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
- 三角形メッシュ -

金森 由博*

2014 年 5 月 23 日

1 課題

ソースコード TriangleMesh.cpp ファイルの hit 関数および shadowHit 関数を実装し、レイと三角形メッシュの交差判定を実現しよう。また、スムースシェーディングを実現するために、三角形メッシュの法線ベクトルの計算 (MeshLoader.cpp ファイルの computeSmoothNormals 関数) を実装しよう。

2 三角形メッシュ

複雑な形状を表現するために、コンピュータグラフィクスでは**ポリゴン** (*polygon*; 多角形) の集合で形状を表現する。特に、平面を表現するのに一番単純なポリゴンとして、三角形が使われる。数学的には 3 頂点あれば平面を定義でき、曖昧性がない。4 頂点使って四角形にしたり、それ以上の数の頂点を使ってポリゴンを定義したりすることもできるが、すべての頂点が同一平面上に乗る保証がないので、ねじれた面ができてしまうことがある[*1]ため、三角形が好まれる。

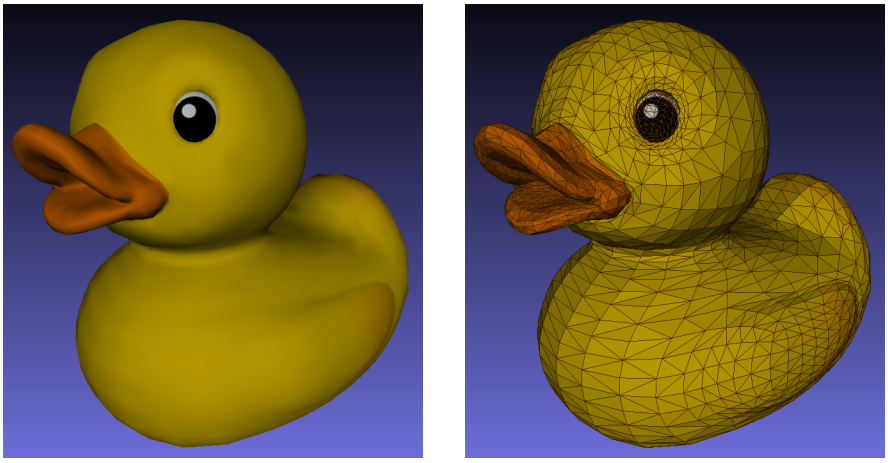
この実験では、三角形のみで構成された形状を扱う。表面を三角形で網目状に覆うことから、このような形状を**三角形メッシュ** (*triangle mesh*; より一般に、ポリゴンから構成されるのであれば**ポリゴンメッシュ**、あるいは単に**メッシュ**) と呼ぶ (図 1)。内部的には、すでに「三角形とレイの交差判定」のところで作成した Triangle クラスの配列 m_Triangles (実際には C++ 標準の動的配列 std::vector) を保持している。

3 総当りによるレイと三角形メッシュとの交差判定

レイと三角形メッシュの交差判定を行うには、単純には、三角形ひとつずつについて調べればよい。つまり、配列に含まれる三角形ひとつひとつと、レイとが交差するかどうかを調べ、レイと交差する三角形のうち、一番手前で交差する三角形との交点を返せばよい。まずこの方針で、TriangleMesh クラスの hit 関数および shadowHit 関数を実装して、交差判定を行ってみよう。実装方針は、RecursiveRayTracer クラスの traceRec 関数と同様である。以下、TriangleMesh クラスの hit 関数の擬似コードを示す。

* kanamori@cs.tsukuba.ac.jp

*1 OpenGL などのグラフィクスライブラリでは、曖昧性をなくすために、ポリゴンをライブラリ内で三角形に分割している。



3D モデル (スムースシェーディング)　三角形メッシュをワイヤフレームで表示

図 1　3D モデルとその三角形メッシュ.

