衝突判定

◆衝突判定(直線)

ゲームプログラミングでは、2 つの直線が交差するかどうか、交差するならどこで交差するかを調べなければならないことがあります。ゲームではこれらの直線は建物の側辺であったり、地面であったり、物体の経路であったりします。こういった直線の 2 つが交差する場合、その交点に基づいて条件式を書かなければなりません。これらの直線の方程式の求め方はわかりますか？

そうですね。交点を求めるには 2 つの直線の方程式を連立させて連立方程式を組み、それを解きます。解(交点)は、2 つの方程式を満たすすべての点の集まりです。

直線と交点について、例を通してまとめてみましょう。

【例】

(1) $\begin{cases} 2x+3y=3 & ① \\ -x+3y=-6 & ② \end{cases}$

この 2 式を変形すると、①は、$y=-\frac{2}{3}x+1$　②は、$y=\frac{1}{3}x-2$ となり、**傾きは異なります**。

①+②×2　$2x+3y=3$

$\underline{+(-2x)+6y=6}$

$9y=9$　$y=1$　①に代入　$2x+3\times1=3$　$2x=0$　$x=0$

この 2 式は、1 点 $(0,1)$ で交差します。つまり、2 つの直線の交点は、$(0,1)$ です。

(2) $\begin{cases} -3x+6y=6 & ① \\ -x+2y=2 & ② \end{cases}$

この 2 式を変形すると、①は、$y=\frac{1}{2}x+1$　②は、$y=\frac{1}{2}x+1$ となり、**傾き・y 切片ともに、一致します**。つまり、2 つの直線は、ぴったり重なります。この場合の解(交点)は、1 つの点だけでなく、いずれかの直線上にあるすべての点からなる無限集合です。

(3) $\begin{cases} -x+2y=2 & ① \\ -x+2y=-2 & ② \end{cases}$

①は、$y=\frac{1}{2}x+1$　②は、$y=\frac{1}{2}x-1$ と変形でき、2 つの直線の**傾きは同じで、y切片は異なります**。従って、2 つの直線は「平行」となり、共通点を持ちません。解集合は空集合となります。

このように、連立方程式の解の個数は、方程式のグラフの傾きと y 切片に関係があります。

連立 2 元 1 次方程式の解

同一平面における連立 2 元 1 次方程式の解の個数は、

①2 つの方程式のグラフの**傾きが異なるとき、1 つ**

②2 つの方程式のグラフが**傾きも y 切片も同じとき、無限個**

③2 つの方程式のグラフの**傾きが同じで y 切片が異なるとき、ゼロ**

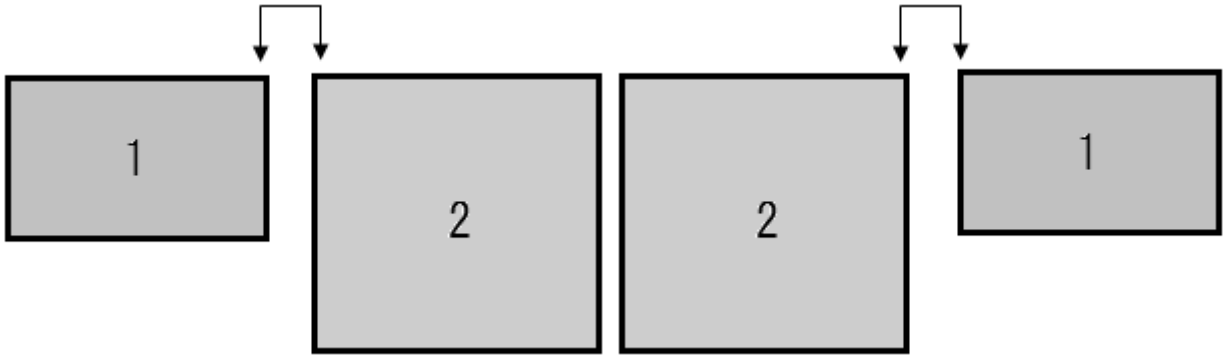
「　衝突判定　」

東京情報クリエイター工学院

◆衝突判定(矩形)

当たり判定のうち 2D では最も基本となる**矩形**(くけい)(長方形)同士の当たり判定について考えてみましょう。

矩形同士が当たっているかどうかを考えるとき、「どうなっていれば当たっている可能性が**<u>あるか</u>**」を考えるよりも、まずは「どうなっていれば当たっている可能性が**<u>ないか</u>**」を考える方がわかりやすいでしょう。



上図のように、矩形 1 と矩形 2 を考えてみます。まず、矩形 2 を基準に考えた場合、もし、矩形 1 の右端が矩形 2 の左端よりも左にあれば、2 つの矩形は当たっていません。また、矩形 1 の左端が矩形 2 の右端よりも右にあれば、2 つの矩形は当たっていません。

これをまとめれば、「**矩形 1 の右端が矩形 2 の左端よりも左にあるか、または、矩形 1 の左端が矩形 2 の右端よりも右にあれば、矩形 1 と矩形 2 が当たっている可能性はない**」ということです。

ということは、この逆があたっている可能性がある場合です。この逆とは？

「矩形 1 の右端が矩形 2 の「　　」端よりも「　　」にあり、「　　」、
矩形 1 の左端が矩形 2 の「　　」端よりも「　　」にあれば、矩形 1 と矩形 2 が当たっている可能性がある」

