レイトレーシングによるコンピュータグラフィクス入門
- アンチエリアシング -

金森 由博*

2014 年 5 月 7 日

1 課題

ソースコード `JitterSampler.h` および `MultiJitterSampler.h` の `sample` 関数を実装しよう。実験のプログラムでそのまま実装しても確認しづらいので、別途用意されたプログラムで実装し、その後、実装したファイルをコピーしよう。

2 アンチエリアシング

前回のテクスチャマッピングの課題で、白黒のチェッカー模様の平面を表示したとき、遠くの模様がギザギザになってしまったはずである (図 1)。この原因を考えてみよう。この実験のプログラムでは、1 ピクセルあたり 1 つのレイを放ち、そのレイが当たった点の色をピクセルの色としている。遠くの場所にレイが当たる場合、レイの方向がほんの少し変わるだけで、交点の位置が大きくずれ、交点の色が白か黒か変わってしまう。1 ピクセルごとに等間隔にレイを放ったとき、たまたま当たったところが白か黒かで、汚いギザギザした模様が出てしまう。

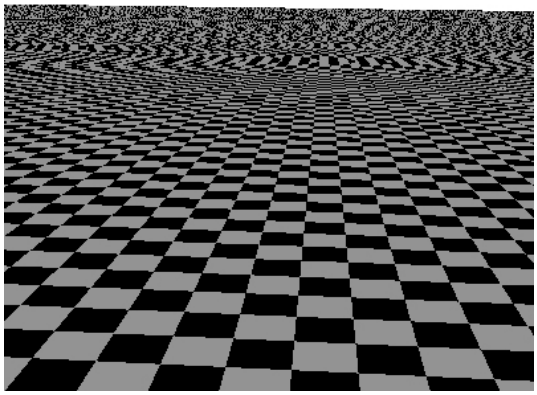


図 1 チェッカー模様の平面で発生するエリアシング.

画面上で考えると、レイを放つのは、あるピクセル位置 (x, y) での入力信号 (この場合は色) を観測 (サンプ

* kanamori@cs.tsukuba.ac.jp

リング) して、標本 (サンプル) を得ることである。一般に、もし入力信号が、サンプリングの間隔よりもある一定以上激しく変化するなら、得られたサンプルでは入力信号を再現できないことが知られている[*1]。こうして、入力信号の急激な変化を捉えきれずに、入力信号とは違う信号 (この場合は画像) が得られてしまうことを、**エリアシング** (aliasing) と呼ぶ。上記の、遠くの場所にレイが当たるというのは、サンプル間隔に対して入力信号 (物体の色) が非常に激しく変化している場合に相当し、エリアシングの好例である。レイトレーシングでは他にも、物体の輪郭や、画面上で小さく表示される物体で、エリアシングが発生しやすい。

エリアシングは、いくらサンプリング間隔を狭くしてもコンピュータで離散的に計算する以上、完全になくすことはできないが、人間の目に目立たなくすることはできる。エリアシングを低減して目立たなくする操作を**アンチエリアシング** (antialiasing) と呼ぶ。要するに、ギザギザをぼかす操作である。そのぼかし方には 2 通りある。入力信号をサンプリングする前にぼかして見た目の変化を急激でなくしておく方法 (pre-filtering) と、入力信号を多めにサンプリング (supersampling) して得られたサンプルの重み付き平均を取る方法 (post-filtering) の 2 つである。レイトレーシングの場合、一般的に、レイを飛ばしてみないことには入力信号がわからないので、後者が一般的に用いられる。つまり、1 ピクセルごとに少しずつ方向の異なる複数のレイを放ち、それらから得られる色の重み付き平均を取る、ということである。実は、どのようにレイの方向を決めるか (サンプリング; sampling)、および、どのように重み付き平均を取るか (フィルタリング; filtering) によって、アンチエリアシングの効率が大幅に違ってくる[*2]。以下、サンプリングとフィルタリングのそれぞれについて説明する。

3 サンプリング

1 ピクセル当たり 1 サンプルで問題なら、1 ピクセルをさらに等間隔に区切ってサンプリングすればよいのでは、と思うかもしれない。しかし、規則的なサンプリング (regular sampling) では、サンプル数を増やしてもエリアシングが起きる。