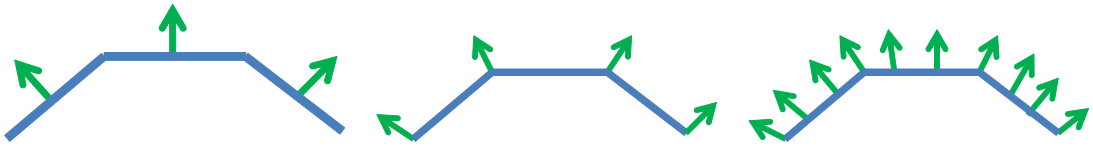
TriangleMesh.cpp

```
bool TriangleMesh::hit(const Ray &r, Real tmin, Real tmax, HitRecord &record) const
{
    // 交点でのレイのパラメータを記録する変数を用意;

    // for (配列 m_Triangles に含まれるすべての三角形) {
    //      三角形の hit 関数を呼び出してレイと交差するか検査;
    //      if (レイが交差 && すでに記録したレイのパラメータよりも値が小さい (より手前)) {
    //          レイのパラメータを更新;
    //          交点での情報を record に記録;
    //      }
    // }

    // 交点が見つかったかどうかを return;
}
```

テスト用のマテリアルとして、法線ベクトルを擬似カラーで表示できるマテリアル (pseudo_normal_color_material) が用意されている。この擬似カラーをどのように計算しているかを説明する。法線ベクトルが単位ベクトルであるとすると、x, y, z の各成分はそれぞれ [-1,1] の範囲の値を取りうる。それぞれ RGB の各チャンネルに対応させて表示するには、それぞれ [0,1] の値に収めなければならない。そこで法線ベクトルの各成分を 1/2 倍して、さらに 1/2 を足す。こうして計算すると、法線ベクトルが x の正の方向に向いていれば赤 (R) っぽく、y の正の方向なら緑 (G)、z の正の方向なら青 (B) っぽくなる。このようにして法線ベクトルを擬似カラーで表示する方法は、レンダリングのデバッグでよく使われる技法である。この pseudo_normal_color_material マテリアルを使ってレイトレーシングすると、OpenGL での表示と同じ結果が現れるはずである。

4　スムースシェーディング

描画結果には、木彫りの熊のように、カクカクした平面でできた形状が現れたはずである。本当であればなめらかな曲面が出てきてほしいところであるが、曲面を三角形メッシュで不連続に近似しているので、よほどメッシュを細かくしても、つなぎ目で不連続であるのが目立ってしまう。このように各ポリゴンに対して、一様な陰影を持つ平面として陰影付け計算 (**シェーディング**; *shading*) を行うことを、**フラットシェーディング** (*flat shading* あるいはコンスタントシェーディング; *constant shading*) と呼ぶ。一方、不連続さが目立たないようなシェーディング方法を、**スムースシェーディング** (*smooth shading*) と呼ぶ。スムースシェーディングには 2 通りあり、いずれも補間を用いる。1 つは**グーローシェーディング** (*Gouraud shading*) と言い、OpenGL や DirectX などのグラフィクスライブラリで標準的に使われている。この方法は、三角形の頂点で陰影を計算し、三角形の内部は頂点の陰影の値を補間する、というものである。陰影の計算を減らせるので計算負荷が低いが、ハイライト (光沢のある材質の表面で光が強く反射される領域のこと) をうまく表現できない。もう 1 つの方法は、**フォンシェーディング** (*Phong shading*)[*2] と呼ばれるものである。この方法は、頂点の法線ベクトルを三角形内で補間し、補間した法線ベクトル (を正規化したベクトル) を使って陰影計算を行う、というものである。グーローシェーディングに比べ計算負荷が多少増えるものの、よりなめらかな曲面、ハイライトを表現できる。このように見かけ上、法線ベクトルを変動させるのは、CG のレンダリングでよく使われる技術である。図 2 にフラットシェーディング、グーローシェーディングおよびフォンシェーディングの模式図を示す。



(a) フラットシェーディング　(b) グーローシェーディング　(c) フォンシェーディング

図 2　フラットシェーディング、グーローシェーディングおよびフォンシェーディングの模式図. フラットシェーディングは面の法線を使ってシェーディングするので, 陰影は面ごとに決定される. グーローシェーディングでは頂点ごとの法線ベクトルで陰影を計算し, その結果を面内で補間する. グーローシェーディングではハイライトをうまく捉えられない. フォンシェーディングでは頂点ごとの法線ベクトルを面内で補間したのち, 補間した法線ベクトルで陰影を計算する. フォンシェーディングではハイライトをうまく捉えられる.

