Computing Random Points on Sphere

Colas Schretter[*]

August 25, 2006

Monte-Carlo simulations of the photons transport require to pick, very quickly, random points on the surface of a unit sphere. I tried three ways to randomly generate unit direction vectors in three dimensions.

Rejection sampling (naive method) A point is picked randomly in the unit cube with a uniform random generator. If this point lies inside the unit sphere, it is projected on the surface of the sphere by normalizing its coordinates. If the point do not lies inside the unit sphere, then we iterate this procedure. Since the volume of the unit sphere is $\pi/6$, the average number of iterations is equal to 1.909.

Random picking in spherical coordinates (trig method) Any point on the surface of a unit sphere can be represented by two coordinates. The trig method pick randomly the $z \in [-1, +1[$ coordinate to select a slice of the sphere then a point is uniformly selected on this circle by randomly picking an azimuthal angle $\phi \in [0, 2\pi[$. Note that the random generator is only called twice, and no trial-error loop is needed.

Using the uniform_on_sphere distribution of Boost The random_on_sphere distribution of the Boost library uses Normal distributions to fill an stl-type container with coordinates. The procedure work for any dimensions and it is always more elegant to use trusted third party libraries instead of coding yourself.

Benchmarks

Ten millions of three dimensional random vectors have been generated on various machine architectures, using the three methods mentioned above. The GNU gcc 4.0.1 compiler was used, with and without optimization flags (-O6).

Without optimization flags

- On my powerbook 12, 1.33Mhz (IBM G4): 17.78 sec. for naive method, 10.97 sec. for trig method and 81.66 sec. for Boost uniform_on_sphere.

[*]cschrett@ulb.ac.be

- On HP XC Cluster Platform 4000 with AMD Opteron dual-core at 2.4 GHz nodes: 3.91 sec. for naive method, 2.94 sec. for trig method and 15.70 sec. for Boost uniform_on_sphere.

With optimizations flags

- On my powerbook 12, 1.33Mhz (IBM G4): 4.14 sec. for naive method, 5.15 sec. for trig method and 16.65 sec. for Boost uniform_on_sphere.
- On HP XC Cluster Platform 4000 with AMD Opteron dual-core at 2.4 GHz nodes: 1.06 sec. for naive method, 1.46 sec. for trig method and 5.54 sec. for Boost uniform_on_sphere.

Conclusions

When I do not switch on optimization flags of my compiler, then the timings confirm the results of Ken Sloan, referenced in the FAQ of comp.graphics.algorithms. However, with optimization, the naive method is 20% faster than the trig method.

Similar tests have been conducted for the two-dimensional case. Since the area of the unit disc is $\pi/4$, the average number of iterations in the naive method is equal to 1.273. Using polar coordinates, picking a point on a disc only require one random toss and two call to sin/cos trigonometric functions. I constated that the performance of the naive method also outperforms the trigonometric calculations in two dimensions.

It was interesting to experimentally determine the threshold where the naive method begins to perform worse than the current implementation of Boost because of the curse of dimensionality. In four and five dimensions, the naive method significantly outperforms the current implementation based on normal distributions. However, for six and more dimensions, the Boost implementation is significantly faster. In fact in five dimensions, the naive method is one third faster than Boost but is one third slower in six dimensions.

I think that it is mandatory to improve the performances of the Boost implementation. I suggest to convert the dimension parameter to a template argument, the declaration syntax becomes

```
uniform_on_sphere<3> sphere;
```

instead of the current syntax:

```
uniform_on_sphere<> sphere(3);
```

Then, this is possible to specialize template implementations for the 2 and 3 dimensional cases that are so common. The 4 and 5 dimensional cases can also be implemented with the naive method.

The current interface requires that the container have to provide stl-like iterators. I suggest to implement the algorithms by using [ ] accessors instead. This will allow to assign valarrays or custom-made vector classes that are popular in implementations of linear algebra libraries.

The benchmarks should be run on more computer architectures and compilers to comfirms my conclusions. For example, the Visual C++ compiler is often used nowadays, but I do not have any windows machine with installed compiler at hand.

Source Code

```
// Benchmark of several methods to pick randomly points on the unit sphere
// Colas Schretter <cschrett@ulb.ac.be> 2006

#include <cmath>
#include <iostream>
using namespace std;

#include <boost/random.hpp>
#include <boost/timer.hpp>
using namespace boost;

typedef lagged_fibonacci607 generator_type;

const int nbr_samples = 10000000;

struct Vector {
    float x,y,z;

    Vector() {}
    Vector(const float x, const float y, const float z): x(x), y(y), z(z) {}

    float length2() const {
        return x*x + y*y + z*z;
    }

    Vector& normalize() {
        const float inverse = 1.0 / sqrt(length2());
        x *= inverse, y *= inverse, z *= inverse;
        return *this;
    }
};

inline Vector naive(uniform_01<generator_type>& random) {
    Vector direction;

    do {
        direction.x = 2 * random() - 1,
        direction.y = 2 * random() - 1,
        direction.z = 2 * random() - 1;
    } while(direction.length2() > 1);

    return direction.normalize();
}

inline Vector trig(uniform_01<generator_type>& random) {
    const float phi = 2 * M_PI * random();
    const float cos_theta = 2 * random() - 1;
    const float sin_theta = sqrt(1 - cos_theta * cos_theta);
    return Vector(cos_theta, sin_theta * cos(phi), sin_theta * sin(phi));
}

int main(int argc, char** argv) {
    cout << "Random on Sphere Benchmark by Colas Schretter <cschrett@ulb.ac.be> 2006" << endl;
    generator_type generator(42);
    uniform_01<generator_type> random(generator);
    uniform_on_sphere<> sphere(3);
    variate_generator<generator_type&, uniform_on_sphere<> > random_on_sphere(generator, sphere);

    float s = 0; // dummy result
    timer t;

    t.restart();
    for(int i = 0; i < nbr_samples; ++i) {
        const Vector direction = naive(random);

        // do something with the computed values
        s += direction.x + direction.y + direction.z;
    }
    cout << "naive method: " << t.elapsed() << " sec." << endl;

    t.restart();
    for(int i = 0; i < nbr_samples; ++i) {
        const Vector direction = trig(random);

        // do something with the computed values
        s += direction.x + direction.y + direction.z;
    }
    cout << "trig method (spherical coordinates): " << t.elapsed() << " sec." << endl;

    t.restart();
    for(int i = 0; i < nbr_samples; ++i) {
        const vector<double> random_direction = random_on_sphere();
        const Vector direction(random_direction[0], random_direction[1], random_direction[2]);

        // do something with the computed values
        s += direction.x + direction.y + direction.z;
    }
    cout << "boost random_on_sphere: " << t.elapsed() << " sec." << endl;

    // use the dummy result to be sure that the compiler optimizations do not skip computations
    cout << "dummy result = " << s << endl;

    return EXIT_SUCCESS;
}
```