オブジェクト指向とゲームプログラミング

§　アルゴリズム編 － 第２回　衝突検出　§

衝突検出

ゲームの中で、衝突検出(Collision Detection:衝突判定、当たり判定、ヒットチェックなどとも呼ばれます)は、コンピュータゲームが出始めたころからすでに必要とされていました。たとえば、シューティングゲームです。プレイヤーが発射した弾が敵に衝突したかどうかを検出しなければ、ゲームは成り立ちません。また、２次元のＲＰＧでは、キャラクタが街を歩き回るときに壁などの障害物に衝突すると、そこで足踏みしていました。これも、そこに障害物があることを検出できたからこその動きなわけです。

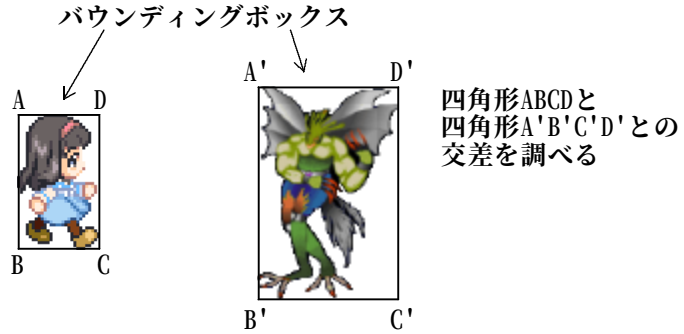
ゲームでは、キャラクタ同士の衝突を検出するという処理は、ひんぱんに必要になります。一般的なシューティングゲームを見ても、自機と敵、弾(自機)と敵、自機と弾(敵)など、さまざまな種類の衝突検出が行われます。さらに、それらが１秒間に６０回の移動が行われるとすると、そのぶんだけ衝突検出が必要となります。

このような衝突検出は、２次元でも３次元でも基本は同じです。３次元の場合は、２次元の衝突検出を拡張した処理が行われています。

バウンディングボックス

ゲームのようなリアルタイム性を追求するアプリケーションの場合、キャラクタの複雑な形状を完全に考慮した衝突検出をするのはたいへんです。実際には、バウンディングボックス(Bounding Box:衝突範囲)と呼ばれる架空の領域を用いて、その領域の交差判定を行います。

２次元の場合には、矩形や円を用いることが多く、この領域ともう一方のキャラクタの交差判定を行うことで、擬似的に衝突を検出します。たいていの場合、このバウンディングボックスは、そのキャラクタをすっぽりと包むようにします。そうでない場合には、キャラクタのめり込みが起こる可能性が出てきます。



矩形のバウンディングボックスの場合、多くはその長方形の軸を画面座標系のｘｙ軸に平行にとります。この形式のものをAABB(Axis-Aligned Bounding Box)と呼びます。この場合には、バウンディングボックスの領域は、左上の頂点座標と右下の頂点座標(または右上と左下の頂点座標)で与えることができます。ゲームでは、キャラクタのだいたいの位置関係は把握できていることが多いので、AABBを使うと条件判定回数が非常に少なくて済むという利点があります。しかし、キャラクタが回転などをする場合には、それに合わせてボックスの頂点座標を計算し直すことになるので、その手間がかかることがあります。

矩形の衝突検出

上の図のバウンディングボックスの頂点Aの座標を(Ax, Ay)、頂点Cを(Cx, Cy)、頂点A'を(A'x, A'y)、頂点C'を(C'x, C'y)と表します。このとき、２つのバウンディングボックスが交差しているかどうかを調べるプログラムは、次のようになります。

```
if(Ax <= C'x && Cx >= A'x && Ay <= C'y && Cy >= A'y)
    交差している
else
    交差していない
```

画面座標系のｙ軸は、数学と違って下方向に向いていることに注意しましょう。これをC++で記述すると、

```
// 衝突判定([ax1, ay1]が頂点A、[ax2, ay2]が頂点C、[bx1, by1]が頂点A'、[bx2, by2]が頂点C')
bool IsIntersect(int ax1, int ay1, int ax2, int ay2, int bx1, int by1, int bx2, int by2) {
    if((ax2 < bx1 || ax1 > bx2 || ay1 > by2 || ay2 < by1) == true)
        return false;  // 衝突していない
    return true;       // 衝突している
}
```