この実験ではフォンシェーディングを用いてスムースシェーディングを実現しよう。このためには、各頂点の法線ベクトルの計算を追加するのと、`Triangle` クラスの `hit` 関数の計算を少し変更する必要がある。三角形メッシュが読み込まれると (読み込み部分の詳細は、このレジュメの「付録: 三角形メッシュの読み込み」参照)、`MeshLoader` クラスの `computeSmoothNormals` 関数で各頂点の法線ベクトルを計算することになっている。まずここを実装しよう。法線ベクトルは、なめらかな曲面を表現するために、どれかひとつの面の法線ベクトルではなく、その頂点を含むような、隣接するすべての面の法線ベクトルの加重平均とする。加重平均といっても、重み付けの方法はいくつか考えられる。ここでは、三角形の面積に応じた重みをつけて平均を

[*2] フォンの反射モデルを考えた Phong 氏が考えたシェーディング方法なので、これも Phong という名前が付いているが、混同しないよう注意。

とることにする。ある頂点 $\mathbf{v}_0$ が、K 個の三角形に含まれているとする。それらの三角形のうちのひとつを三角形 j $(j = 1, 2, 3, \ldots K)$ として、三角形 j のうち頂点 $\mathbf{v}_0$ 以外の 2 頂点を $\mathbf{v}_1^j$, $\mathbf{v}_2^j$ とする。ただし、面の向き (法線ベクトルの向き) を決めるために、$\mathbf{v}_0$, $\mathbf{v}_1^j$, $\mathbf{v}_2^j$ が反時計回りに並んでいるものとする。すると三角形 j の法線ベクトル $\mathbf{n}_j$ は、

$$\mathbf{n}_j = \frac{(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)}{||(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)||} \tag{1}$$

となり、三角形 j の面積 A_j は

$$A_j = \frac{1}{2}||(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)|| \tag{2}$$

である。ここで $\times$ は外積を表す。各三角形 j の面積 A_j を重みとして、法線ベクトルの加重平均 $\mathbf{n}_0$ を計算すると

$$\mathbf{n}_0 = \frac{\sum_j A_j \mathbf{n}_j}{||\sum_j A_j \mathbf{n}_j||} \tag{3}$$

$$= \frac{\sum_j \frac{1}{2}||(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)|| \cdot \frac{(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)}{||(\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)||}}{||\sum_j A_j \mathbf{n}_j||} \tag{4}$$

$$= \frac{\sum_j (\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)}{||\sum_j (\mathbf{v}_1^j - \mathbf{v}_0) \times (\mathbf{v}_2^j - \mathbf{v}_0)||} \tag{5}$$

となる。

この実験のプログラムでは、`MeshLoader` クラスで読み込まれた形状データとして、すべての頂点の配列 `verts` と、各三角形を構成する頂点のインデックスを保持する配列 `tris` が、`computeSmoothNormals` 関数の引数として提供されている。これらを使って、フォンシェーディングのための法線ベクトルを計算し、引数として与えられている各頂点の法線ベクトルを格納した配列 `normals` に法線ベクトルを格納する。まずは、各三角形の法線ベクトル (単位ベクトルを面積で重み付けしたもの、つまり正規化していないベクトル) を計算して、自分で用意した配列に保存する。そして、各頂点について、その頂点を含むような三角形の法線ベクトルを足し合わせ、正規化したものをその頂点の法線ベクトルとすればよい。

続いて、`Triangle` クラスの `hit` 関数の計算を少し変更しよう。各頂点の法線ベクトルを、重心座標 (*barycentric coordinates*; レジュメ「三角形とレイの交差判定」を参照) を使って三角形内部の点で補間する (図 3)。レイと三角形の交点 $\mathbf{p}$ での重心座標 (α, β, γ) と、三角形の各頂点での法線ベクトル $\mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2$ を用いて、交点 $\mathbf{p}$ での法線ベクトル $\mathbf{n}_\mathbf{p}$ を求めると

$$\mathbf{n}_\mathbf{p} = \frac{\alpha\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2}{||\alpha\mathbf{n}_0 + \beta\mathbf{n}_1 + \gamma\mathbf{n}_2||} \tag{6}$$

となる。ソースコードでは $\mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2$ がそれぞれ `m_Normals[0]`, `m_Normals[1]`, `m_Normals[2]` というメンバ変数に対応するので、それらを用いる。同様に、テクスチャ座標についても変更しておこう。交点 $\mathbf{p}$ でのテクスチャ座標を、メンバ変数 `m_Texcoords[0]`, `m_Texcoords[1]`, `m_Texcoords[2]` [*3]を、α, β, γ を重みとした線形補間として計算する。

