Simple Collision: Triangle with Vector

Yasuhiro Yamazaki
m5041134@u-aizu.ac.jp

平成14 年 2 月 6 日

目 次

1　この文章は

自由に改編して再配布、コピーすることを許可します。間違いがあれば指摘して下さいませ。説明にはサンプルプログラムについても多少記述されていますので、配布する際はなるべくサンプルプログラムも梱包してください。

2　衝突判定とは

ここで言う衝突判定とは、ある物体と物体が互いに重なる領域を持つ状態を検出することを言います。物体と物体が衝突しているかどうかを検知するための計算だけではなく、多数の物体同士を効率良く検索するアルゴリズムを含め、様々な方法が web を始め多く文章化されて紹介されていますが、このドキュメントでは、その中で最も単純であろう「ベクトルと三角形ポリゴン」について記述します。

3　ベクトルの定義

空間における線分をベクトルで表現すると式 (1) のようになります。

$$\overrightarrow{V}_e = t\overrightarrow{V}_d + \overrightarrow{V}_s \tag{1}$$

$\overrightarrow{V}_s$ を開始点として、そこから t で示される倍数分だけ $\overrightarrow{V}_d$ が伸縮します。これによって 2 点間の線分を示すことができ、空間にある線分を自由に表現できます。
まとめると、このベクトルは、始点が $\overrightarrow{V}_s$ で、終点が $\overrightarrow{V}_e$ で表現できます。

4　三角形の定義

三角形は、空間のベクトル 3 つで表現できることから、式 (2) のように 3 点の集合で表記してしまいます。

$$(\overrightarrow{v}_0, \overrightarrow{v}_1, \overrightarrow{v}_2) \tag{2}$$

それぞれが三角形の頂点を示していますので、辺の集合にしたい場合は、式 (3) と表記します。

$$(\overrightarrow{e}_0, \overrightarrow{e}_1, \overrightarrow{e}_2) = (\overrightarrow{v}_1 - \overrightarrow{v}_0, \overrightarrow{v}_2 - \overrightarrow{v}_1, \overrightarrow{v}_0 - \overrightarrow{v}_2) \tag{3}$$

4.1 面を計算する

三角形は面と違い、有限な平面です。この三角形を包括する面の式を計算することは簡単です。後で、まずベクトルがこの面を貫いている可能性があるかどうかを検査する際に利用できますので、あらかじめ面を計算しておくことにします。面の式は式(4)のように定義されます。

$$\overrightarrow{N} \cdot \overrightarrow{X} = c \tag{4}$$

$\overrightarrow{N}$ を法線として、内積が常に d となるような点 $\overrightarrow{X}$ の集合が面になります。逆に言うと、$\overrightarrow{X}$ は面上の点を示しています。ここで言えば、三角形の頂点 $\overrightarrow{v}_0$ などは全て面上の点の代表として使えます。

後のためにもう少し補足しますと、移項された式(5)のように c を引き算してゼロになる $\overrightarrow{X}$ が平面なわけですから、式(6)や式(7)のように、ゼロにならないような $\overrightarrow{X}$ は面の法線方向にある点かその逆方向にある点どちらかを示していることになります。

$$\overrightarrow{N} \cdot \overrightarrow{X} - c = 0 \tag{5}$$

$$\overrightarrow{N} \cdot \overrightarrow{X} - c > 0 \tag{6}$$

$$\overrightarrow{N} \cdot \overrightarrow{X} - c < 0 \tag{7}$$

さて、それでは三角形の情報から面を計算してみましょう。まず法線を計算します。三角形を代表する2辺である $\overrightarrow{e}_0$ と $\overrightarrow{e}_1$ を利用して外積を用いて式(8)算出できます。分母は単位ベクトル化する為に必要です。

$$\overrightarrow{N} = \frac{\overrightarrow{e}_0 \times \overrightarrow{e}_1}{|\overrightarrow{e}_0 \times \overrightarrow{e}_1|} \tag{8}$$

さて、この式(8)と $\overrightarrow{X}$ と内積をとれば、c が計算できますが、$\overrightarrow{X}$ は面上の点であることが求められています。先程、三角形の頂点は全て面上の代表点として使えることを述べました。従って、式(9)のように $\overrightarrow{v}_0$ を利用してやれば計算できます。

$$c = \overrightarrow{N} \cdot \overrightarrow{v}_0 \tag{9}$$

これで面の情報は全て揃いました。

5 衝突を計算する手順

衝突を計算する手順は以下のようになります。

- ベクトルが面を貫いているか否か
  これをここで最初に調べることによって、後の全く関係無いベクトルとの計算を省くことができます。
- 面上で衝突しているの点を計算
  ここで面上の点を算出します。
- その点が三角形の中にあるか
  面上にあった衝突点が、三角形の枠内に入っているかどうかを調べます。
- 衝突点の処理
  これは任意の処理ですが、いくつか例を挙げておきます。

