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 - ユニフォームグリッドを用いた レイと三角形メッシュの交差判定の高速化 -

金森 由博*

2013 年 2 月 13 日

1　課題

ユニフォームグリッドを用いて三角形の交差判定を高速化しよう。実装するのは以下の部分である。

1. `AxisAlignedBox` クラスの、`hit` 関数および `shadowHit` 関数
2. `UniformGridTraverser` クラスの、`calculateRayBoxIntersection` 関数、コンストラクタおよび `calculateNextIntersectionParameters` 関数
3. `TriangleMeshUniformGrid` クラスの、`hit` 関数および `shadowHit` 関数

2　空間データ構造を用いた交差判定の高速化

レイトレーシングでは、レイと交差判定すべき対象物が増えてくると、計算に時間がかかるようになる。特に、各レイに対してすべての物体と交差判定をする、というような単純な実装であれば、物体の数に比例した計算時間がかかってしまう。前のレジュメの三角形メッシュはまさにこの好例である。このレジュメでは、空間データ構造のうちで最も単純な、**ユニフォームグリッド** (*uniform grid*; 一様格子) を用いて、計算の高速化を図る。まずは空間データ構造の概略について説明する。

2.1　主な空間データ構造

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。これによって交差判定の対象が減り、計算時間を減らすことができる。空間データ構造の例としては、ユニフォームグリッド、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。それぞれ簡単に説明する。

ユニフォームグリッド (図 1a):　一様な大きさの箱 (セル; cell あるいはボクセル; voxel) を格子状に敷き詰めたもの。

* kanamori@cs.tsukuba.ac.jp

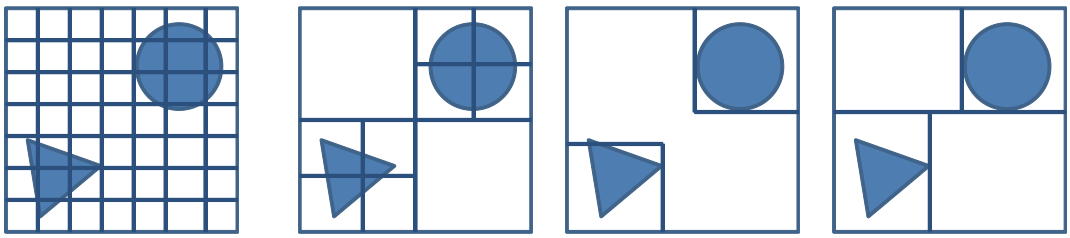
八分木 (Octree) (**図** 1b):　ひとつの箱を平面 $x = 0, y = 0, z = 0$ に平行な面で分割する (つまり 8 分割する)、という操作を繰り返して得られる木構造。

階層的バウンディングボリューム (BVH) (**図** 1c):　各物体をぴったりと囲む箱 (バウンディングボックス; bounding box) を、その箱の中の物体の集合を 2 つに分割し、それらをそれぞれ囲む箱を作り、それらをさらに 2 分割して…という風にして階層的に作られる木構造。

kd 木 (**図** 1d):　空間を平面で繰り返し 2 分割して得られる BSP(Binary Space Partitioning) 木の一種で、x, y, z 軸に垂直な平面で空間を分割する。

これらの空間データ構造がどのように使われるかを説明する。まず、空間を分割し、分割された各領域に、その領域に含まれる物体を登録する。この手順を**ビルド** (build; 構築) という。そして、各レイが分割された領域を通過するたびに、その領域に含まれる物体とだけ交差判定の計算が実行される。この手順はレイの**トラバーサル** (traversal; 巡回) という。空間データ構造は、ビルド時間、トラバーサルの効率 (次の領域に移動するコストや交差判定となる物体の数をいかに減らせるか) やメモリ効率などの点で、長所・短所がある。