最後に、設定ファイルを変更しよう。初期設定ではフラットシェーディングを使うようになっているが、これをスムースシェーディングを使うようにする。具体的には、下記 `triangle_mesh` の `shading_type` を `flat` から `smooth` に変更する。

[*3] 各頂点のテクスチャ座標は `MeshLoader` クラスで読み込まれる。

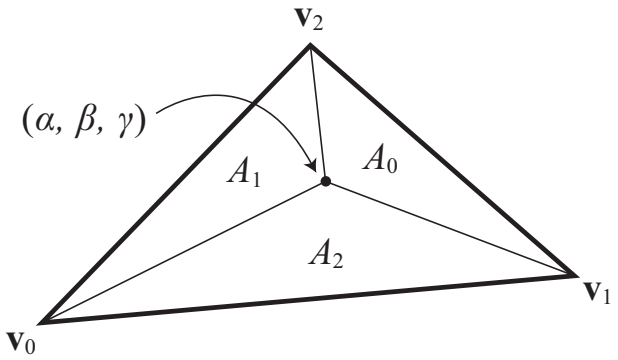


図 3　重心座標.

mesh_test.cfg

```
triangle_mesh
{
  pseudo_normal_color_material; // 法線ベクトルを疑似カラーとして表示するマテリアル
/*
  phong_material {
    ambient = (0,0,0);
    diffuse = (0.5,1,0.5);
    specular = (0,1,0);
    shininess = 32;
  };
*/

// シェーディングの方法 … flat または smooth
    shading_type = flat;
//  shading_type = smooth;

  filename = "bunny_409.obj"; // ファイル名 … obj 形式か ply 形式をサポート
}
```

ここまでの実装によって、なめらかな曲面が表示されるようになったはずである。しかし、この実装では、三角形の数が増えるとレンダリングに非常に時間がかかる。なぜなら、1 つのレイごとに、すべての三角形との交差判定を行なっているからである。人間が見れば明らかに交差しないと思えるような三角形も、計算の対象に含まれてしまう。レイと交差しなさそうなものを間引いて計算するために、様々な空間データ構造が提案されている。空間データ構造は空間を分割して管理し、レイが通過する付近にあるものとだけ交差判定を行えるようにする。空間データ構造の例としては、一様格子 (ユニフォームグリッド; uniform grid)、八分木 (octree)、階層的バウンディングボリューム (Bounding Volume Hierarchy; BVH)、kd 木 (kd-tree) などがある。この次のレジュメで、一番単純な一様格子を使って、レンダリングを高速化しよう。

5 プログラムについて

MeshLoader クラスの computeSmoothNormals 関数には、次のような記述がある。

MeshLoader::computeSmoothNormals 関数

```
void MeshLoader::computeSmoothNormals(int nVerts, const vec3 *verts,
    vec3 *normals, int nTris, const unsigned int *tris) const
{
    vec3 *faceNormals = new vec3[ nTris ]; // 各三角形の面法線ベクトルを格納

    // 各頂点について、その頂点を含む三角形のインデックスを格納
    vector<int> *vertexFace = new vector<int>[ nVerts ];

    // computing face normals and areas
    for (int tri=0; tri<nTris; tri++)
    {
        // TODO: fill here
    }

    // computing vertex normals by averaging face normals
    for (int v=0; v<nVerts; v++)
    {
        // TODO: fill here
    }

    delete [] faceNormals;
    delete [] vertexFace;
}
```

この関数の引数について、三角形メッシュの nVerts は頂点数、verts は頂点座標の配列、normals は法線ベクトルの配列、nTris は三角形の数、tris は各三角形を構成する頂点のインデックス (0 から nVerts-1 まで) が 3 つずつ格納された配列である。配列 tris について補足すると、配列の要素数は nTris×3 個であり、i 番目の三角形を構成する頂点のインデックスは tris[3*i], tris[3*i+1], tris[3*i+2] で得られる。

この関数の最初の for ループでは、面ごとの法線ベクトル (ただし正規化は不要) を計算し、faceNormals[] に格納する。それと併せて、tri 番目の三角形を構成する 3 つの頂点インデックス tris[3*tri], tris[3*tri+1], tris[3*tri+2] で指定される頂点に対し、「頂点が tri 番目の三角形に含まれている」という情報を vertexFace[] に登録する。vertexFace[] は vector<int> の配列である。vector<int> への値の登録には、push_back 関数を使う。

