Ensimag

Computer Graphics II: TP3

Enseignants :

Nicolas Holzschuch
- Retour à la page du cours

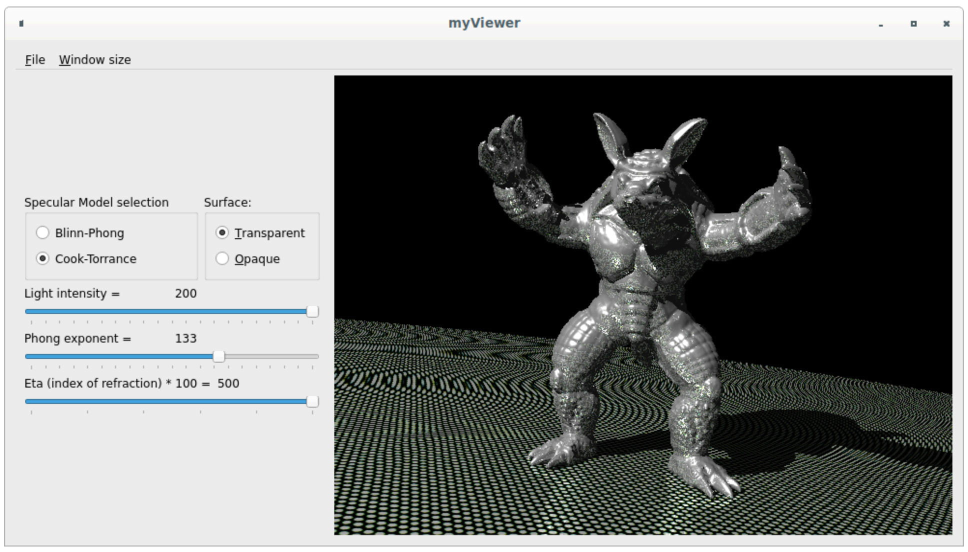
TP3 : distributed Ray-Tracing on the GPU

» TP1
» TP2
» **TP3**

- Spécifications GLSL
- OpenGL 4.4 API Quick Reference Card

Problem statement

The ray-tracer we designed at the previous step is sending only one ray per pixel. For high-frequency textures, this results in a lot of noise:



Specular Model selection: ○ Blinn-Phong ● Cook-Torrance
Surface: ● Transparent ○ Opaque
Light intensity = 200
Phong exponent = 133
Eta (index of refraction) * 100 = 500

Noise is visible with the Armadillo model and the Black_mesh_pxr128 texture.

Our goal is send several rays per pixel, to reduce this noise (it's also the basis for Monte-Carlo ray-tracing, which will be done in the Project).

Generating random rays

The code generating the ray is inside main(), and uses ivec2 pix, which is the integer coordinates of the pixel on the screen (and also the coordinates of the current job in the global invocation). Our first step is to perturb pix by adding random values to it. We want the new pix to still be inside the pixel, but not necessarily at the center.

You will need to code the random() function, since it is not available in GLSL. One possibility is:

```
// Gold Noise ©2015 dcerisano@standard3d.com
// - based on the Golden Ratio
// - uniform normalized distribution
// - fastest static noise generator function (also runs at low precision)
// - use with indicated fractional seeding method.

float PHI = 1.61803398874989484820459;  // Φ = Golden Ratio

float gold_noise(in vec2 xy, in float seed){
       return fract(tan(distance(xy*PHI, xy)*seed)*xy.x);
}
```

There are many other pseudo-random functions, see: https://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl.

Once you have this random ray generator, you need to generate rays continuously. By default, Qt only redraws the scene if there has been an event (e.g. if the mouse has been clicked and moved). To force Qt to redraw the scene regularly, you need to start a QTimer() when you create the glShaderWindow, then set what you need in timerEvent() function.

Once you have all this, your shader is generating multiple random rays. All that remains is to compute the *average* value. You will need to read and write to framebuffer for that. You will also need a counter that tells you how many values have been averaged so far. This should provide visible noise reduction.

Quasi-Random sequences

The random number generator we have designed at the previous sequence is purely random. It can generate samples that are very close to each other, just by bad luck. As we have seen in class, pseudo-random sequences such as Halton sequence or Sobol sequence can provide a better distribution of sample points.

In this step, we are going to use the Halton sequence:

- Generate a 256-points Halton sequence, and store it in a pre-defined array.
- Use the Halton sequence to perturb the sampling point pix. It's better to start at a different point in the sequence for each pixel.
- Check the difference in convergence speed between random generation and Halton sequence.
- You can also use a larger Halton sequence (1024).

Computing variance

One frustrating point with our distributed ray-tracer is that it keeps sending rays, even for pixels where the computation has converged, such as background pixels or smooth areas. In this step, we want to focus our computations on areas where there is a lot of variance, and stop the computation where it has converged. To do this, we will need to compute the *variance* for each pixel:

$$V = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

As you remember, the variance can also be computed using the average of the values and the average of the square of the values:

$$V = \left(\frac{1}{n}\sum_{i=1}^{n}x_i^2\right) - \overline{x}^2$$

You need to edit your shader so that it also computes and stores the average of the square of the values for each pixels (you are already computing the average of the values). Then, use that to compute the variance at each pixel. You can stop the computations if the standard deviation $\sigma = \sqrt{V}$ is smaller than 10 % of the average (use a different threshold if the average is 0!).

For debugging, it's a good idea to visualize where the computations have converged, and where they keep happening.
Contact