この論理は数学的には、「**ド・モルガンの法則**」と呼ばれているものです。

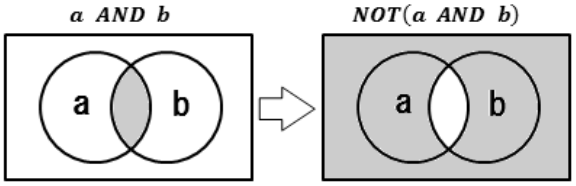
ド･モルガンの法則

$$NOT(a\ AND\ b) = (NOT\ a)\ OR\ (NOT\ b)$$
$$NOT(a\ OR\ b) = (NOT\ a)\ AND\ (NOT\ b)$$

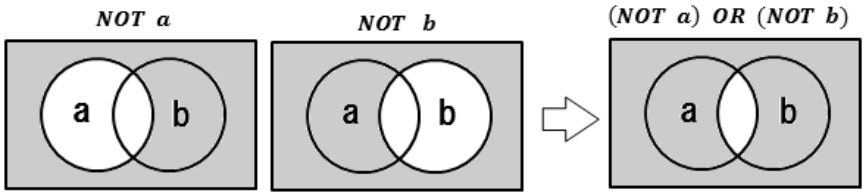
【証明】

(1) $NOT(a\ AND\ b) = (NOT\ a)\ OR\ (NOT\ b)$

左辺

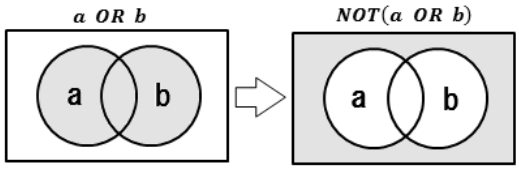


右辺

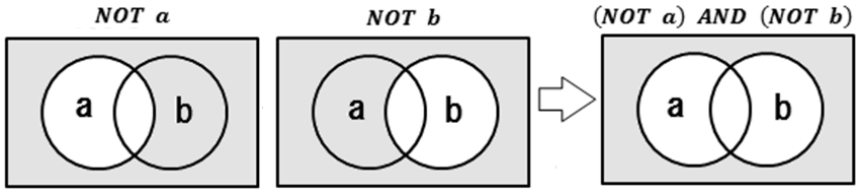


(2) $NOT(a\ OR\ b) = (NOT\ a)\ AND\ (NOT\ b)$

左辺

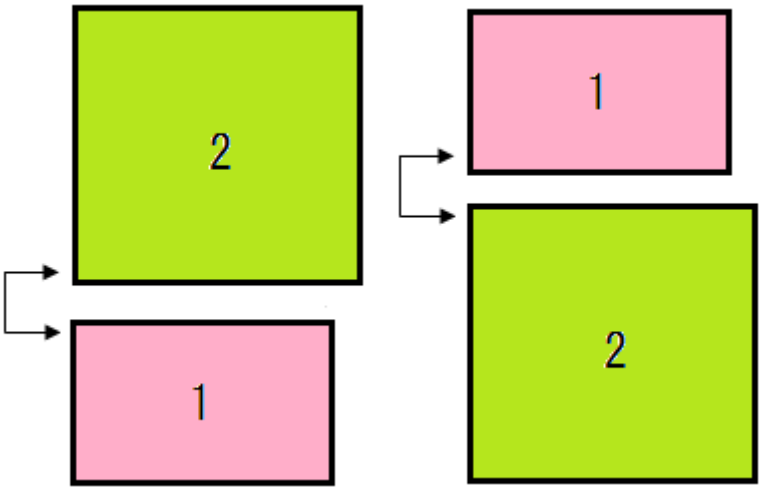


右辺



「ド・モルガンの法則」は、知らなくても考えれば用は足りる場合が多いのですが、知っていれば考える効率がよくなり、時間の節約になりますから、覚えておくと便利ですよ。

次に、横方向だけを見て当たっている可能性があった場合、さらに縦方向についても同じようにチェックします。つまり、「**矩形 1 の下端が矩形 2 の「　　」端よりも「　　」にあり、「　　」、矩形 1 の上端が矩形 2 の「　　」端よりも「　　」にある**」場合に、縦方向だけ考慮すると矩形 1 と矩形 2 が当たっていることになります。