ユニフォームグリッドの長所は、ビルド時間が圧倒的に速いことと、実装が容易なことである。また、空間中にびっしりと物体が詰まっているような場合は、トラバーサルやメモリ効率もよい。しかし、物体が広い空間に散らばっているような場合、何もないところにも箱を敷き詰めることになってしまい、空の箱を何度も巡回することでトラバーサルの効率が落ち、メモリも浪費してしまう。八分木は、物体の存在する領域だけ空間を分割し、空の部分は分割しないので、トラバーサルやメモリ効率が改善される。が、その点は BVH や kd 木の方が優れている。BVH と kd 木とでは、一般にビルド時間が速いのが BVH、トラバーサルの効率がよいのが kd 木、とされているが、BVH と kd 木のビルド・トラバーサルを改善する研究が最近盛んに行われていて、どちらがよいとも言い切れない状況である。



(a) ユニフォームグリッド　(b) 八分木　(c) BVH　(d) kd 木

図 1　レイの交差判定の高速化に使われる主な空間データ構造．説明の都合で 2 次元の図として描いている．なお，八分木は 3 次元では 8 分割だが，2 次元では 4 分割なので四分木 (quadtree) と呼ばれる．

3　ユニフォームグリッドによるレイと三角形メッシュの交差判定の高速化

前述のとおり、このレジュメではユニフォームグリッドを用いて、レイと三角形メッシュの交差判定の高速化を図る。以下、ユニフォームグリッドを単にグリッドと書く。グリッドは、3D 空間中のすべての物体を包括するようにして使うこともできるが、物体がまばらに散らばっているようなシーンには向かない。ここでは、三角形メッシュのバウンディングボックスを分割してグリッドとし、格子の各セルに三角形を登録する、ということにする。ビルドとトラバーサルの手順は次のようになる。

■ビルド　(いずれも実装済み)

1. 三角形メッシュのバウンディングボックスを計算する (`TriangleMeshUniformGrid` クラスの `computeBoundingBox` 関数で実装済み)。
2. グリッドの x, y, z 方向の分割数 (あるいはセルの大きさ) を決めて分割する (`UniformGrid` クラスの `init` 関数で実装済み)。
3. 各セルと交差する三角形を、そのセルに登録する (`UniformGrid` クラスの `addObject` 関数で実装済み)。

■トラバーサル

1. レイが三角形メッシュのバウンディングボックスと交差するか調べる。交差しなければ終了。
2. レイの始点がバウンディングボックスの中か外かに応じて、計算を開始するセルを特定する。併せて、セルを順に移動するときに必要なパラメータを計算しておく。
3. レイが通過する各セルで、そのセルに含まれる三角形と、レイの交差判定を行う。もし交点が見つかったら、そのセルの中で一番手前の (レイの始点に近い) 交点を返す。
4. 交点が見つかるかレイがバウンディングボックスの外に出るまで、セルを移動しては 3 を繰り返す。

上記のうち、ビルドはすでに実装が済んでいる (詳細はそれぞれのクラスが定義されたソースコードを参照)。以下では、グリッドの各セルに三角形がすでに登録されているものとして、トラバーサルの方法を検討する。グリッドのセルのトラバーサル処理は、`UniformGridTraverser` クラスの該当する関数に実装する。

3.1　トラバーサルの手順 1: レイとバウンディングボックスとの交差判定

まず粗い判定として、レイがグリッドのバウンディングボックスと交差するかどうかを調べる。この判定は `UniformGridTraverser` クラスの `calculateRayBoxIntersection` 関数に実装する。しかし、いきなりこの関数を実装しても正しくできたか確認しづらい。そこで、まずは箱型の形状 `AxisAlignedBox` [*1] クラスの `hit` 関数および `shadowHit` 関数を実装し、箱がうまく表示されるかどうかを調べることにする。