5.1 ベクトルが面を貫いているか否か

これは与えられたベクトルに対して式(6)や式(7)を利用することで判別できそうです。要するに、符号だけでそのベクトルが面の表か裏のどちらにあるかを確定することは出来ませんが、符号が同じか違うかを見ることで、面の表か裏の一方にあるであろう事は確定できます。すなわち、ベクトルの始点と終点それぞれで計算し、符号の相違を見る事で、少なくともそのベクトルが面を貫通している事が言えます。ベクトルの始点は式(1)にあるように、$\overrightarrow{V}_s$で、終点は$\overrightarrow{V}_e$なので、始点の式(10)と終点の式(11)それぞれの計算結果w_sとw_eの符号を見てやります。

$$w_s = \overrightarrow{N} \cdot \overrightarrow{V}_s - d \tag{10}$$

$$w_e = \overrightarrow{N} \cdot \overrightarrow{V}_e - d \tag{11}$$

同符号の乗算は正に、異符号の乗算は負になることを利用し、式(12)に示す通りの場合分けで書き表せます。

$$\begin{cases} (A) & w_s w_e > 0 \\ (B) & w_s w_e = 0 \\ (C) & w_s w_e < 0 \end{cases} \tag{12}$$

ここでは 3 つの場合分けが出来ますが、(A) は、「始点終点とも片面一方に存在している (すなわち貫通していない)」と言う状態です。(B) は、「少なくとも始点終点のどちらか一方が面の上に乗っている」状態で、(C) は、「始点終点がそれぞれ面を境界にして一方ずつに存在している (すなわち貫通している)」状態になります。

衝突していると判定できるのはこの内 (B) と (C) ですが、始点か終点が完全に面の上に乗っている場合は稀なケースです。接触は衝突していないと見ることもできるでしょう。しかし、この衝突判定ルーチンが利用される次のシーンを考えてみます。まず、点がベクトル分だけ直進するアルゴリズムで動いていたとします。さらに、その途中で何らかの三角形をベクトルが貫通したと過程しましょう。ベクトルは終点を衝突点として停止することになりますが、継続して直進するベクトルを加算した際に、次回の点の位置は面の上になってしまいます。この時、やはり式 (12) による衝突のチェックで「一方 (この場合は始点) が面の上にある」状態、場合分けにある (B) が再び現れます。(B) を衝突していないとしてあるわけですから、点は再びベクトルを加算して面を通り抜けてしまうでしょう。

ではどうすれば良いのか。まず 1 つ目は (B) のケースを衝突すると定義してしまうことです。これによって貫通してしまう事が無くなります。2 つ目は ベクトルの終点を面の接触点と同値にしないことです。これはどう言うことかと言うと、次回そのベクトルを利用して面の上に来ないように、少しベクトルを短く修正する必要があると言うことです。これにより稀なケースが発生するのをなるべく少なくしようと言う目的があります。このある程度短くする長さを ε としましょう。例えば、プログラムではこのサイズを 式 (13) のように適当に定義しておきます。

$$\varepsilon = 0.001 \tag{13}$$

5.2 面上で衝突している点の計算

さて、前節で衝突していると判定できたベクトルについては、その面とベクトルとの衝突点を正確に求める必要があります。式 (1) の式にある t を変化させることで、ベクトルを伸縮させ、その先が面の上にくるような値を探す方程式を立てることで解けます。衝突点を示す t_c を算出して出来た新しいベクトルを 式 (14) で示します。

$$\overrightarrow{V}_c = t_c \overrightarrow{V}_d + \overrightarrow{V}_s \tag{14}$$

このベクトルが面の上を示すわけですから、式 (4) に代入し、式 (15) を作ることで式 (16) のように解けます。

$$\overrightarrow{N} \cdot \overrightarrow{V}_c = d \tag{15}$$

$$\overrightarrow{N} \cdot (t_c \overrightarrow{V}_d + \overrightarrow{V}_s) = d$$

$$t_c \overrightarrow{N} \cdot \overrightarrow{V}_d + \overrightarrow{N} \cdot \overrightarrow{V}_s = d$$

$$t_c = \frac{d - \overrightarrow{N} \cdot \overrightarrow{V}_s}{\overrightarrow{N} \cdot \overrightarrow{V}_d} \tag{16}$$