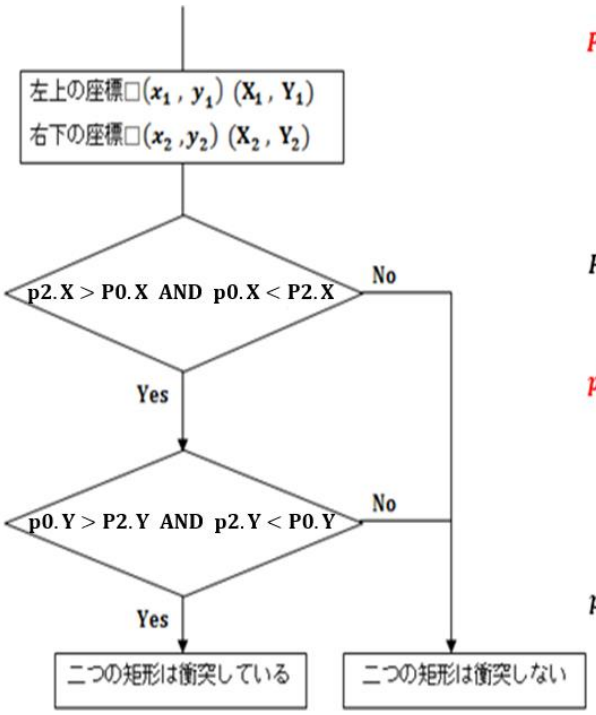


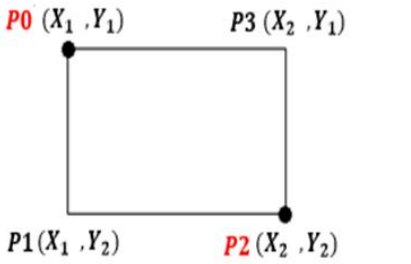
さて、今までの話では、2 つの矩形のちょうど境界同士が当たっていた(触れていた)場合について考慮されていません。つまり、「**矩形 1 の右端が矩形 2 の左端とちょうど同じ位置**」という場合です。その場合はどうするか考慮する必要があります。

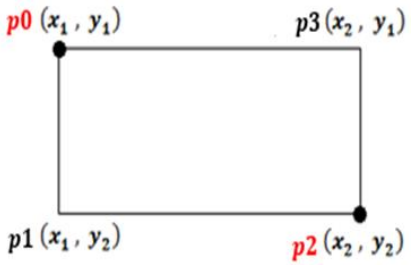
「　衝突判定　」

東京情報クリエイター工学院

この関係をゲーム作成のときに使用するには、以下のようなアルゴリズムになります。







◆矩形の衝突判定　プログラム

１．準備

(0)テンプレートの作成

①[C_番号_氏名]フォルダー[MathCalc]フォルダー[MathCalc]ソリューションを起動。

②[プロジェクト]－[テンプレートのエクスポート]

（ｉ)[プロジェクトテンプレート]・作成元のプロジェクト[MathTemplate]を選択し、次へクリック

（ii)テンプレート名[Math02Ex0]・テンプレートの説明[ゲーム数学２用テンプレート]・テンプレートを自動的にVisual Studioにチェックを入れ、完了クリック

(1)プロジェクトの追加

①「ソリューションエクスプローラ」の「ソリューション」にマウスポインタを合わせて「右クリック」－「追加」－「新しいプロジェクト」

②「新しいプロジェクトの追加」で、「Visual C#」－「Math02Ex0」をクリックする。

（名前が「Math02Ex01」となっているはず）

(2)「Math02Ex01」をスタートアッププロジェクトに設定

２．プログラムの作成

(1) [MyDrawClass.cs]に衝突判定の処理を追加しましょう。

```
namespace Math02Ex01
{
    public class MyDrawClass : Draws, IDraws
    {
        //フィールド
        InputState input;
        Vector2 p0, p1, p2, p3;   //修正
        Vector2 P0, P2;           //追加
```

```
            //コンストラクタ
            public MyDrawClass()
            {
                input = new InputState();
                input.MouseOn();
            }

            //図形データの入力処置
            public void InputData()
            {
                  :  //後で修正する！
             }

            public void Update(){}

            public void Draw(){}

            //追加　衝突判定
            private static bool Collision_Square
                (Vector2 p0, Vector2 p2, Vector2 P0, Vector2 P2)
            {

                衝突判定処理を追加します！

            }
        }
    }
```

(2)衝突判定の次に、矩形の描画処理を加えましょう。

```
            //追加　衝突判定
            private static bool Collision_Square
                (Vector2 p0, Vector2 p2, Vector2 P0, Vector2 P2)
                    {
                        :
                    }

            //追加　矩形の描画
            private void Square(Vector2 p0,Vector2 p2)
            {
                p1 = new Vector2(p0.X, p2.Y);
                p3 = new Vector2(p2.X, p0.Y);
                Line(p0, p1);
                Line(p1, p2);
                Line(p2, p3);
                Line(p3, p0);
             }
```

「　衝突判定　」

東京情報クリエイター工学院

(3)入力処理を修正しましょう。

```
            //図形データの入力処置
            public void InputData()
            {
                Init();//属性の初期化（色・太さ・文字高さを既定値にする）
                Clear();//描画領域のクリア（座標軸のみ描画されます）

                //マウス入力
                List<PointF> p = new List<PointF>();
                p = input.GetPoint(2, "矩形１の対角線の頂点を２点入力");   //修正

                //入力キャンセル処理
                if (p.Count == 0)  {  Clear(); return;  }

                p0 = new Vector2(p[0].X, p[0].Y);

                //以下　追加
                P0 = p0;

                p2 = new Vector2(p[1].X, p[1].Y);
                P2 = p2;

                Square(P0 , P2);
                Render();  //矩形１の描画

                p = input.GetPoint(2, "矩形２の対角線の頂点を２点入力");

                //入力キャンセル処理
                if (p.Count == 0)  {  Clear();  return;  }

                p0 = new Vector2(p[0].X, p[0].Y);
                p2 = new Vector2(p[1].X, p[1].Y);

                Square(p0 , p2);
                Render();  //矩形２の描画

                //衝突判定の結果表示
                if(Collision_Square(p0, p2, P0, P2))
                {
                    Text(new Vector2(-200, 200), "衝突しました");
                    SetColor(Color.Red);

                }
                else
                {
                    Text(new Vector2(-200, 200), "衝突してません");
                    SetColor(Color.RoyalBlue);
                }
                Square(p0, p2);
                Square(P0, P2);

                Render();
            }
        }
    }
```