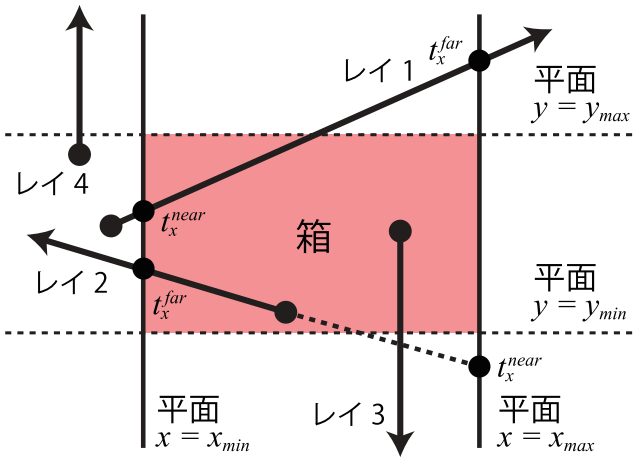


図 2　レイと箱の交差判定. x 方向の計算についてのみ図示. 図を簡単にするため, 2 次元に投影した図としている.

[*1] axis-aligned とは、各辺が x, y, z 軸に沿っている、という意味である。

■レイと箱の交差判定:　処理の手順としては、次の通りである。箱の 6 つの面それぞれとレイが交差するかを調べる方法も考えられるが、有限の大きさの面とレイの交差判定は計算効率が悪い。そこで、箱の各面を含むような、無限に広がる 6 つの平面を対象とする。ここで、箱の頂点のうち、x, y, z 座標がいずれも最小の頂点の座標を $\mathbf{x}_{min} = (x_{min}, y_{min}, z_{min})^T$、$x, y, z$ 座標が最大の頂点の座標を $\mathbf{x}_{max} = (x_{max}, y_{max}, z_{max})^T$ とする (ソースコードでは、`AxisAlignedBox` クラスの `vec3` 型のメンバ変数 `m_Pos[0]` が $\mathbf{x}_{min}$, `m_Pos[1]` が $\mathbf{x}_{max}$ にあたる) [*2]。すると、無限に広がる 6 つの平面とは、平面 $x = x_{min}$, $x = x_{max}$, $y = y_{min}$, $y = y_{max}$, $z = z_{min}$, $z = z_{max}$ である (図 2 参照)。

以下、各平面とレイとが交差するかどうかを判定する。レイの始点を $\mathbf{o} = (o_x, o_y, o_z)^T$, レイの方向ベクトル (単位ベクトル) を $\hat{\mathbf{d}} = (d_x, d_y, d_z)^T$ とする。まず平面 $x = x_{min}$ および $x = x_{max}$ について考えよう。もし平面 $x = x_{min}$ および $x = x_{max}$ とレイが平行 (つまり $d_x = 0$) ならば、レイはこれらの平面とは交差しない (図 2 のレイ 3 およびレイ 4)。この場合、レイが箱と交差するためには、レイの始点の x 座標 o_x が、平面 $x = x_{min}$ と $x = x_{max}$ の間に入っていなければならない (図 2 のレイ 3)。もしレイが各平面と平行でなければ、レイはどこかで平面と交差する (図 2 のレイ 1 およびレイ 2)。そこで、平面 $x = x_{min}$ と $x = x_{max}$ で挟まれた領域と、レイが交差するときの、レイのパラメータ t の区間を調べる。これは、レイの始点の x 成分と方向ベクトルの x 成分を用いて計算できる。レイが平面 $x = x_{min}$ と $x = x_{max}$ のうちどちらと先に交差するかは、レイの方向ベクトルの向きによる。レイの始点から見て交点でのレイのパラメータがより小さい方のパラメータを t_x^{near}, より大きい方のパラメータを t_x^{far} とすると、これらは次の式で求められる。

$$\begin{cases} d_x > 0 \text{ の場合}: \ t_x^{near} = \frac{x_{min} - o_x}{d_x}, \ t_x^{far} = \frac{x_{max} - o_x}{d_x} \\ d_x < 0 \text{ の場合}: \ t_x^{near} = \frac{x_{max} - o_x}{d_x}, \ t_x^{far} = \frac{x_{min} - o_x}{d_x} \end{cases} \tag{1}$$