ここで得られた t_c を、式(14)に代入することで、面上にある衝突点を取得することができます。

5.3 その点が三角形の中にあるか

さて、前節で取得された衝突点はあくまでも面とベクトルとの交点ですので、その点が式(2)で示された目的の三角形の領域内にあるかどうかを調べなくてはいけません。そこで利用できるのが外積の持つ特性です。三角形は、それ以上の多面体(四角形など)と違い、必ず常に凸な図形である事にも注目します。図1に示されるように、衝突点が三角形内部にある場合は、各頂点から衝突点までのベクトルと、頂点から次の頂点までのベクトル(辺)との外積を取って出来たベクトルの向きが面に対して表を向くか裏を向くかのどちらかで全て揃います。衝突点が三角形の外に来ると、どこかの辺で外積されて出来たベクトルの向きが逆になります。

では頂点から衝突点までのベクトルと、頂点から次の頂点までのベクトルの外積を取って、それが表か裏かを示すにはどうしたら良いでしょうか。そこに面の法線を利用します。面の法線 $\overrightarrow{N}$ と内積を取り、その符号を調べることで、法線の向きに対してそのベクトルとの開きが判別できます。正であれば、法線方向と90度以内の開きですし、負であれば90度より大きいの開きがあるわけですから、法線方向を表と言うのであれば、そのベクトルは面の裏側を向いていることが言えます。そこで式(17)を作り、それぞれを計算します。

$$\begin{cases} s_0 = \overrightarrow{N} \cdot ((\overrightarrow{V}_1 - \overrightarrow{V}_0) \times (\overrightarrow{V}_c - \overrightarrow{V}_0)) \\ s_1 = \overrightarrow{N} \cdot ((\overrightarrow{V}_2 - \overrightarrow{V}_1) \times (\overrightarrow{V}_c - \overrightarrow{V}_1)) \\ s_2 = \overrightarrow{N} \cdot ((\overrightarrow{V}_0 - \overrightarrow{V}_2) \times (\overrightarrow{V}_c - \overrightarrow{V}_2)) \end{cases} \tag{17}$$

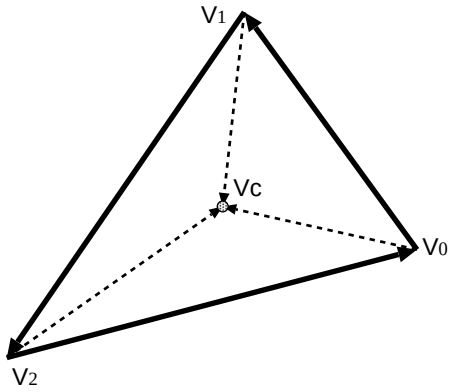

図 1: 三角形内部に衝突点 $\vec{V_c}$ がある場合

式 (17) の s_n の符号が全て同じであることを判定する為に、式 (18) を使います。

$$\begin{cases} \text{範囲外} & s_0 s_1 < 0 \text{ または } s_0 s_2 < 0 \\ \text{範囲内} & \text{上 2 つに当てはまらない場合} \end{cases} \tag{18}$$

いくつかのステップを踏んで来ましたが、これでやっと衝突点と衝突したかどうかの 2 つを知る事ができるようになりました。

5.4 衝突点の処理

さて、これだけでも最初に掲げた衝突判定としては十分だと思いますが、ここで衝突点を少し加工する処理を入れる事でもう少し柔軟なベクトルの扱いが出来るようになりますので、いくつか例を挙げておきます。

衝突点は三角形のポリゴン面上に乗っているため、このまま補正されたベクトルとして、空間における点の移動などに加算するなどして単純な利用をすると、次回点の位置が面の上に来てしまうため、衝突判定上、少々稀なケースとなり不都合が出る場合もありそうです。そこで、衝突点を 式 (13) で示したように ε を使って少々ベクトルの始点に近づけ、面から浮いた場所にずらしてやる必要があります。

5.4.1 始点から衝突点のベクトルを短くする方法

これは始点 $\vec{V}_s$ から衝突点 $\vec{V}_c$ までのベクトル $t_c \vec{V}_d$ の長さを 式 (19) で計算し、そこから ε 分引き算して 式 (20) のように長さを再設定するだ

けです。

$$n = |t_c \overrightarrow{V}_d| \tag{19}$$

$$\overrightarrow{V}_n = (n - \varepsilon)\frac{t_c \overrightarrow{V}_d}{n} \tag{20}$$

この手法で補正されたベクトルを移動にすると、点が三角形に衝突した時点で見かけ上はピタっと停止します。

5.4.2 法線を使って面から離す方法

次に衝突点 $\overrightarrow{V}_c$ から面の法線 $\overrightarrow{N}$ を使って持ち上げる方法。これは式 (21) に示す通り、最も計算が単純ですが、面の法線方向を表とすると、裏側からの衝突の際に突き抜けてしまう欠点があります。

$$\overrightarrow{V}_n = \varepsilon \overrightarrow{N} + \overrightarrow{V}_c \tag{21}$$

