ESライブラリ＆＆
ゲームプログラミング

○　３Ｄ編 - 第１４回　衝突検出の基本　○

衝突検出

・３次元空間の衝突検出には、数学を利用したさまざまな方法が考案されている
・ベクトルの基本的な考え方や内積と外積を応用した境界球、境界ボックス、レイなど

概要

ゲームでは、キャラクター同士の衝突を検出する処理(Collision Detection:衝突検出、衝突判定、当たり判定、ヒットチェックなど)は、頻繁に必要となります。一般的なシューティングゲームを見ても、自機と敵、弾(自機)と敵、自機と弾(敵)など、１フレーム中に、さまざまな種類の衝突検出が行われます。さらに、それらが１秒間に６０回の移動が行われるとすると、そのぶんだけ衝突検出の回数が増えます。

ゲームのようにリアルタイム性を追求する場合、キャラクターの複雑な形状を完全に考慮した衝突を検出しようとするとたいへんです。２次元の場合には、バウンディングボックス(Bounding Box:衝突範囲)という矩形やバウンディングサークルと呼ばれる円といった架空の領域を用います。この領域と、もう一方のキャラクターの領域の交差判定を行うことで、擬似的に衝突を検出します。

３次元でも衝突検出の基本は同じで、２次元の衝突検出を応用した方法が使われています。

二点間の距離

２次元での二点間(x0, y0)と(x1, y1)の距離Dは、ピタゴラスの定理から

$D = sqrt((x0 - x1)^2 + (y0 - y1)^2)$　　※sqrtは平方根(√)を表します

という計算式で求めることができます。３次元の場合は、ｚを加えたものとなります。

$D = sqrt((x0 - x1)^2 + (y0 - y1)^2 \underline{+ (z0 - z1)^2})$

ベクトルの正規化

ベクトルの正規化とは、ベクトルの長さを１にすることです。照明演算などでは計算式の性質上、正規化したベクトルでなければ正しい結果にならない場合があります。また、正規化することにより、別の用途に使えるものもあります。たとえば内積です。正規化した２つベクトルの内積は、２つのベクトルのなす角のcosと同じ値になります。本来は、２つのベクトルの角度を求め、そこからcosを求めなければなりませんが、そのような手順を踏まなくても、正規化をして内積を求めるだけよくなります。
さらに、正規を行うとベクトルの成分を±１の範囲に制限することにもなるので、プログラムの最適化が行いやすくなります。

内積

内積(inner product)は以下のように定義されています。内積を表す記号が"・"のためドット積や、結果が数値になるためスカラー積とも呼ばれます。

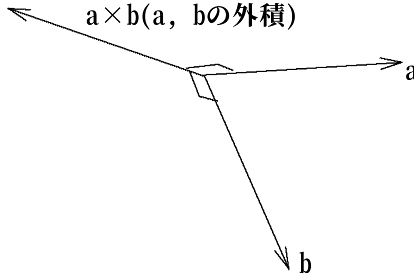
ベクトルa, bの内積＝ $a \cdot b = |a||b|\cos\theta$

内積は、「２つのベクトルの角度を考慮して掛け合わせたときのベクトルの長さ」を求めるもの、２つのベクトルが「どのぐらい似ているか」を示すものといえます。２つのベクトルがまったく同じ場合は１、垂直の場合は０、正反対を向いている場合は－１になります。
内積は、２つのベクトルの始点をそろえた場合の織りなす角度(のcos)が求められます。この値が正なのか負なのかで、「２つのベクトルは鈍角か鋭角か」がわかります。たとえば、面の法線と組み合わせることにより、面の中に点が含まれているかどうか、モデルの面に飛ばしたレイが当たるかどうか、ライトを当てた場合に陰になるかどうかなど、多くの場面で利用できます。
また、ある位置(位置ベクトル)から平面(直線)の距離を求めることもできます。これを応用すると、モデルとカメラの座標から、距離関係を調べることができます。

外積

３次元での外積(outer product)とは、２つのベクトルに垂直なベクトルのことをいいます。外積を表す記号が"×"のためクロス積とも呼ばれます。外積は以下のように定義されています。

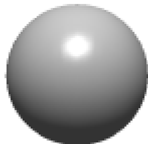
ベクトルa，bの外積＝ a×b ＝ |a||b|sinθ



外積を用いると、面の向きを調べることが出来ます。ほとんどの場合、面は三角形で構成されるので、３つの頂点を持つ平面といえます。頂点の１つを始点とすると、ほかの２つの頂点までの軌跡はベクトルになります。外積は「２つのベクトルに垂直なベクトル」です。つまり、２つの軌跡の外積を計算すれば、面の法線を得ることができます。計算順によって符号(向き)が正反対になってしまいますが、面がどの方向を向いているかを判定することができます。これは、凹凸のある地面の向きを調べるときに利用できます。また、平行なベクトルの外積は０になるという性質を利用し、２つのベクトルが平行かどうかを調べることもできます。