レイが 2 つの平面に挟まれた領域と交差するのは、区間 $[t_x^{near}, t_x^{far}]$ の間である。ここまでの処理を、平面 $y = y_{min}$ と $y = y_{max}$ で挟まれた領域、平面 $z = z_{min}$ と $z = z_{max}$ で挟まれた領域についても計算すると、それぞれレイが交差する区間 $[t_y^{near}, t_y^{far}]$ および $[t_z^{near}, t_z^{far}]$ が求まる。ここでレイと箱の交差判定の話に戻ると、箱というのは、平面 $x = x_{min}$ と $x = x_{max}$ で挟まれた領域、平面 $y = y_{min}$ と $y = y_{max}$ で挟まれた領域、そして平面 $z = z_{min}$ と $z = z_{max}$ で挟まれた領域の、共通部分である。ということは、レイのパラメータの区間 $[t_x^{near}, t_x^{far}]$, $[t_y^{near}, t_y^{far}]$ および $[t_z^{near}, t_z^{far}]$ の 3 つが共通部分を持てば、レイは箱と交差する。3 つの区間のひとつでも他の 2 つの区間と共通部分がなければ、レイは箱と交差しない。

面の法線ベクトルについては、レイがどの面と交差するかがわかれば、x, y, z 軸のいずれかに平行な単位ベクトルを設定できる。例えば、図 2 のレイ 1 は、レイの方向が $+x$ 向きで平面 $x = x_{min}$ と交差するので、交点での法線ベクトルは $(-1, 0, 0)^T$ である。なお、レイの始点が箱の内側にある場合も、法線ベクトルは箱の外向きのものを選ぶ。例えばレイ 2 についても交点での法線ベクトルは $(-1, 0, 0)^T$ とする。この理由は、もし箱がガラスのような材質であった場合に、光の屈折を計算するときに内外判定に必要だからである。

プログラムとしては、x 方向、y 方向、z 方向の順にレイのパラメータの区間 $[t^{near}, t^{far}]$ を計算し、計算するたびに区間を狭めていって、共通部分が残れば「交差する」、残らなければ「交差しない」、と判定すればよい。`AxisAlignedBox` の `hit` 関数の擬似コードは次の通りである。

AxisAlignedBox.cpp

```
bool AxisAlignedBox::hit(const Ray &r, Real tmin, Real tmax, HitRecord &record)
```

[*2] $\mathbf{x}_{min}, \mathbf{x}_{max}$ を行ベクトルのように横向きに書いているが、実際は列ベクトルであることを示すために転置を表す T をつけている。

```
 const
{
    // レイのパラメータ tnear, tfar をそれぞれ tmin, tmax で初期化

    // x 方向について
    // レイの方向ベクトルの x 成分 dx について
    //     もし dx == 0 なら
    //         レイの始点の x 座標 ox が区間 [xmin, xmax] の間になければ false を返す
    //     そうでなければ (dx != 0 なら)
    //         平面 x = xmin, x = xmax と交差するときのパラメータ txnear, txfar を計算
    //         もし tnear < txnear なら
    //             平面 x = xmin, x = xmax のうち、txnear に対応する平面を記録
    //             tnear を txnear で更新
    //         もし tfar > txfar なら
    //             tfar を txfar で更新
    //         もし tnear > tfar なら
    //             false を返す

    // y 方向および z 方向についても同様に計算

    // どの平面と交差したかに応じて法線を計算、その他 record に必要な情報を計算

    // true を返す
}
```

プログラミングできたら、箱を表示するシーンファイルを読み込んで、レイトレーシングの結果が OpenGL での表示と同様になるかを確認する。

■UniformGridTraverser クラスの calculateRayBoxIntersection 関数への移植: AxisAlignedBox の hit 関数および shadowHit 関数の実装が完成したら、UniformGridTraverser クラスの calculateRayBoxIntersection 関数に移植する。この関数で計算する必要があるのは、交差があったかどうか (返り値)、交点でのレイのパラメータ (第 1 引数) である。AxisAlignedBox の hit 関数から移植するとき、不要な処理は適宜削除する。