人間は、規則的なサンプリングで発生するエリアシングは不快に感じるが、ノイズはそれほど不快でないと感じる。これは人間の目における光の受容器の配置が、一定のランダム性を持っている[*3]ことと関係がある。そこで、ある程度ランダムにレイを飛ばし、エリアシングをノイズに置き換えるようにする (そして得られた色を後でフィルタリングする)。

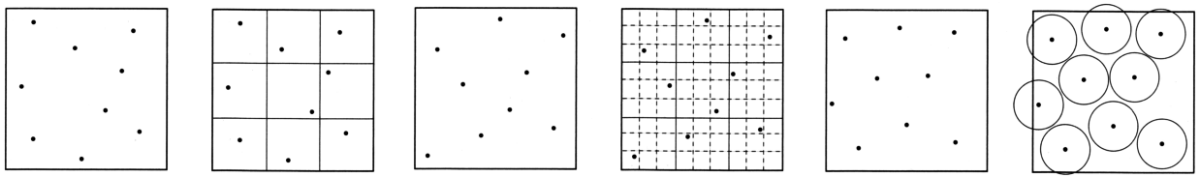
ランダムと言っても、完全にランダムでは問題がある。偏りが生じる可能性があり、例えば、あるピクセルが白と黒の境界にあるとき、白い方に偏ったり黒い方に偏ったりしてはエリアシングが減らない。そこで、満遍なく (一様に; uniform) 分布していながら、ランダム性を持つサンプル点の分布が必要になる。

そのようなサンプリング方法のうち、比較的単純なものが **jittered サンプリング** (層化サンプリング; stratified sampling) である (図 2(a))。これは、1 ピクセルを均等な幅の格子に区切って、格子のセル内でのみ完全にランダムに 1 つサンプル点を決める、というものである。しかしこの方法は、格子内の縦または横の同じ並びでサンプル点が選ばれる可能性があり、縦線や横線に沿ってレイを飛ばすとエリアシングが発生する

[*1] これをナイキスト (Nyquist) の標本化定理 (sampling theorem) という。一定間隔でサンプリングして入力信号を復元するには、入力信号の周波数の少なくとも 2 倍の周波数でサンプリングしなければならない。このサンプリング周波数をナイキスト周波数という。

[*2] CG の研究分野の中でも未だに研究が盛んな領域であり、最新の統計学の知識が導入されている。

[*3] 光の受容器は、ランダムでありながら、互いにある一定間隔よりも離れている、という分布になっている。このような分布を Poisson-disk 分布と呼ぶ (図 2(c))。CG の分野ではこの Poisson-disk 分布がアンチエリアシングに最も効果的とされ、その効率的な生成方法が盛んに研究されている。



(a) jittered sampling (b) multi-jittered sampling (c) Poisson-disk sampling

図 2 代表的なサンプリング手法. 図は “Realistic Ray Tracing, second ed.” から抜粋.

場合がある。これを改良したものに *N-rooks* サンプリングがある。チェスの駒のルーク (rook) は、上下左右に自由に移動できる。このサンプリング方法では、1 ピクセルを $N^2 \times N^2$ 個のセルからなる格子に一様分割し、サンプル点をルークの駒と見なして、N 個のルークの駒を互いに駒の利きが重ならないようランダムに配置する。さらに jittered サンプリングと N-rooks サンプリングを組み合わせたのが *multi-jittered* サンプリング である (図 2(b))。この方法は、jittered サンプリングのようにセルに 1 つサンプル点を配置しつつ、しかも N-rooks サンプリングのように $N^2 \times N^2$ 個のセルに配置したルークの利きが上下左右で互いに重ならない。

この他にも、散らばりの少ない擬似乱数列である Hammersley 点列や Halton 点列などが利用できる。これらは乱数を用いないため高速に計算できるが、規則性があるのでエリアシングを生じる。