2 番目の for ループでは、各頂点について、vertexFace[] からその頂点を含む三角形のインデックスを取り出す。そのインデックスから面の法線ベクトルを faceNormals[] から取り出す。その頂点を含むすべて

の三角形の法線ベクトルを使い、最終的に正規化された頂点の法線ベクトルを計算する。

付録: 三角形メッシュの読み込み

この実験のプログラムでは、三角形メッシュの読み込みを MeshLoader クラスで行なっている。ポリゴンメッシュを記述したファイルフォーマットは様々なものがあるが、この実験のプログラムで読み込めるのは、Wavefront OBJ 形式と PLY 形式の 2 種類である。Wavefront OBJ 形式は昔 Wavefront という会社のモデラーで使われていた形式で、単純なテキストファイルなのでよく利用される。具体的には、一行ごとに、頂点の座標 (v)、面を構成する頂点 (および法線やテクスチャ座標) のインデックス (f)[*4]、法線ベクトル (vn)、テクスチャ座標 (tx) などが記述されている。例えば、単純なサイコロ (立方体) の形状を記述した OBJ ファイルは次のようなものである。

box.obj

```
v 0.500000 0.500000 0.500000
v -0.500000 0.500000 0.500000
v 0.500000 -0.500000 0.500000
v -0.500000 -0.500000 0.500000
v 0.500000 0.500000 -0.500000
v -0.500000 0.500000 -0.500000
v 0.500000 -0.500000 -0.500000
v -0.500000 -0.500000 -0.500000

f 1 2 3
f 4 3 2
f 1 3 5
f 7 5 3
f 1 5 2
f 6 2 5
f 8 6 7
f 5 7 6
f 8 7 4
f 3 4 7
f 8 4 6
f 2 6 4
```

一方、PLY 形式は、Stanford 大学で開発されたフォーマットで、ファイルの先頭に内容についての情報を記述したヘッダ部分があり、その後にデータ部分が続く、というものである。ヘッダ部分に定義を書いておくことで、比較的自由に記録するデータの内容を決めることができる。なお、用途によってはもっと単純化したデータが使われる。また、データをテキスト形式だけでなくバイナリ形式でも記録できるので、ファイルサイ

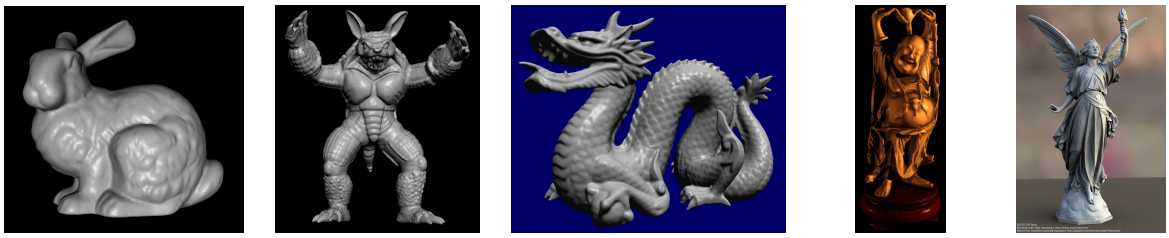
[*4] ただしこのインデックスは 0 からではなく 1 から始まることに注意。

ズの大きなデータでも高速に読み書きが可能である。PLY ファイルのヘッダは例えば次のようになっている。

duck.ply

```
ply
format binary_little_endian 1.0
comment VCGLIB generated
comment TextureFile images/duckCM.jpg
element vertex 2108
property float x
property float y
property float z
property float nx
property float ny
property float nz
element face 4212
property list uchar int vertex_indices
property list uchar float texcoord
end_header
(※これ以降はバイナリデータなのでテキストとしては表示できない)
```

比較的ファイルサイズの大きな形状データとして、Stanford 大学の 3D Scanning Repository[*5] で、3D スキャナを用いて実物をスキャンして得られた 3D データが公開されている。例えば、公開されている Stanford bunny は頂点数が 35947、三角形数が 69451 のデータである。これらの形状データは、CG の研究分野で標準的なテストデータとして用いられていて、Stanford bunny、Stanford buhhda、Stanford dragon などが論文の適用例でよく使われている (図 4)。



bunny　armadillo　dragon　happy buhhda　lucy

図 4　Stanford 大学の 3D Scanning Repository で公開されている形状モデルデータ. 画像は Web サイトから引用.

[*5] URL は http://graphics.stanford.edu/data/3Dscanrep/