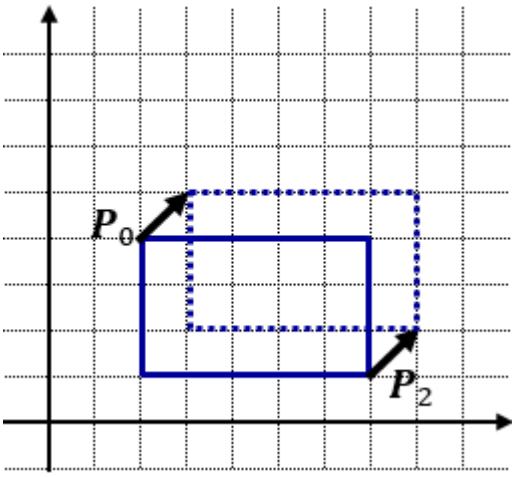
「　衝突判定　」

東京情報クリエイター工学院

[衝突判定処理　プログラム例]

```
        //追加　衝突判定
        private static bool Collision_Square
            (Vector2 p0, Vector2 p2, Vector2 P0, Vector2 P2)
        {
            if ((p2.X > P0.X) && (p0.X < P2.X))
            {
                if ((p0.Y > P2.Y) && (p2.Y < P0.Y))
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
            else
            {
                return false;
            }
        }
    }
```

◆アニメーションをつけてみよう！



矩形を動かしてみましょう。

矩形を動かすには、図のように P_0P_2 を同じ向きで同じ量だけ(速度)同時に動かし、矩形を描きます。そしてこれを繰り返すことで、矩形は動きます。

速度は、プログラムでは

$Vector2\ mov;$

$mov = new\ Vector2(1,1);$

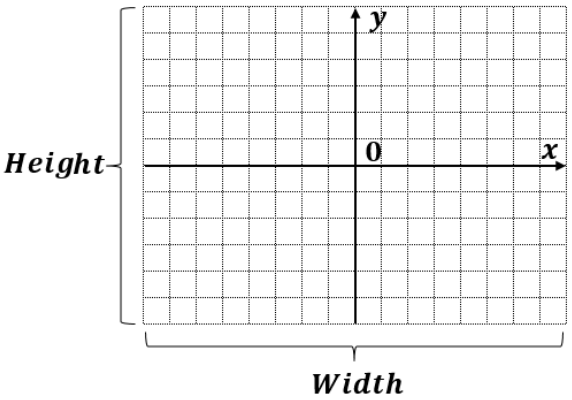
と記述します。

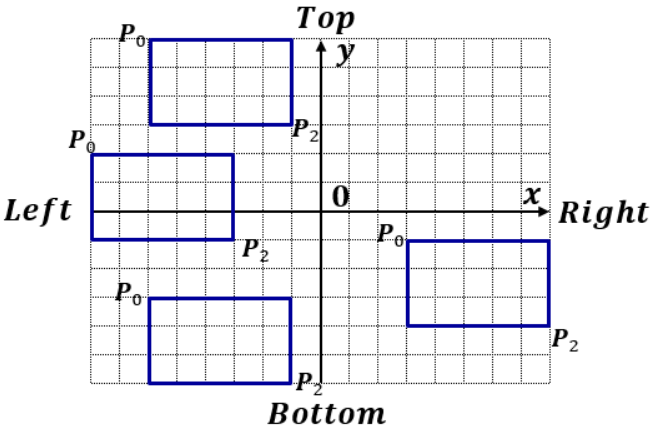
これを、P_0P_2 に加えていきます。

$P_0 = P_0 + mov\ ;\ P_2 = P_2 + mov;$

ここで、矩形を動かしていくと、表示ウィンドウから矩形がはずれます。そこで、表示ウィンドウに衝突したら、速度の方向を逆にします。

表示ウィンドウと座標の関係は左図のようになっています。ここに、矩形を配置(右図)して、どのような状態になったときに、速度の方向を逆にするか考えてみましょう。





「　衝突判定　」

東京情報クリエイター工学院

$Left =$　　$Right =$

$Top =$　　$Bottom =$

速度の x 方向を逆にするのは、

速度の y 方向を逆にするのは、

１．上記の処理をプログラム(MyDrawClass.cs)に追加しましょう。

```
//フィールドに追加
Vector2 mov1, mov2;     //速度
float Left, Right, Top, Bottom;  //ウィンドウ処理に使用
bool Flag;   //衝突判定に使用
//コンストラクタに追加
mov1 = new Vector2(1, 2);       //矩形１の速度の初期値
mov2 = new Vector2(-2, -2);     //矩形２の速度の初期値
Flag = false;                   //フラグの初期値(false：衝突していない)

//表示ウィンドウの衝突処理
private void AreaMove(ref Vector2 p0, ref Vector2 p2, ref Vector2 mov)
{
    衝突判定処理を追加しましょう！
}
```

２．追加したら、以下を加えます。

```
 public void Update()
{
    AreaMove(ref p0, ref p2, ref mov1);
    AreaMove(ref P0, ref P2, ref mov2);

    if (Collision_Square(p0, p2, P0, P2))
    {
        Flag = true;
    }
    else
    {
        Flag = false;
    }
}

public void Draw()
{
    if (Flag)
    {
        Text(new Vector2(-200, 200), "交差しました");
        SetColor(Color.Red);
    }
    else
    {
        Text(new Vector2(-200, 200), "交差してません");
        SetColor(Color.RoyalBlue);
    }

    Square(p0, p2);
    Square(P0, P2);
}
```

