# Embree – Distributed Ray Tracing

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:. This is the same as Tutorial 1, but if you don't have it available then:

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
```

## Jittering

When using viewer you will notice that there are lots of jagged edges along the borders of the objects and shadows. These happen because the image plane is being sampled regularly and at a very low spatial frequency (one primary ray per pixel).

Have a look at the renderTileStandard() method, which is responsible for shooting primary rays:

```
/* renders a single screen tile */
void renderTileStandard(…)
{
	…

	for (unsigned int y = y0; y < y1; y++) for (unsigned int x = x0; x < x1; x++)
	{
		Vec3fa color = renderPixelStandard((float)x, (float)y, camera,
g_stats[threadIndex], samplers[taskIndex]);

		/* 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;
	}
}
```

Note that the direction of the rays will be given by the coordinates of the pixel being sampled, which in this case are (x,y). **Jittering** consists on slightly stochastically perturbing the direction of the rays, pointing them to (x+epsilon_x-0.5,y+epsilon_x-0.5), where (epsilon_x, epsilon_y) are pseudo random numbers uniformly distributed in [0, 1[.

Let's change code such that the point of the image plane passed to is `renderPixelStandard()` jittered. First you must know that every tile (which is eventually handled by a different *thread*) has access to its own random number generator, which is given by: `samplers[taskIndex]`.

You can generate a pair of random numbers uniformly distributed in [0,1[ by:

```
Vec2f jitter = RandomSampler_get2D(samplers[taskIndex]);
```

And then call `renderPixelStandard()` as

```
Vec3fa color = renderPixelStandard((float)x + jitter.x -.5f, (float)y +
jitter.y -.5f, camera, g_stats[threadIndex], samplers[taskIndex]);
```

Try it out, by building and running `viewer`.

What happened? The jagged edges disappeared and were substituted by noise. However, since you are using different jitters for different frames, the noise looks dynamic, i.e., it moves. Re-seed the random number generator at the beginning of the rendering of each tile using `RandomSampler_init(samplers[taskIndex], taskIndex);`

Your function should now look as this:

```
/* renders a single screen tile */
void renderTileStandard(…)
{
	…
	/* initialize sampler */
	RandomSampler_init(samplers[taskIndex], taskIndex);

	for (unsigned int y = y0; y < y1; y++) for (unsigned int x = x0; x < x1; x++)
	{
		Vec2f jitter = RandomSampler_get2D(samplers[taskIndex]);
		Vec3fa color = renderPixelStandard((float)x + jitter.x -.5f, (float)y +
jitter.y -.5f, camera, g_stats[threadIndex], samplers[taskIndex]);

		/* 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;
	}
}
```

## Multiple samples per pixel

Evaluating a single sample per pixel (spp=1) results often in very noisy images. This noise is the way the Human Visual System perceives the variance associated with the stochastic process (jittering is stochastic).

The most naïve form of reducing noise is increasing the number of samples; variance reduces with the square root of the number of samples (i.e., if you take 100 more samples per pixel, the variance in the image will be 10 times smaller).

Note that on the beginning of your C file the constant SPP is defines as having the value 1. Let's change the code such that SPP primary rays are shot per pixel.

The `Vec2f jitter = RandomSampler_get2D(sampler);` code line selects a random position in the pixel area with uniform probability (all points in the pixel area are equally probable). The probability of selecting a given point $p_i$ is equal to the inverse of the pixel area. Since all pixels are identical and the image plane is a virtual entity we can assume the pixel area to be equal to 1 and thus the probability $p(p_i)$ is also 1.

Remember that for each pixel *p* we will combine the values of the *N* different samples (i.e., the radiance evaluated for each primary ray, $L(p_i)$) using the Monte Carlo quadrature:

$$L_p = \frac{1}{N}\sum_{i=0}^{N-1}\frac{L(p_i)}{p(p_i)} = \frac{1}{N}\sum_{i=0}^{N-1} L(p_i)\,, \text{since } p(p_i) = 1/A_p \text{ and } A_p = 1$$

You can therefore change your `renderTileStandard(…)` such that it looks like:

```
for (unsigned int y = y0; y < y1; y++) for (unsigned int x = x0; x < x1; x++)
{
	int sample;
	Vec3fa color(0.f);
	for (sample = 0; sample < SPP; sample++) {
		Vec2f jitter = RandomSampler_get2D(samplers[taskIndex]);
		color += renderPixelStandard((float)x + jitter.x -.5f, (float)y +
		jitter.y -.5f, camera, g_stats[threadIndex], samplers[taskIndex]);
	}
	color /= SPP;


