【実習４】　衝突処理　　バーとボールの反射

１　ソースコード

（１）テンプレート　Jump の①の箇所に　Jump040View　を入力してください。

（２）次のアプリケーションを新規作成してください。

Jump030View をコピー&ペーストして、ファイル名を「Jump040View」に変更してください。

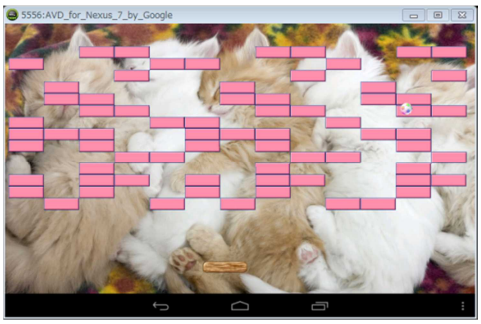
プロジェクト名：JumpPro????(年組席)　アプリケーション名：Jump040View

```
/*    年    組    席   名前
 * Jump040View
 *       バーとボールとの跳ね返り（衝突処理）
 */
・・・
import java.util.Random;

public class Jump040View extends SurfaceView
                         implements SurfaceHolder.Callback
{
    ・・・
    private int barWIDTH;//バーの幅
    private int barHEIGHT;//バーの高さ

    public Jump040View(Context context)
    {
        ・・・
        barWIDTH=bmp[3].getWidth();
        barHEIGHT=bmp[3].getHeight();

        ・・・
    }
    //サーフェイス生成時に呼ばれる
    public void surfaceCreated(SurfaceHolder holder)
    {
        ・・・
    }
    public void draw(Canvas canvas)
    {
        ・・・
    }
    //ボールの移動
    private void moveBall()
    {
        ・・・
        if(ballY>getHeight()-ballHEIGHT)
        {
            ballY=getHeight()-ballHEIGHT;
            ballVY=-ballVY;
        }
```



```
        //バーとの衝突判定
        int r=hitLineRect(
                ballX-ballVX,ballY-ballVY,ballX,ballY,
                barX,barY-ballHEIGHT,barWIDTH,barHEIGHT);
        if (r>=0)
        {
            if(r==0) ballVY=-4-rand(2);//上
            if(r==1) ballVY=-ballVY;//下
            if(r==2 || r==3) ballVX=-ballVX;//左、右
        }
    }
    ・・・
    //タッチイベントの処理
    public boolean onTouchEvent(MotionEvent event)
    {
        ・・・
    }
    //直線と矩形の衝突判定
    private static int hitLineRect(
        int x0,int y0,int x1,int y1,
        int x2,int y2,int w2,int h2)
    {
        if(x2<=x1 && x1<=x2+w2 && y2<=y1 && y1<=y2+h2)
        {
            if(hitLineLine(x0,y0,x1,y1,x2,   y2,   x2+w2,y2   )) return 0;//上
            if(hitLineLine(x0,y0,x1,y1,x2,   y2+h2,x2+w2,y2+h2)) return 1;//下
            if(hitLineLine(x0,y0,x1,y1,x2,   y2,   x2,   y2+h2)) return 2;//左
            if(hitLineLine(x0,y0,x1,y1,x2+w2,y2   ,x2+w2,y2+h2)) return 3;//右
        }
        return -1;
    }
    //2 つの直線の衝突判定
    private static boolean hitLineLine(
        float x0,float y0,float x1,float y1,
        float x2,float y2,float x3,float y3)
    {
        float r=((y3-y2)*(x2-x0)-(x3-x2)*(y2-y0))/
                                    ((x1-x0)*(y3-y2)-(y1-y0)*(x3-x2));
        float s=((y1-y0)*(x2-x0)-(x1-x0)*(y2-y0))/
                                    ((x1-x0)*(y3-y2)-(y1-y0)*(x3-x2));
        return (0<r && r<=1 && 0<s && s<=1);
    }
    //乱数の取得
    private static Random rand=new Random();
    private static int rand(int num) { return (rand.nextInt()>>>1)%num; }
}
```