「　衝突判定　」

東京情報クリエイター工学院

[表示ウィンドウの衝突判定処理　プログラム例]

```
//表示ウィンドウの衝突処理
private void AreaMove(ref Vector2 p0, ref Vector2 p2, ref Vector2 mov)
{
    Left = -Width / 2;
    Right = Width / 2;
    Top = Height / 2;
    Bottom = -Height / 2;

    if ((p0.X < Left) || (Right < p2.X)) mov.X *= -1;
    if ((p2.Y < Bottom) || (Top < p0.Y)) mov.Y *= -1;
    p0 += mov;
    p2 += mov;
}
```

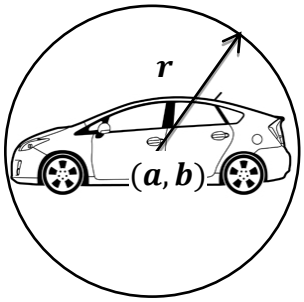
「　衝突判定　」

東京情報クリエイター工学院

◆衝突判定(円と球)

１．円を利用した衝突判定

ゲームでの境界の形には、円・球がよく使われます。もちろん、他の形を使うこともできますが、円や球は数値上の扱いが簡単なうえ、円どうしあるいは球どうしの衝突テストが他の形を使った場合よりも格段に【　**速い**　】というメリットがあります。しかし、デメリットもありそれは【　**正確さ**　】には欠けることがあるという点です。

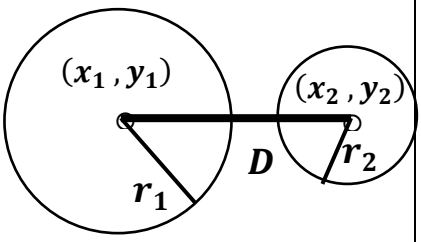
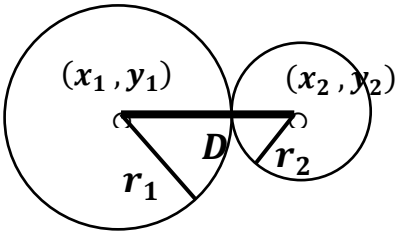
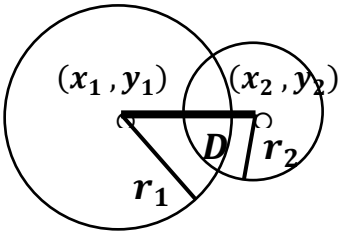


右図は，車に境界円を導入した例で，車の上の部分と下の部分が空いていて正確に接していないことが分かります。しかし、計算の速さや客観的な分かり易さは、このデメリットをしのぎ、第１段階の衝突チェックとしては十分です。

中心座標：$(a,\ b)$　　**半　径**：r

円と球の方程式を使って 2 つの物体が衝突したかどうかを数学的に判断する方法を見ていきましょう。まずは、2 次元の円について説明した後、同じプロセスを 3 次元に拡張します。

2 台の車をそれぞれの境界円の中において衝突検知を考えてみましょう。
まず、2 つの境界円の関係を図で描きます。その際、境界円には半径 r_1　r_2 とし、2 つの円の中心「$(x_1, y_1)(x_2, y_2)$」をつなぐ線を D　(Distance：距離)とします。

離れている 2 つの境界円	**接している 2 つの境界円**	**重なりあっている 2 つの境界円**
		
D と半径 $r1$　$r2$ の関係		

このように、衝突しているかの判定は、2 つの円の中心間の距離 D と 2 つの円の半径の和 $(r_1 + r_2)$ によって容易に導くことができます。
簡単にまとめると、2 つの円の中心間の距離が 2 つの円の半径の和以下の状態の時 2 つの円は衝突していることになります。
したがって、衝突判定では条件式が【　**小さいか等しい**　】かをチェックすればよいことになります。
あとは、以前に学習した距離の公式を使えば、2 つの円の中心間の距離を計算することができます。

