ESライブラリ＆＆
ゲームプログラミング

○　ゲーム制作編 - 第５回　衝突検出　○

衝突検出

・Meteorは3Dを使っているが、実際の処理はｙ座標固定の2D(x-z平面)で行われている
・よって、キャラクター同士の衝突検出には、2Dのアルゴリズムでも十分な判定ができる

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とも呼ばれます)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。2D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。

3Dでも衝突検出の基本は同じで、2Dの衝突検出を拡張した処理が行われています。

バウンディングサークル

衝突範囲を円で設定する方法です。円の衝突範囲は、その中心と半径を定義するだけで扱うことができます。それぞれの円の半径を足した値が基準です。円の中心間の距離が半径の和を超えていれば、円同士ははなれており、衝突していないことになります。逆に、円の中心間の距離が半径の和未満であれば、円同士が衝突していることになります。中心間の距離と半径の和が一致した場合は、厳密には接触していることになりますが、ゲームバランスを考慮してプレイヤー有利とする場合は、衝突していないことにします。

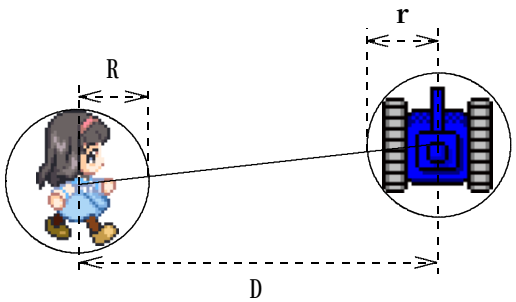
キャラクターの衝突を検出する場合には、それぞれの衝突範囲を定義している円の中心同士の距離を測定し、これが各半径の和以下(もしくは未満)であれば、このふたつのキャラクターは衝突しているとみなします。

距離は、三平方(ピタゴラス)の定理をもとに

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

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。

$D^2 \leqq (R + r)^2$ … 衝突(※'＝'の扱いはどちらでもよい)
$D^2 > (R + r)^2$ … 非衝突

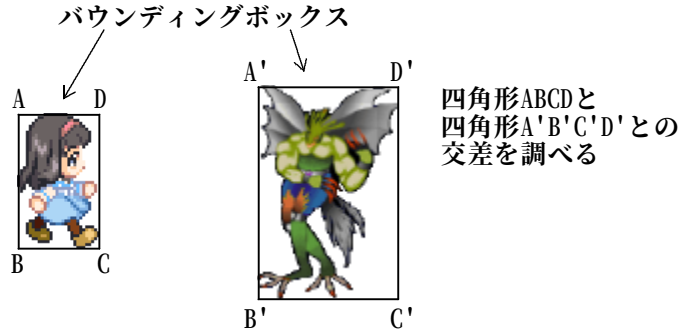


D ＝ R ＋ r … 接触
D ＜ R ＋ r … 交差
D ＞ R ＋ r … 非接触
なので
D ≦ R ＋ r … 衝突
D ＞ R ＋ r … 非衝突

▲円の衝突検出

バウンディングボックス

衝突範囲を矩形(長方形)で設定する方法です。たとえば、以下のようにキャラクターをすっぽり包むようにバウンディングボックスを設定するものです。



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、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、キャラクター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、キャラクターが回転をする場合には、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。

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
```

2Dの場合、画面座標系のｙ軸は下方向に向いていることに注意しましょう。これをＣ＃で記述すると以下のようになります。

```
// 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
}
```

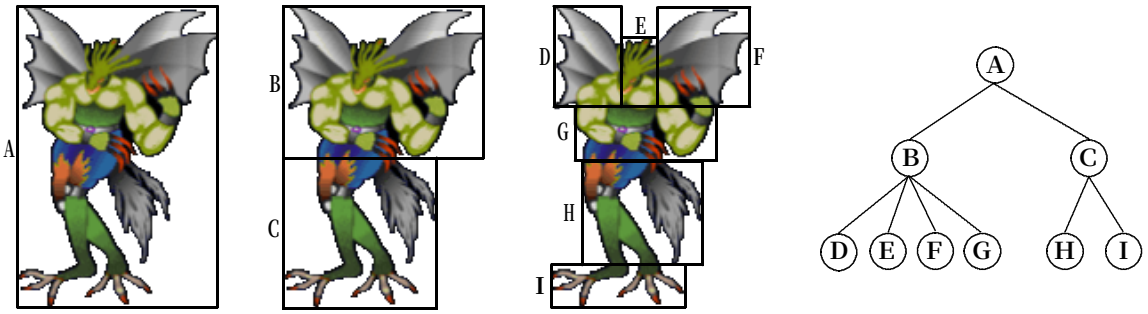
バウンディングボックスの細分化

AABBはよく使われますが、キャラクター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ーでは、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ーであれば、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、キャラクター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ーの衝突を、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ー同士の衝突判定に、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

課　　題

プレイヤーと隕石の衝突を検出し、衝突している場合は画面中央に"DAMAGE"と表示してみましょう。

(１)内部処理で衝突判定、描画処理でその結果を受けて"DAMAGE"と表示するかどうかを判断します。そのための衝突しているかどうかを示す変数が必要です。
以下のプログラムを適切な場所に追加しましょう。

```
bool  hitFlg;        // 衝突フラグ
```

※さらに、適切な場所で'false'に初期化してください

(２)"DAMAGE"と表示する以下のプログラムを適切な場所に追加しましょう。

```
// ダメージ描画
if(hitFlg)
    SpriteBatch.DrawString(DefaultFont, TEXT("DAMAGE"), Vector2(600.0f, 380.0f),
                           Color(1.0f, 0.0f, 0.0f));
```

※座標、色は任意に変更してください

(３)衝突しているかどうかを判定する以下のプログラムを適切な場所に追加しましょう。

```
// 衝突検出
hitFlg = false;
const float   plyR = 0.6f, meteoR = 0.4f;           // 半径
const float   Rr = playerR + meteoR;                // 接触しているとみなす最大の距離
for(int i = 0; i < 100; i++) {
    // ２点間の距離
    const float   D = Vector3_Distance(plyPos, meteoPos[i]);
    if(D < Rr)
        hitFlg = true;
}
```

※変数名、半径は適切に訂正してください。また、今回は衝突していたらbreakしてもかまいません
※※if(Vector3_Distance(plyPos, meteoPos[i]) < Rr) と１行にまとめることもできます
※※※Vector3_Distance関数より、Rrを２乗するアルゴリズムの方が高速です
(Vector3_DistanceSquared関数という、距離が２乗された値を返す関数があります)
※hitFlgをint型にして数値を増減させることにより、"DAMEGE"描画時間を調整できます