3.2 トラバーサルの手順 2: トラバーサルを開始するセルの特定とパラメータの計算

グリッドのトラバーサルをどこから開始するかは、レイの始点が、グリッドの内側にあるか外側にあるかによって異なる。もしグリッドの内側にあれば、レイの始点を含むセルから、トラバーサルを開始する。一方、もしグリッドの外側にあれば、バウンディングボックスでの交点を含むセルから開始する (図 3 参照)。現在のセルは、整数値のインデックス (m_CurrentX, m_CurrentY, m_CurrentZ) で識別される。ある点を含むグリッドのセルのインデックスを調べるには、UniformGrid クラスの getIndex 関数を使えばよい。

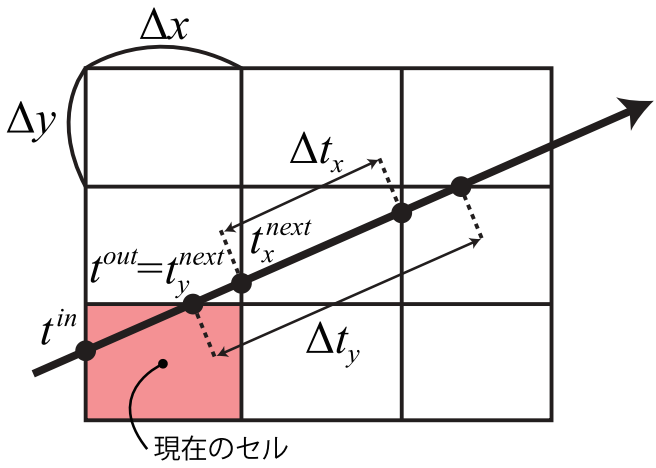


図 3　トラバーサルに必要なパラメータの計算. 図を簡単にするため，2 次元に投影した図としている.

UniformGrid.h

```
// 入力: 問い合わせしたい点の 3 次元座標 p
// 出力: グリッドのセルのインデックス xi, yi, zi
inline void getIndex(const MyAlgebra::vec3 &p, int &xi, int &yi, int &zi);
```

レイのトラバーサルでは、現在のセルに含まれる三角形と交差判定したあと、もし交差がなければ、隣接する上下左右前後のセルのいずれかにひとつ進む (プログラムでは `m_CurrentX`, `m_CurrentY`, `m_CurrentZ` のいずれかが更新される)。なお、正の方向に進むか負の方向に進むかは、レイの方向ベクトルによる (プログラムでは `m_IncrementX`, `m_IncrementY`, `m_IncrementZ` のそれぞれに＋ 1 か− 1 の値が入る)。このとき、次にどのセルに進むかをレイの方向ベクトルなどからいちいち計算していたのでは、大変効率が悪い。そこで、次のように考える。レイがグリッドのセルとぶつかるのは、x の壁 (平面 $x = 0$ に平行な平面)、y の壁、z の壁のいずれかである。x, y, z それぞれの壁の間隔 Δx, Δy, Δz (セルの x, y, z 方向の幅) は一定であり、レイのパラメータに換算してどれだけの間隔であるかを計算しておくことができる。レイの方向ベクトル (単位ベクトル) が $\hat{\mathbf{d}} = (d_x, d_y, d_z)^T$ だとすると、レイのパラメータに換算して x, y, z の壁の間隔は $\Delta t_x = |\Delta x/d_x|$, $\Delta t_y = |\Delta y/d_y|$, $\Delta t_z = |\Delta z/d_z|$ (プログラムではそれぞれ `m_DeltaTX`, `m_DeltaTY`, `m_DeltaTZ`) となる。レイの始点から出発して、次に x, y, z の壁とぶつかるときのレイのパラメータをそれぞれ t_x^{next}, t_y^{next}, t_z^{next} とする (プログラムでは `m_NextTX`, `m_NextTY`, `m_NextTZ`)。レイが次に x, y, z のどの方向に進めばよいかは、x, y, z のうちの始点に一番近い壁を選べばよい。つまり、t_x^{next}, t_y^{next}, t_z^{next} のうち一番小さいパラメータに対応する壁を選ぶ。その壁を選んだら、その壁に対応するレイのパラメータ (つまり t_x^{next}, t_y^{next}, t_z^{next} のいずれか) に、レイのパラメータに換算した壁の間隔 (つまり Δt_x, Δt_y, Δt_z のいずれか) を足す。このようにして、隣接するセルに移動するアルゴリズムを、3DDDA (3D Digital Differential Analyzer) という (図 4)。このアルゴリズムは、2D で線分を効率よく描画する Bresenham のアルゴリズムに似ている。しかし、Bresenham のアルゴリズムは線分の見た目を近似するだけが目的なので、線分がセルとわずかに交差していてもセルが塗りつぶされないことがあるのに対し、3DDDA ではレイが少しでも交差したセルは計算対象となる。