「　衝突判定　」

東京情報クリエイター工学院

2 つの円の衝突検知

2 つの円　$(x-a_1)^2+(y-b_1)^2={r_1}^2$ および $(x-a_2)^2+(y-b_2)^2={r_2}^2$ が与えられたとき、

であれば、2 つの円は衝突している。

★覚えておこう

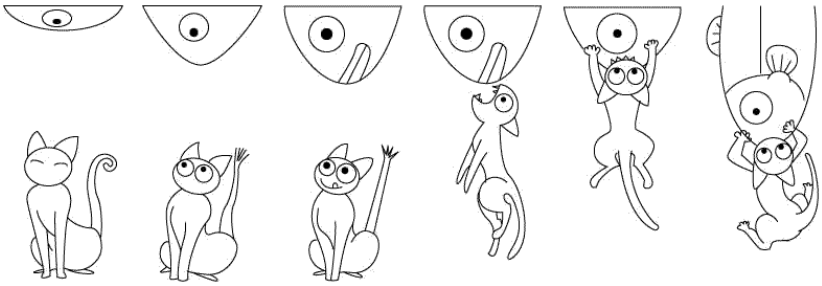
(1) フレーム単位で動く

ゲームでは、物体はフレーム単位で動く【 **60fps＝1 秒間に 60 フレーム**　】

(2) 衝突検知はフレームごとに処理される

衝突検知のプログラムは、【 **1 フレーム** 】ごとに呼び、チェックできるようにしなければならない。

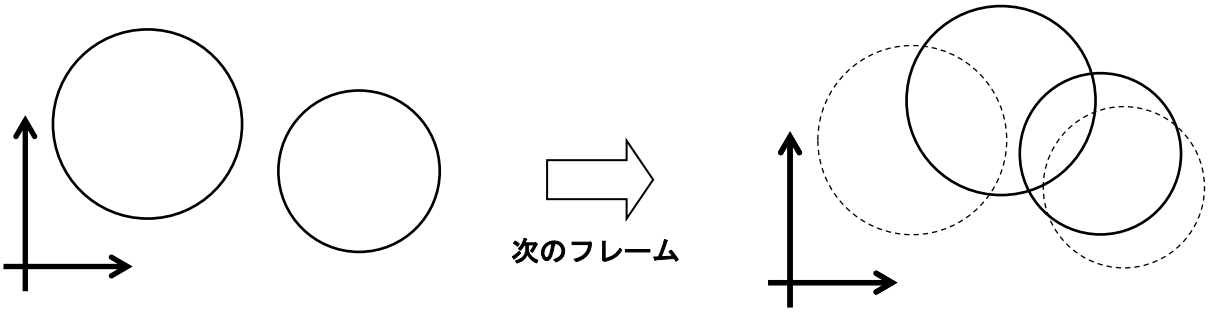
(3) キャラクタはスムーズに動かない



ゲームキャラクタの動きは、マンガのアニメーションと同じ概念で、上記の間隔（1/60 秒）で表示されています。(映像は 1/30, または 1/24 秒）また、キャラクタはフレーム単位で動くため、計算を考える場合に動きはスムーズではないことにも注意が必要となります。

(4) ゲームは錯覚を利用

人間の目では連続して動いているように見えるかもしれませんが、コンピュータの中では、飛ばしています。したがって、2 つの物体があるフレームではまったく接触していなくても、次のフレームでは重なり合っているということもありえます。つまり、接している状態を飛び越えて、いきなり重なった状態になっているわけです。



(5) チェックとしての判定条件

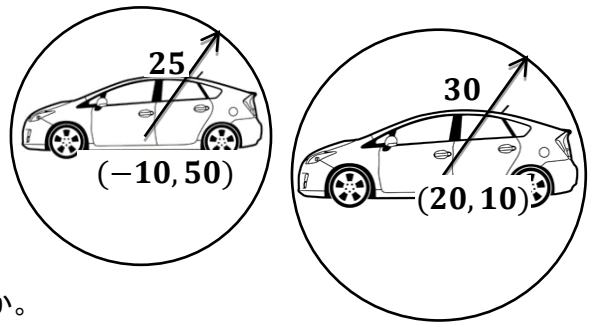
衝突検知は「　**等しいかどうか**　」チェックではなく、「　**小さいか等しい**　」チェックが必要なのはこのためです。

「　衝突判定　」

東京情報クリエイター工学院

【例題１】

2D レーシングゲームの衝突検知プログラムを作成しているとします。あなたは、境界円を使って衝突を検知することにしました。現在のフレームでは、車 $\boldsymbol{A}$ の境界円は方程式 $(\boldsymbol{x-20})^2+(\boldsymbol{y-10})^2=\boldsymbol{900}$ で定義され、車 $\boldsymbol{B}$ の境界円は、$(\boldsymbol{x+10})^2+(\boldsymbol{y-50})^2=\boldsymbol{625}$ で定義されます。2 つの車は衝突しているでしょうか。



【解答】

⑴2 つの境界円の中心と半径を求める。

車 $\boldsymbol{A}$ の境界円の中心は「　　」半径は「　　」

車 $\boldsymbol{B}$ の境界円の中心は、「　　」半径は「　　」

⑵中心間の距離を求める

$\mathbf{D}=$

⑶衝突判定チェック

最後に、$\mathbf{D}\leqq(\boldsymbol{r_1+r_2})$ かどうかをチェックする。

$\boldsymbol{r_1+r_2}=$

従って、「　　**2 つの円は　衝突**　　」

２．2 つの円の最適化衝突判定

距離の公式ではルート(√) を使った方法で表しました。このルートをプログラムで組む場合、プログラム中に数学計算用 Math. クラス にある Sqrt メソッドを利用すると、意外と計算に時間がかかってしまいます（これを計算コストが高いという)。ゲームプログラマは、プログラミングにおいて計算コストがかからないように常に考えてプログラムを書かなければいけません。

ここでは Math.Sqrt () を使わなくてもよいように、衝突判定の条件を改良してみることにします。改良方法としては、下記のように両辺を 2 乗すると良いでしょう。

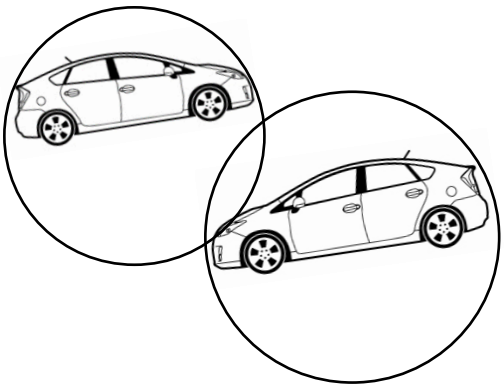
不等式は、両辺を 2 乗してもその関係は変わりません。先ほどの例題で確認してみると、「55」は「50」より大きいですし、「3025」は「2500」より大きいですね。この方法を一般的に記述すると、以下のようになります。