解決策としては、式 (10) で計算された結果を用いて、始点と面の法線との位置関係を考慮して法線ベクトルに負を掛算して逆向きに調整する方法などが考えられます。サンプルプログラムでは衝突点の補正アルゴリズムの一つにこの方法を採用しています。

5.4.3 終点をスライドさせる方法

最後に比較的自然な動きに補正する手法を示しておきます。終点 $\overrightarrow{V}_e$ と平面までの垂直な距離を m_0 とした時、終点 $\overrightarrow{V}_e$ を法線 $\overrightarrow{N}$ 方向に $m_0 + \varepsilon$ の分だけ補正し、面から少し浮いた位置までずらされた $\overrightarrow{V}_n$ を算出します。イメージとしては 図 2 にある通りです。

m を 式 (22) にあるように定義すると、図中のパラメータを使って $\overrightarrow{V}_n$ は 式 (23) に示すように書き表せます。

$$m = m_0 + \varepsilon \tag{22}$$

$$\overrightarrow{V}_n = \overrightarrow{V}_e + m\overrightarrow{N} \tag{23}$$

これもサンプルプログラムに実装されているアルゴリズムで、面の上を滑べるような補正をベクトルに施せます。

欠点は、図中にあるように、ずれた点 $\overrightarrow{V}_n$ が必ずしも三角形の上にあるとは限らないことです。これによって、複雑に入り組んだ三角形群に

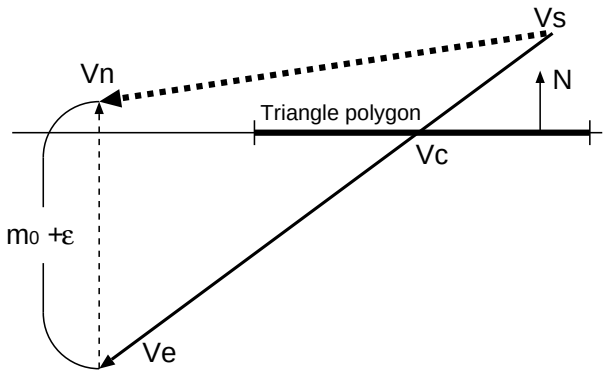

図 2: 終点 $\vec{V}_e$ をずらす場合のイメージ

おいては、補正されたベクトル $\overrightarrow{V}_n$ を再び使ってまた衝突判定を行う必要があります。これを衝突が無くなるまで繰り返すことで、ベクトルは入り組んだ三角形群の中にでいつか必要な長さに補正されることが期待できます。もちろん無限ループになるパターンがありますので、ある程度ループしたら脱出するようにしておいて、別の補正アルゴリズムに切替える必要があるかも知れません。気を付けてください。

さて、計算方法ですが、終点 $\overrightarrow{V}_e$ と面までの距離 m_0 を算出しなくてはいけません。その式は 式 (24) を 式 (4) に代入することで作り出せます。

$$\overrightarrow{V}_{n0} = \overrightarrow{V}_e + m_0 \overrightarrow{N} \tag{24}$$

代入後に計算された式は 式 (25) に示します。

$$\overrightarrow{N} \cdot \overrightarrow{V}_{n0} = d$$

$$\overrightarrow{N} \cdot (\overrightarrow{V}_e + m_0 \overrightarrow{N}) = d$$

$$\overrightarrow{N} \cdot \overrightarrow{V}_e + m_0 \overrightarrow{N} \cdot \overrightarrow{N} = d$$

$$m_0 = d - \overrightarrow{N} \cdot \overrightarrow{V}_e \tag{25}$$

この計算結果と 式 (22) より、式 (23) が作れます。

6　終わりに

ここで挙げた例は最も簡単な衝突を検出する計算です。サンプルプログラムはこの計算をベースに組まれています。実際使ってみると分かるのですが、この衝突判定は使えそうで実際の用途にはあまり使えません。例えば、部屋のウォークスルーの場合、部屋モデルの中を歩き回るわけですが、カメラ位置からの移動ベクトルとモデルの衝突判定をしてやることで作れそうな気がします。しかし実際はカメラ位置は点ですから、壁に衝突した際にどうしてもカメラの位置が壁沿いに来てしまい不自然な結果になります。こんな場合には範囲のある球と三角形の衝突などが有効です。その衝突アルゴリズムについて熟考してみるのも楽しいかも知れませんので、是非挑戦してみてください。

また、衝突を検査したいモデルが複雑すぎて、ポリゴンの数が膨大である場合、それらポリゴンと全て検査するのは効率的ではないと思う方がいらっしゃるかも知れません。そんな場合に様々なアルゴリズムが考案されています。簡単なもので Octree、少し複雑ですが、Binary Space Partition (BSP) などが使えると思います。これらは空間を分断して木の構造を作り出し、空間内のポリゴン検索を効率的にする事ができます。実装は大変ですが、興味のある人は挑戦してみてください。