以上をまとめると、プログラムでは、トラバーサルを開始する前に、次の変数を計算しておく。

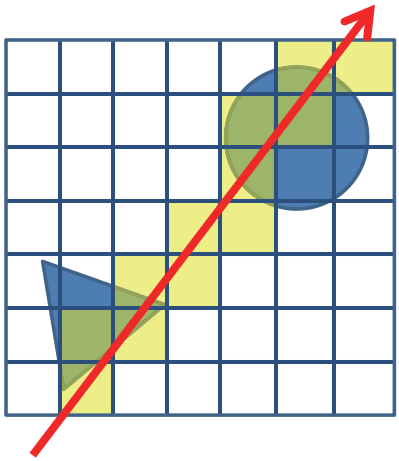


図 4　3DDDA (3D Digital Differential Analyzer). 黄色いセルが交差判定の対象となるセル. 図を簡単にするため，2 次元に投影した図としている.

1. m_CurrentX, m_CurrentY, m_CurrentZ：トラバーサルを開始するセルのインデックス
2. m_DeltaTX, m_DeltaTY, m_DeltaTZ: x, y, z 方向のレイが壁とぶつかる間隔
3. m_IncrementX, m_IncrementY, m_IncrementZ: 隣接するセルのうち上下左右前後のどの方向に進むかの増分（+1 あるいは-1）
4. m_NextTX, m_NextTY, m_NextTZ：レイが次に x, y, z の壁とぶつかるときのパラメータ

上記の計算のうち 1, 2 および 3 は UniformGridTraverser クラスのコンストラクタ (UniformGridTraverser という名前の返り値のない関数) で行い、4 は calculateNextIntersectionParameters 関数の中で行う。なお 1 は実装済みなので、自分で実装するのは 2, 3 および 4 である。

3.3　トラバーサルの手順 3: セル内の三角形とレイの交差判定

各セルでは、そのセルに格納されたすべての三角形と交差判定を行う。もし三角形とレイとの交点が見つかった場合、その交点でのレイのパラメータ t が、いま検査しているセルの入り口および出口でのパラメータ t^{in}, t^{out} (図 3 参照) の間にあることを確認する必要がある (t^{in}, t^{out} は プログラムではそれぞれ getCurrentCellTMin 関数および getCurrentCellTMax 関数を呼ぶことで得られる)。$t^{in} \leq t \leq t^{out}$ なら、交点が見つかったとして、トラバーサルを終了する。このチェックをしないと、場合によってはレイの始点から一番近いはずの交点よりも先に別の交点が見つかってしまい、レンダリング結果に不具合が発生する。

この計算は、後述する通り、TriangleMeshUniformGrid クラスの hit 関数および shadowHit 関数で実装する。

3.4　トラバーサルの手順 4: セルの移動

交点が見つかるかレイがバウンディングボックスの外に出るまで、セルを移動しては手順 3 を繰り返す。現在のセルから次に $\pm x, \pm y, \pm z$ 方向のどのセルに移動するかは、上記の手順 2 で述べた通りである。つまり、レイが次に x, y, z の壁とぶつかるときのレイのパラメータ m_NextTX, m_NextTY, m_NextTZ のうち、一番小さいパラメータに対応する壁を選ぶ。そして、次の変数を更新する。