バウンディングスフィア(境界球)

２次元の境界円(バウンディングサークル)を３次元の球(スフィア)に拡張したものです。バウンディングスフィアを表現するためには、最低限中心座標と半径の長さだけでよく、中心間の距離を求めれば衝突しているかどうか判定できるので、計算効率がもっともよい方法です。中心間の距離を求める計算式もピタゴラスの定理そのものなので難しいものにはなりません。モデルが回転しても境界球を回転させる必要がないのも利点ですが、モデルの形状が球に近いものでなければフィット率が低くなり、無駄なスペースが発生したり、球からかなりの範囲のモデルが飛び出してしまいます。



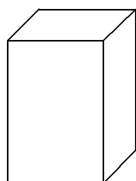
▲架空の球を衝突領域として
設定、衝突を判定する

▲長い形状は余分な領域
が増える

バウンディングボックス

２次元のバウンディングボックスを３次元の直方体に拡張したものです。ボックスの面が各軸に対し垂直のものをＡＡＢＢ(Axis Aligned Bounding Box)と呼び、ボックスを回転できるものをＯＢＢ(Oriented Bounding Box)と呼びます。

ＡＡＢＢは、ボックスを表現するための変数が中心座標と各辺の長さと少なくてすみ、衝突判定がif文だけでできるため、処理負担が少ないのが利点です。しかし、ボックスは回転できないため、モデルの形によってはフィット率が低くなってしまい、形状的な部分で効率が悪くなることがあります。



▲架空のボックスを衝突領域
として設定、衝突を判定する

▲軸に平行、垂直でない形状
になると余分な領域が増える

　ＯＢＢでは、モデルの形や向きにそって回転できるため、形状的にはもっとも効率がよいと言われています。ただし、衝突判定を行うには、分離軸の概念と内積、外積を総動員する必要があります。

レイ

　任意の地点から任意の方向へのばした直線のことをレイ(光線)と呼びます。レイは指定した方向にモデル(ポリゴン)があるかどうか、モデルまでの距離といったものを調べるときに活用します。
　たとえば、プレイヤーの進行方向にレイを飛ばし、地形モデルと当たるかを調べれば、進行方向に壁などの障害物があるかどうかを調べることができます。また、モデルの足下方向へレイを飛ばせば、地面までの距離や地面に当たっているか(めり込んでいるか)を調べることができます。

ESライブラリでの対応

　ESライブラリでは、衝突判定に役立つ以下の関数と構造体があります。

Math_Sqrt関数	平方根を求める
Vector3_Distance関数	二点間の距離を求める
Vector3_DistanceSquared関数	平方根にせず２乗のままの距離を返す
Vector3_Length関数	ベクトルの長さを求める
Vector3_LengthSquared関数	平方根にせず２乗のままの長さを返す
Vector3_Normalize関数	ベクトルを正規化する(長さを１にする)
Vector3_Dot関数	２つのベクトルの内積(ドット積)を求める
Vector3_Cross関数	２つのベクトルの外積(クロス積)を求める
BoundingSphere構造体	境界球。境界球との衝突判定のメンバ関数あり
OrientedBoundingBox構造体	ＯＢＢ。ＯＢＢ、境界球との衝突判定のメンバ関数あり
Ray構造体	レイ。ＯＢＢ、境界球との衝突判定のメンバ関数あり
ViewFrustum構造体	視錐台構造体。境界球、点との衝突判定のメンバ関数あり
MODELクラス	境界球とＯＢＢの生成が可能。 レイとモデルの衝突判定用のメンバ関数あり
ANIMATIONMODELクラス	境界球とＯＢＢの生成が可能
VERTEXBUFFERクラス	境界球とＯＢＢの生成が可能

課　　題

地面のモデルを読み込み、地形の凹凸にあわせてモデルが動くようにしましょう。

(１)凹凸のある地面用のモデルを読み込みます。
　１．ヘッダーファイルでMODEL型変数の宣言
　２．LoadContent関数内でモデルを読み込む

(２)移動用のモデルを読み込ます。

(３)移動用のモデルから足下方向へレイを飛ばし、地面用のモデルまでの衝突距離を測定します。
　１．MODEL型の変数には、IntersectRayという関数があります
　２．IntersectRay関数は、指定座標から指定方向へ直線を引き、それがモデルと衝突するかどうか、衝突する場合には、その距離、衝突面の傾き(法線)を取得することができます
　３．
```
// 地形判定
float   dist;   // 地面モデルまでの距離
ground->IntersectRay(anime->GetPosition(), Vector3_Down, &dist, NULL);
```
　４．移動用モデルの原点座標によっては、正常な値が取得できないので、GetPosition関数にVector3型を足して補正しなければならない場合があります
　５．IntersectRay関数から返された距離が基準値より増減していれば、地面の凹凸部分になるので、移動用モデルのｙ座標を補正することにより、地面の高さに合わせることができます