4 フィルタリング

ここでいうフィルタリングとは、注目するピクセルを中心とした $K \times K$ ピクセルに含まれるサンプルに対して、信号の変化が平滑化されるよう、重み付き平均を取ることである (low-pass filtering)。一番単純なフィルタリングの方法は、各サンプルに均等な重みをつけて平均化する、というものである。これは、重みをグラフにすると箱型になるので、ボックスフィルタ (box filter) と呼ばれる。理論的には、観測されたサンプルから元の入力信号を復元するには、sinc フィルタ $\mathrm{sinc}\ x = \frac{\sin x}{x}$ が最適であるとされる。sinc フィルタは、中心の重みが最も大きく、中心から離れるにしたがって、重みの大きさが振動しながら減衰する。しかし sinc フィルタは広い範囲のピクセルに渡って計算しなければならないため、実用的にはその範囲を狭めたようなフィルタが用いられる。単純なものとして、中心の重みが最大で、中心から離れると重みが線形に減衰し、一定範囲より外側の重みは 0、となるフィルタを tent フィルタという。重みのグラフが三角形のテントの形になるためこのように呼ばれる。より平滑化の効果が大きいものとして、グラフの形が釣鐘状になるガウスフィルタ (Gaussian filter)、それを 3 次多項式で近似したキュービックフィルタ (cubic filter) などがある。

5 プログラムについて

この実験のプログラムでは、レイのサンプリングと色のフィルタリングを、`RecursiveRayTracer.cpp` の `renderScene` 関数で行っている。

renderScene/RecursiveRayTracer.cpp

```
for (int yi=0; yi<height; yi++)
{
```

```
  for (int xi=0; xi<width; xi++)
  {
    // assume that the provided samples are within [0, 1]^2
    pixel_sampler->sample(pixelOffsets, nSamples);

    for (int si=0; si<nSamples; si++)
    {
      // setup viewing ray
      vec3 dir = (xi+pixelOffsets[si].x-halfWidth)*u
               + (yi+pixelOffsets[si].y-halfHeight)*v
               - screenDist*w;
      dir.normalize();

      samples[si] = trace( Ray(eye, dir) );
    }

    imageBuffer(xi, yi) = pixel_filter->filter(samples, pixelOffsets, nSamples);
  }
}
```

サンプリングの方法は、Sampler クラスを継承した以下のクラスで実装される。

SingleSampler 1 ピクセル当たり 1 サンプルを生成。
RegularNxNSampler 1 ピクセルを $N \times N$ の格子に区切って、格子のセルごとにサンプルを生成。
RandomSampler 1 ピクセル当たり完全にランダムに N サンプルを生成。
JitterSampler jittered sampling でサンプルを生成。
MultiJitterSampler multi-jittered sampling でサンプルを生成。

これらのうち、JitterSampler および MultiJitterSampler の sample 関数を実装してほしい。ただしこのレジュメの冒頭で書いた通り、実験のプログラムでそのまま実装しても確認しづらいので、別途用意されたプログラムで実装し、その後、実装したファイルをコピーするのがよい。

以下、生成するサンプル数が $N \times N$ で、1 画素を $N \times N$ 分割した 1 つの正方形のことをセルと呼ぶことにする。JitterSampler の sample 関数は次のような方針で実装すればよい。

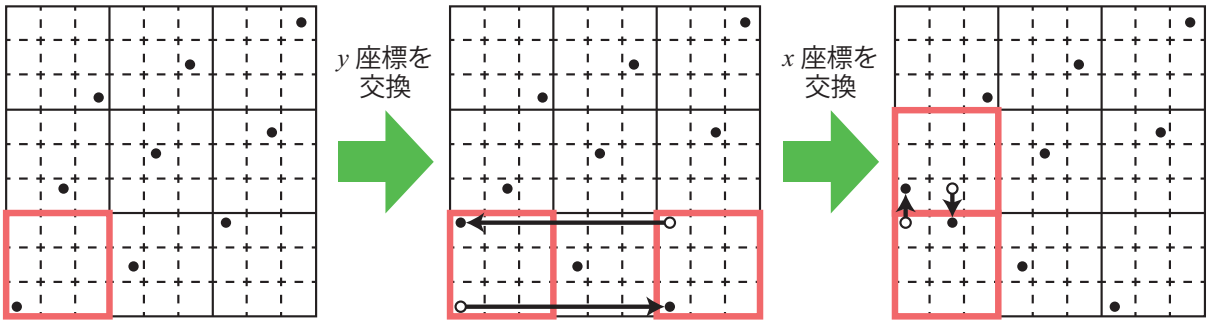


図 3　Multi-Jittered サンプリングの手順の一部.