- レイが現在のセルの交点のうち、手前の (入り口での) レイのパラメータ m_CurrentTMin
- m_NextTX, m_NextTY, m_NextTZ のうち、該当するもの
- m_CurrentX, m_CurrentY, m_CurrentZ のうち、該当するもの

これらの計算は、UniformGridTraverser クラスの advanceToNext 関数の中で行う (実装済み)。

4 TriangleMeshUniformGrid クラスの実装

ここまでの手順で、グリッドのトラバーサルを行うための UniformGridTraverser クラスの実装が終ったはずである。あとは、UniformGridTraverser クラスのインスタンスを、三角形メッシュを保持している TriangleMeshUniformGrid クラスの、hit 関数および shadowHit 関数の中で呼び出して使うことになる。hit 関数および shadowHit 関数の計算の流れは次の通りである。

1. レイのトラバーサルに必要な情報を計算 (UniformGridTraverser クラスのコンストラクタ内で計算される)
2. グリッドのバウンディングボックスの内部にある (isInsideGrid 関数で判定) 間、以下を計算
3. 現在のセルに含まれるすべて三角形のリストを取得 (getCurrentCell 関数を使う)
4. 三角形と交差判定し、交点でのレイのパラメータが t^{in}, t^{out} の間であれば、交点が見つかったとしてトラバーサル終了
5. 交点がなければ、次のセルに進む (advanceToNext 関数を呼び出す)

ここまで実装できたら、設定ファイルを読み込んでテストする。三角形メッシュのシーンファイルに、以下の項目を追加する。

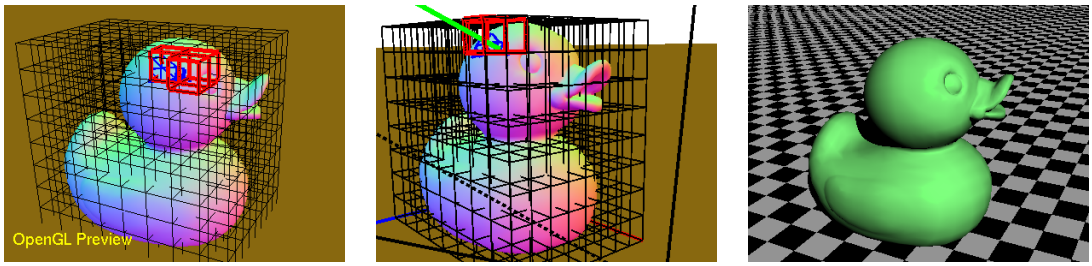
mesh_test.cfg

```
triangle_mesh
{
    /* 略  */

    grid_acceleration = true;
    grid_resolution = (8,8,8);

    /* 略  */
}
```

grid_acceleration はグリッドによる高速化のオン/オフを切り替える。grid_resolution はグリッドの x, y, z 方向の分割数である。最適な分割数は、三角形メッシュ内の三角形の大きさによって異なる。入力された三角形メッシュごとに調整してほしい。

トラバーサルのデバッグは煩雑なので、デバッグをサポートする機能を用意している (図 5)。レイトレーシングの結果を表示するウィンドウ (”Rendered Result”というタイトルがついているウィンドウ) 上で、Shift キーを押しながらマウスを左クリックすると、そのクリックしたピクセルを通過するレイについて、レイが巡回するグリッドのセルが、”World View” というタイトルのウィンドウで赤いセルとして表示される (図 5b)。

この機能を使って、正しくトラバーサルが行われているか確認できる。



(a) プレビュー画面　　(b) 別視点表示　　(c) レイトレーシング結果

図 5　グリッドトラバーサルのデバッグ．(a) OpenGL のプレビュー画面で Shift キーを押しながら左クリックすると，レイが通過するグリッドのセルが赤く表示される．(b) ワールド座標系のカメラからみた別視点からの表示．(c) レイトレーシング結果．