Computing Random Points on Sphere

Colas Schretter[*]

June 6, 2007

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

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.

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.

Hybrid naive and trig method A point is picked randomly in the unit disk with a trial-error loop. Since the area of the unit disk is $\pi/4$, the average number of iterations is equal to 1.2732. Hence, the probability of rejection is lower than in three dimensions. The third coordinate is deduced by using the fact that the coordinates of a point on the unit sphere ensure $x + y + z = 1$. Finally the x and y coordinates are scaled according to the value of z.

Using the uniform_on_sphere distribution of Boost The random_on_sphere distribution of the Boost library uses Normal distributions to fill an STL 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 million of three dimensional random vectors have been generated on various machines and compilers, using the four methods mentioned above. The GNU

[*]cschrett@ulb.ac.be

CPU	Naive	Trig	Hybrid	Boost
Without optimization				
IBM G4 1.33 GHz	18.43	11.35	9.09	80.88
Intel T2300 1.66 GHz	4.58	3.87	2.81	23.60
AMD Opteron 2.4 GHz	3.91	2.94	2.00	15.70
With optimizations				
IBM G4 1.33 GHz	4.33	5.17	2.8	17.30
Intel T2300 1.66 GHz	2.06	2.39	1.14	9.36
AMD Opteron 2.4 GHz	1.06	1.46	0.57	5.54

Table 1: Benchmarks results in seconds w/o compiler optimizations.

gcc 4.0.1 compiler was used on unices with and without optimization flags (-O6). The Visual C++ 2005 compiler was used on a Windows platform with disabled (/Od) and full (/Ox) optimizations. Results are shown in Table 1 for the three machines that has been tested:

- Apple Powerbook 12, 1.33 GHz (IBM G4 1.33 GHz)
- Dell latitude D620 laptop with Intel T2300, 1.66 GHz (Intel T2300 1.66 GHz)
- HP XC Cluster Platform 4000 with AMD Opteron, 2.4 GHz (AMD Opteron 2.4 GHz)

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. In any cases, the hybrid naive and trig method is the winner.

Similar tests have been conducted for the two-dimensional case. Using polar coordinates, picking a point on a disc only require one random toss and two call to sin/cos trigonometric functions. I observed that the performance of the naive rejection 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.

The benchmarks should be run on more computer architectures and compilers to confirm my conclusions. For example, the Intel C++ compiler is often used nowadays...

Proposal to update the Boost implementation

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 new syntax is also more semantic and elegant.

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.

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;

#define PI 3.1415926535897932385

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 * 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));
}

inline Vector hybrid_naive_and_trig(uniform_01<generator_type>& random) {
    Vector direction;
    float s;

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

    const float r = 2 * sqrt(1 - s);

    direction.x *= r;
    direction.y *= r;
    direction.z = 2 * s - 1;

    return direction;
}

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: " << t.elapsed() << " sec." << endl;

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

        // do something with the computed values
        s += direction.x + direction.y + direction.z;
    }
    cout << "hybrid naive and trig method: " << 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;
}
```