	/* 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;
}
```

Try it for `SPP =1, 4, 16`. What is your opinion on image quality and rendering time?

Do not forget to set `SPP` back to 1 for the remaining of this tutorial.

## Soft Shadows

Our Cornell Box is lighten by a point light source. Let's change this by an area light source.

In the `device_render()` method you will find the following commented line:

```
//add_QuadLights(1, 1);
```

Uncomment it. It will add the area light source just next to the box ceiling. Simultaneously edit the `cornell_box_VI2.ecs` file and delete the point light source (`-pointlight` switch followed by 6 parameters). Rebuild and execute the viewer.

I guess you will agree that the shadows look more realistic, with umbra and penumbra, but they are very noisy. That's because a single ray is being shot per pixel and a single shadow ray is shot per intersection point in order to assess the light source's visibility.

Play around with `SPP` and you get better shadows at a loss in rendering time and interactivity.

A common practice is to shoot several shadow rays per intersection point to each light source, instead of shoothing just one. This is referred to as the number of samples per light (spl) and is obviously ray tracing distributed over the light sources' area.

Note that the constant SPL is defined at the top of your C file. Let's change the direct() method, such that it distributes SPL shadow rays over the light source. Currently it looks like this

```
// direct illumination
Vec3fa direct (RTCRay& ray, RayStats& stats, DifferentialGeometry& dg,
ISPCOBJMaterial* material, RandomSampler& sampler) {
  // do direct lighting with shadows: iterate over lights
  Vec3fa color = Vec3fa(0.f);

  // only do direct if Kd > 0
  if (reduce_max(material->Kd) > 0.f) {
    for (size_t i = 0; i < g_ispc_scene->numLights; i++) {
       const Light* l = g_ispc_scene->lights[i];
       Vec3fa l_color = Vec3fa(0.f);
       Light_SampleRes ls = l->sample(l, dg, RandomSampler_get2D(sampler));
       float cos_L_N = dot(ls.dir, dg.Ns);
       if (cos_L_N > 0.f) {
         RTCRay shadowRay = RTCRay(dg.P, ls.dir, 1.e-2f, ls.dist - 1.e-2f, ray.time);
         rtcOccluded(g_scene, shadowRay);
         RayStats_addShadowRay(stats);
         if (shadowRay.geomID) {  // geomID is 0 if geometry found along shadow ray
           l_color += ls.weight * material->Kd * dot(ls.dir, dg.Ns);
         }
       }
       color += l_color;
    }
  }
  return color;
}
```

The method `l->sample()` stochastically selects a point on the surface of the light source and returns that information on `Light_SampleRes`. In the field `weight` it returns the radiance of the light source at the selected point and direction **already divided by the probability** with which that point was selected. That is why you do not see a division by this probability in this code!

Change the code such that SPL shadow rays are shot and their contributions properly integrated by using the Monte Carlo quadrature:

```
  for (size_t i = 0; i < g_ispc_scene->numLights; i++) {
       const Light* l = g_ispc_scene->lights[i];
       Vec3fa l_color = Vec3fa(0.f);
       for (int s=0 ; s < SPL ; s++) {
         Light_SampleRes ls = l->sample(l, dg, RandomSampler_get2D(sampler));
         float cos_L_N = dot(ls.dir, dg.Ns);
         if (cos_L_N > 0.f) {
           RTCRay shadowRay = RTCRay(dg.P, ls.dir, 1.e-2f, ls.dist - 1.e-2f,
ray.time);
           rtcOccluded(g_scene, shadowRay);
           RayStats_addShadowRay(stats);
           if (shadowRay.geomID) {  // geomID is 0 if geometry found along shadow ray
             l_color += ls.weight * material->Kd * dot(ls.dir, dg.Ns);
           }
         }
       }
       color += (l_color / SPL);
    }
```

Remember to set SPP to 1 and play around with SPL. What happens to rendering time?

## Progressive Render

You should have concluded by now that increasing the sampling rate (SPP, SPL, etc.) reduces variance, but compromises interactivity. The frame rate, i.e., the number of frames per second drops below acceptable values.

A possible solution is to do progressive rendering. We set SPP and SPL to 1. But accumulate each pixel colour in a buffer. This buffer will therefore hold the sum of all values that have been computed for each pixel. Obviously the number of frames for which such accumulation was performed must also be incremented every time a new frame finishes. Then on the `renderTileStandard(…)` method, when the colour is converted to an integer in order to be sent to the frame buffer, the accumulated colour must be divided by the number of frames. This progressive approach allows the image to converge to a better solution as time passes.

Well such buffer already exists in your code and its name is `ProgBuffer`. The variable that counts how many frames have been accumulated also exists, `ProgBuffer_cont`, and is already being incremented every new frame.

Also be aware that every time the camera changes both `ProgBuffer` and `ProgBuffer_cont` are reset to zero.

Therefore you have to change `renderTileStandard(…)` such that progressive rendering is enabled:

```
const float ProgBuffer_countf = (float)ProgBuffer_count;

for (unsigned int y = y0; y < y1; y++) for (unsigned int x = x0; x < x1; x++) {
  int sample;
  Vec3fa color(0.f);
  Vec2f jitter = RandomSampler_get2D(samplers[taskIndex]) - Vec2f(0.5f);
  ProgBuffer[y*width + x] += renderPixelStandard((float)x+jitter.x, (float)y +
jitter.y, camera, g_stats[threadIndex], samplers[taskIndex]);
  color = ProgBuffer[y*width + x] / ProgBuffer_countf;

                    /* 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;
  }
}
```

Remember to set SPP and SPL to 1. Build and execute `viewer`.

Nothing happened? That is probably because you are still initializing the sampler in `renderTileStandard(…)` :

```
      /* initialize sampler */
      RandomSampler_init(samplers[taskIndex], taskIndex);
```

We had to do that when we added jitter to assure that the pseudo random sequence is the same in all frames, and avoid temporal noise. But now we need different samples every frame: that is how the progressive process converges. Therefore just comment that line. Do not worry the samplers are still being initialized once when the renderer starts. Build and run your code. Comment on the results!

## Multiple Light sources

The `add_QuadLights(1, 1)` method allows you to specify multiple light sources: in particular, how many along each axis (x and z) arranged in a square.

Change it such that you have 16 light sources: `add_QuadLights(4, 4);`

Build and run viewer. What happened to your frame rate?

Your renderer is still progressive, but since 1 shadow ray is sent for each light source that takes a lot of time and compromises interactivity.

Can you think of a solution?

Why not distribute the shadow rays among the light sources. After all we are already distributing them over each light area. This is how it goes:

1. Randomly select 1 light source among the existing ones – if uniform probability is used than that probability is the reciprocal of the number of light sources;
2. Shoot a shadow ray to that light source only;
3. Divide that source contribution by the probability with which it was selected; this is the same thing as multiplying by the number of light sources.

There you go: you have your interactive renderer back up again!

```
// direct illumination
Vec3fa direct (RTCRay& ray, RayStats& stats, DifferentialGeometry& dg,
ISPCOBJMaterial* material, RandomSampler& sampler) {
  const float lightPDF = 1.f / (float)g_ispc_scene->numLights;
  // do direct lighting with shadows: iterate over lights
  Vec3fa color = Vec3fa(0.f);

  // only do direct if Kd > 0
  if (reduce_max(material->Kd) > 0.f) {
    do {
       i = (int)(RandomSampler_getFloat(sampler)*g_ispc_scene->numLights);
    } while ((i < 0) || (i >= g_ispc_scene->numLights));

    const Light* l = g_ispc_scene->lights[i];
    Vec3fa l_color = Vec3fa(0.f);
    Light_SampleRes ls = l->sample(l, dg, RandomSampler_get2D(sampler));
    float cos_L_N = dot(ls.dir, dg.Ns);
    if (cos_L_N > 0.f) {
       RTCRay shadowRay = RTCRay(dg.P, ls.dir, 1.e-2f, ls.dist - 1.e-2f,
       ray.time);
       rtcOccluded(g_scene, shadowRay);
       RayStats_addShadowRay(stats);
       if (shadowRay.geomID) {  // geomID is 0 if geometry found along shadow ray
         l_color += ls.weight * material->Kd * dot(ls.dir, dg.Ns);
       }
    }
    color = l_color / lightPDF;
    }
  return color;
}
```