changes the reference for the orientation angle (it is 0 radians above, it becomes `phase`…) :

```
static Vec2f dir2map_coord(Vec3fa dir, float phase) {
	...

	if (fabsf(d) < 1.e-4f) { r = 0.f; }
	else { r = 0.159154943f * (phase + acosf(dir.z)); }

	...
}
```

Remember from tutorial 3 (dynamic scenes) that the `device_render()` method (in `VI2_EmbreeT6_IBL.cpp`) is the one that gets called every time our scene is rendered and it has a parameter which reports `time`. Can you change the code such that the phase angle is updated as time goes by and the environment map gets rotated accordingly?