となります。また、WindowsのAPIには、２つの矩形が交差しているかを調べるIntersectRect関数と矩形内に座標があるかどうかを調べるPtInRect関数があります。これらの関数のほかに、リージョンという機能を使っても、衝突を検出することができます。リージョンでは、多角形や円などを始め、複雑な形状の交差を調べることができます。

円の衝突検出

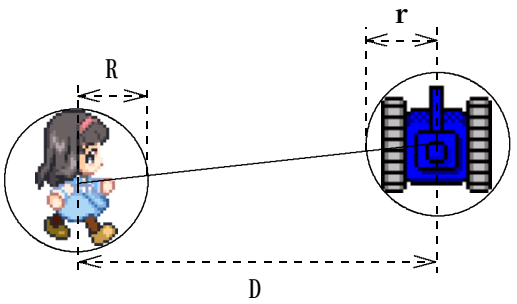
バウンディングボックスを円にすると、その中心と半径を定義するだけで扱うことができます。キャラクタの衝突を検出する場合には、それぞれの衝突範囲を定義している円の中心同士の距離を測定し、これが各半径の和以下であれば、このふたつのキャラクタは衝突しているとみなします。

距離は、

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

と計算しますが、ここでは衝突しているかどうかを調べるだけなので、よぶんな平方根計算は省いて、D^2と$(R + r)^2$の大小比較で判定を行うことができます。

$D^2 \leqq (R + r)^2$ … 接触
$D^2 > (R + r)^2$ … 非接触



$D \leqq R + r$ … 接触
$D > R + r$ … 非接触

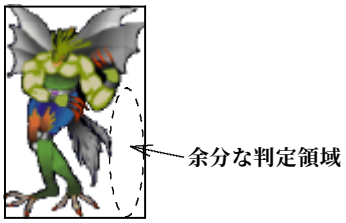
▲円の衝突検出

バウンディングボックスの細分化

AABBはよく使われますが、キャラクタの形状によって誤差が大きく変動します。たとえば、下の右側のようなキャラクタでは、実際には衝突していない部分を多く含んでしまう可能性があります。左側のように、直立しているキャラクタであれば、そのバウンディングボックスは非常に適合していますが、真ん中のように４５度傾いた状態では、これだけの余分な衝突領域が生じてしまいます。

これはいいが…

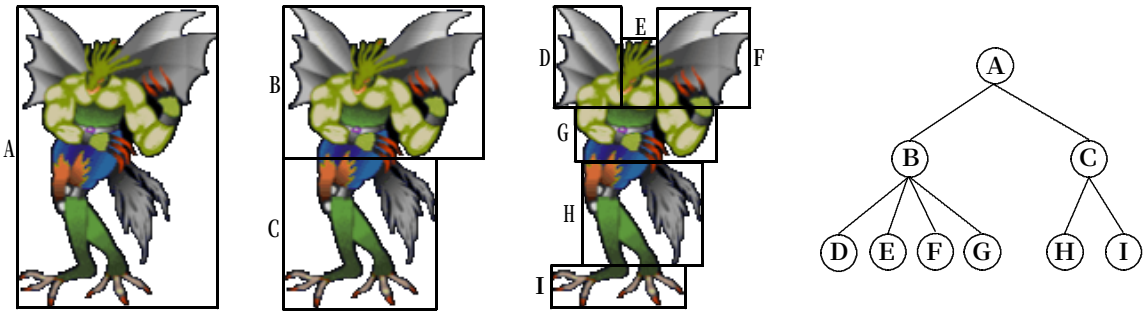
傾くと…



これはムダが多い

これを極力減らすには、AABBによる細分化を考えます。これは、キャラクタをいくつかのパーツと考え、それにバウンディングボックスを被せる方法です。こうしてキャラクタの衝突を、複数の衝突判定で検出するのです。計算量はパーツの分割数にそのまま比例しますが、衝突検出の誤差はかなり改善されます。

また、逆に複雑な形状のキャラクタ同士の衝突判定に、このいくつかの異なったレベルのバウンディングボックスを使うことで、早い段階での判定が可能になります。



▲細分化されたAABBツリー