2 つの円の最適化衝突判定

2 つの円　$(\boldsymbol{x-a_1})^2+(\boldsymbol{y-b_1})^2=\boldsymbol{r_1}^2$ および $(\boldsymbol{x-a_2})^2+(\boldsymbol{y-b_2})^2=\boldsymbol{r_2}^2$ が与えられたとき、

であれば、2 つの円は衝突している。

３．偽の衝突



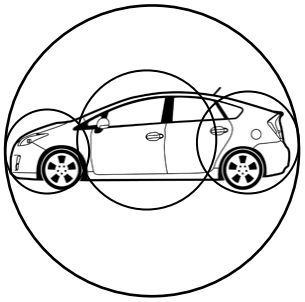
これまで見てきたように、境界円と境界球を使えば衝突を簡単に検知することができます。比較的計算速度が速い方法ですが、これが最も正確な方法かといえば、必ずしもそうとは限りません。
円の中には空白がありますね。この空白がくせものです。この空白によって、偽の衝突を検知する場合があるからです。
右図が偽の衝突を判定した状態です。

４．偽の衝突の回避策

(1)もっとぴったりとフィットする別の形を使う

偽の衝突を防ぐ一つの方法は、モデルにもっとぴったりフィットする別の形を使うことです。この場合、数値的なチェックができる何らかの方程式で定義できる形を使うことが重要です。

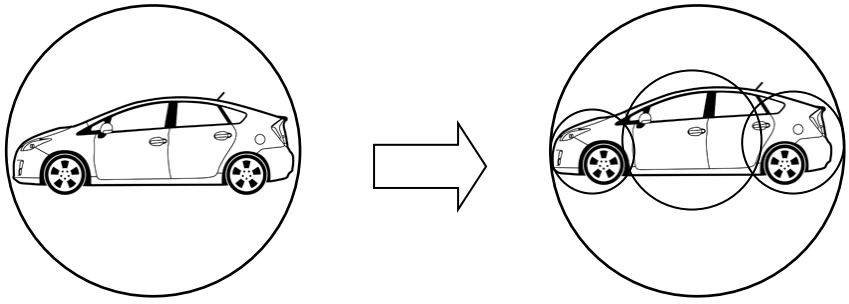
(2)境界円の大きさを段階的に異なる円の階層で設定する方法



もう一つの方法は、大きさが段階的に異なる円の階層を設定するというものです。この方法では、最初に元の 2 つの境界円の衝突をチェックします。衝突がない場合は何の問題もありませんが、衝突がある場合は、それが偽の衝突かもしれないので要注意です。衝突があるという答えが返ってきたら、その時点でもっと細かくチェックします。つまり、衝突がなければそのまま続行し、衝突がある場合は詳細チェックをするという条件文を設定します。
この詳細チェックでは、モデルにもっとぴったりフィットする小さな境界円をいくつか使います。小さい複数の境界円を使うことで検知の精度はあがりますが、衝突が起こる可能性があれば、余分な処理時間がかかります。やみくもに正確に判定することのないよう注意しましょう。

◆車の衝突判定 プログラム

[Collision_Ex0]を実行して、偽のあたり判定を確認しましょう。確認後、偽の当り判定を回避するプログラムを作成しましょう。
先ほどの「**偽の衝突の回避策**」で解説した「**境界円の大きさを段階的に異なる円の階層で設定する方法**」で考えてみましょう。



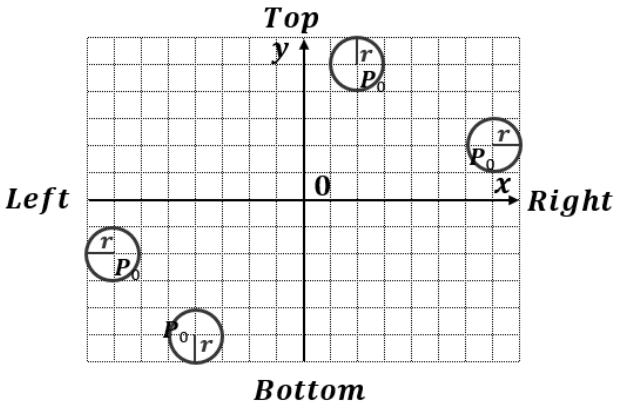
現在の処理は、外側の境界円に当たっているかどうかのみチェックしています。これを、最初に、「**外側の境界円**」をチェックし、「**当たっていたら、内側の境界円をチェック**」するようにします。
[Collision_Ex0]の[MyDrawClass.cs]を修正していきましょう。

「　衝突判定　」

東京情報クリエイター工学院

現段階の外側境界円のみでの当たり判定

```
//壁との衝突判定　（衝突したら移動方向を反転させる。）
      private void AreaMove(ref Vector2 position, ref Vector2 mov, float r)
      {
          if ((position.X - r < Left) || (Right < position.X + r)) mov.X *= -1;
          if ((position.Y - r < Bottom) || (Top < position.Y + r)) mov.Y *= -1;
          position += mov;
      }
```