Jittered サンプリングの擬似コード

```
// サンプルが格納された配列を samples[ ] とする
for (y_i = 0...N-1) {
    y = (y 方向に y_i 番目のセルをランダムにずらした y 座標);
    for (x_i = 0...N-1) {
        x = (x 方向に x_i 番目のセルをランダムにずらした x 座標);
        samples[x_i + y_i * N] = (x, y);
    }
}
```

続いて、`MultiJitterSampler` の `sample` 関数は次のような方針で実装すればよい。

Multi-Jittered サンプリングの擬似コード

```
// サンプルが格納された配列を samples[ ] とする
配列 samples[ ] に N-rooks サンプリングの結果を格納;
for (y_i = 0...N-1) {
    for (x_i = 0...N-1) {
        samples[x_i + y_i * N] と同じ列のセルのサンプル s_y をランダムに選択;
        samples[x_i + y_i * N] と s_y の y 座標を交換;
        samples[x_i + y_i * N] と同じ行のセルのサンプル s_x をランダムに選択;
        samples[x_i + y_i * N] と s_x の x 座標を交換;
    }
}
```

N-rooks サンプリングは例えば次のように実装すればよい。(x_i, y_i) 番目のセルにサンプルを 1 つ配置するとき、そのセルをさらに $N \times N$ 分割したサブセルを考え、(y_i, x_i) 番目のサブセルに、Jittered サンプリングの要領でサンプルを 1 つ生成する。上記の擬似コードにおいて、同じ行のセル同士でサンプルの x 座標を交換したり、同じ列のセル同士でサンプルの y 座標を交換したりしても、N-rooks サンプリングの条件が損なわれないことに注意してほしい。図 3 に手順の一部を示す。

フィルタリングについては、今の実装では 1×1 のボックスフィルタに制限してしまっている。フィルタリングのピクセル幅としてよく用いられるのは、3×3 もしくは 5×5 であるので、アンチエリアシングの効果

は弱いと思われる。

なお、サンプル数を増やすと、レイトレーシングの全体の計算時間は長くなる。この理由は、レイトレーシングで最も時間がかかる計算がレイの追跡であり、サンプル数を増やすということは、追跡するレイの本数を増やすことになるからである。1 ピクセル当たりのサンプル数を N 倍にすると、計算時間はおよそ N 倍になるはずである。サンプル数を多くして実験する場合には、その分だけ画面サイズを小さめにした方が計算時間が短くて済む。

この実験で用いるシーン定義ファイルは pixel_filters.cfg である。このファイルには次のような記述がある。

pixel_filters.cfg

```
/* ...（略）... */

pixel_filter
{
  filter_type = box; // ピクセルのフィルタリング方法

  /* サンプリング方法 */
/*
  sampler_type = single;
  num_samples_per_pixel = 1;  // ピクセルあたりのサンプル数
*/

/*
  sampler_type = random;
  num_samples_per_pixel = 16;  // ピクセルあたりのサンプル数
*/

/*
  sampler_type = jitter;
  num_samples_per_pixel = 16;  // ピクセルあたりのサンプル数
*/

/*
  sampler_type = multi_jitter;
  num_samples_per_pixel = 16;  // ピクセルあたりのサンプル数
*/

//  sampler_type = regular2x2;
```

```
//  sampler_type = regular3x3;

/*
  sampler_type = regular;
  num_samples_per_pixel = 16;  // ピクセルあたりのサンプル数
*/
}

/* ...（略）... */
```

上記の sampler_type および num_samples_per_pixel のペアのいずれかひとつのコメントを外して読みこめば、サンプリング方法とサンプル数を設定できる。ただし、sampler_type として single, regular2x2, regular3x3 を選んだ場合、num_samples_per_pixel によらずサンプル数がそれぞれ $1, 2 \times 2, 3 \times 3$ に固定される。