今回は、内側に 3 つの境界円を考えます。

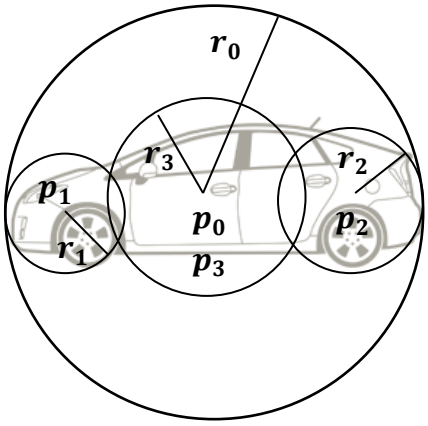
１．外側の境界円の半径 r_0 ・中心座標 $p_0(x, y)$

①外側の境界円の判定メソッドを追加

```
      //追加　壁との衝突判定　（判定のみ）
      private bool AreaMove(Vector2 position, Vector2 mov, float r)
      {
          if ((position.X - r < Left) || (Right < position.X + r)) return true;
          if ((position.Y - r < Bottom) || (Top < position.Y + r)) return true;
          return false;
      }
```

２．内側の境界円の中心を「 p_1 , p_2 , p_3 」　半径「 r_1 , r_2 , r_3 」。

なお、1：前側の円、2：後側の円、3：中側の円とします。



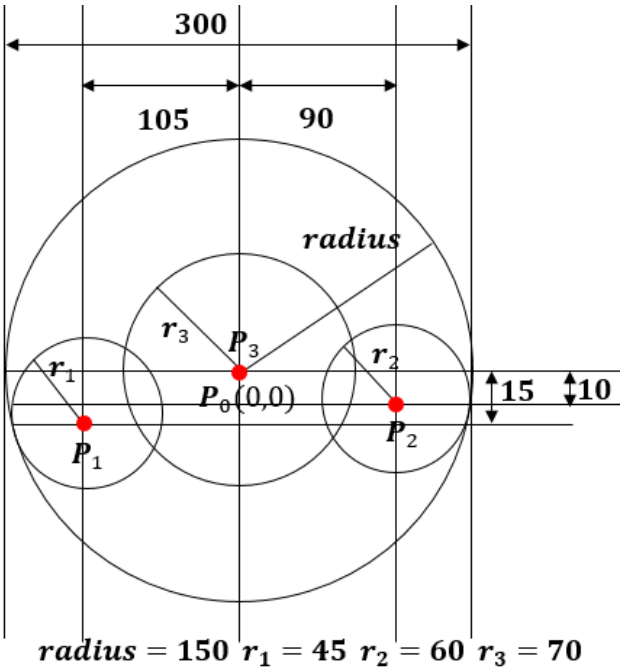
車の境界円

②内側の境界円の判定　メソッドを追加

```
//追加　壁との衝突判定　（衝突したら移動方向を反転させる。）
private bool AreaMove(Vector2 position, ref Vector2 mov, float r)
{
    if ((position.X - r < Left) || (Right < position.X + r))
    {
        mov.X *= -1; return true;
    }
    if ((position.Y - r < Bottom) || (Top < position.Y + r))
    {
        mov.Y *= -1; return true;
    }
    return false;
}
```

この 2 つのメソッドを使って、外側の境界円に衝突していたら、内側の境界円に衝突しているかを判定し、衝突していたら、移動方向を反転させましょう。

３．以下の図を参考にして、フィールド・コンストラクタに追加しましょう。



③変数の定義と初期値

```
//フィールドに追加
float r1, r2, r3;  //内側の境界円の半径
Vector2 p1, p2, p3;  //内側の境界円の中心
//コンストラクタに追加
r1 = 45; r2 = 60; r3 = 70;      //内側の境界円の半径
```

４．メソッド　InputData　の追加・修正

④メソッド　InputData 内に、図を参考にして、p0 に対する p1～p3 の相対的な値を設定

```
p0 = new Vector2(p[0].X, p[0].Y);

//追加　p0 に対する p1～p3 の相対的な値を設定
p1 = new Vector2(- 105, - 15);
p2 = new Vector2(90, - 10);
p3 = new Vector2(0, 0);
```

⑤今回は、車に角度をつけているので、p1～p3 も回転処理します。

```
prius.Rotation(angle);//回転処理

//追加　回転処理
p1 = p1 * Matrix2.Rotate(angle);
p2 = p2 * Matrix2.Rotate(angle);
p3 = p3 * Matrix2.Rotate(angle);
```

⑥内側の境界円を描きます。

```
Circle(p0, radius);  //境界円を描く

//追加　内側の境界円を描く
Circle(p1 + p0, r1);
Circle(p2 + p0, r2);
Circle(p3 + p0, r3);
```

５．Update メソッドの修正

⑦Update メソッドの

AreaMove(ref p0, ref mov, radius);//移動制限

のところを外側の境界円に衝突していたら、次に内側の境界円に衝突しているか判定する処理を [Areamove]メソッドを使って書きましょう。

```
//AreaMove(ref p0, ref mov, radius);//修正　移動制限

//追加
if (AreaMove(p0, mov, radius))
  {
      p0 += mov;
      if((AreaMove(p0+p1,ref mov,r1))
       ||(AreaMove(p0 + p2, ref mov, r2))
       ||(AreaMove(p0 + p3, ref mov, r3)))
      {  p0 += mov;  }
  }
  else
  {  p0 += mov;  }
```

６．Draw メソッドに内側の境界円を描く処理を追加しましょう。

⑧Draw メソッドに追加

```
Circle(p0, radius);//境界円の描画

//追加　内側の境界円の描画
Circle(p1 + p0, r1);
Circle(p2 + p0, r2);
Circle(p3 + p0, r3);
```