2　解説

バーと衝突したときは、バーのどの位置に衝突したかによって、跳ね返り方を変えています。

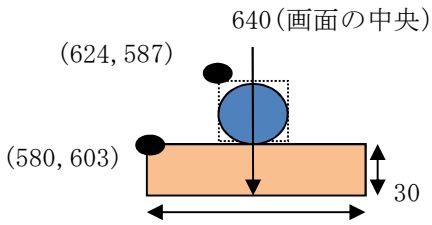
```
    //バーとの衝突判定
    int r=hitLineRect(
                    ballX-ballVX,ballY-ballVY,ballX,ballY,
                    barX,BAR_Y-BMP_2_HEIGHT,BMP_3_WIDTH,BMP_3_HEIGHT);
```

```
    if (r>=0)
    {
        if(r==0) ballVY=-4-rand(2);//上
        if(r==1) ballVY=-ballVY;//下
        if(r==2 || r==3) ballVX=-ballVX;//左、右
    }
          ・
          ・
          ・
//直線と矩形の衝突判定
private static int hitLineRect(
        int x0,int y0,int x1,int y1,
        int x2,int y2,int w2,int h2)
{
    if(x2<=x1 && x1<=x2+w2 && y2<=y1 && y1<=y2+h2)
    {
        if(hitLineLine(x0,y0,x1,y1,x2,    y2,    x2+w2,y2    )) return 0;//上
        if(hitLineLine(x0,y0,x1,y1,x2,    y2+h2,x2+w2,y2+h2)) return 1;//下
        if(hitLineLine(x0,y0,x1,y1,x2,    y2,    x2,    y2+h2)) return 2;//左
        if(hitLineLine(x0,y0,x1,y1,x2+w2,y2    ,x2+w2,y2+h2)) return 3;//右
    }
    return -1;
}
//2 つの直線の衝突判定
private static boolean hitLineLine(
    float x0,float y0,float x1,float y1,
    float x2,float y2,float x3,float y3)
{
    float r=((y3-y2)*(x2-x0)-(x3-x2)*(y2-y0))/
((x1-x0)*(y3-y2)-(y1-y0)*(x3-x2));
    float s=((y1-y0)*(x2-x0)-(x1-x0)*(y2-y0))/
((x1-x0)*(y3-y2)-(y1-y0)*(x3-x2));
    return (0<r && r<=1 && 0<s && s<=1);
}
//乱数の取得
private static Random rand=new Random();
private static int rand(int num) { return (rand.nextInt()>>>1)%num; }
```

乱数を取得するには**Random クラス**を使います。

使い方は、Random オブジェクトを生成して、nextInt()メソッドを呼ぶだけです。これで、int 型の最小値から最大値(-2147483648～2147483647) の乱数を取得することができます。

取得した乱数を＞＞＞演算子を使って、１ビットだけ符号なしビットシフトを行えば、最上位ビットが０になるので、正数の乱数（０～2147483647）を取得することができます。さらに、正数の乱数を任意の値で割った余りを求めれば、０から任意の正数未満の乱数を取得することができます。



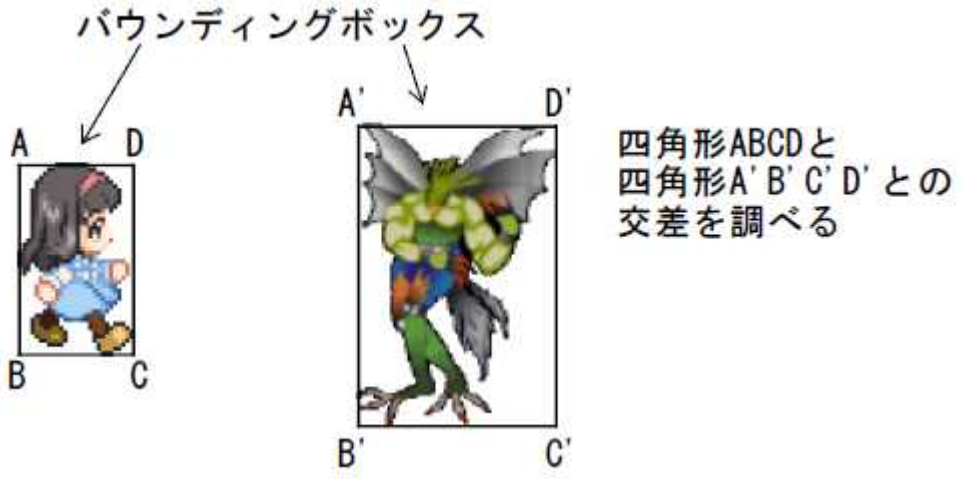
３　衝突検出

ゲームの中で、キャラクタ同士の衝突検出(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、頻繁に必要になります。シューティングゲームでは、プレイヤーと敵、プレイヤーの弾と敵、プレイヤーと敵の弾など、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３０回の移動が行われる(30FPS)とすると、そのぶんだけ衝突検出が必要となります。

3.1　バウンディングボックス

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、キャラクタをすっぽりと包むような矩形にします。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。

3.2　衝突判定のプログラミング

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
```

スクリーン座標系のｙ軸は、下方向に向いていることに注意しましょう。条件式を

まとめると、
　・頂点Ａが頂点Ｃ’の左側にあり、かつ、頂点Ｃが頂点Ａ’の右側にある
　・頂点Ａが頂点Ｃ’の上側にあり、かつ、頂点Ｃが頂点Ａ’の下側にある
となります。この２つの条件を満たしたとき、２つの領域は交差しています。この条件を言い換えれば、四角形ＡＢＣＤの一部または全部が、四角形Ａ’Ｂ’Ｃ’Ｄ’の中にあるかどうかということです。

逆に言えば、
　・頂点Ａが頂点Ｃ’の右側にある
　・頂点Ｃが頂点Ａ’の左側にある
　・頂点Ａが頂点Ｃ’の下側にある
　・頂点Ｃが頂点Ａ’の上側にある
の条件を１つでも満たすと、絶対に交差していないということになります。これを Java 言語で記述すると、次のようなプログラムになります。

```
// 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’)

if(ax2 < bx1 || ax1 > bx2 || ay1 > by2 || ay2 < by1)
    // 衝突していない
else
    // 衝突している_
```

3.3　バウンディングサークル

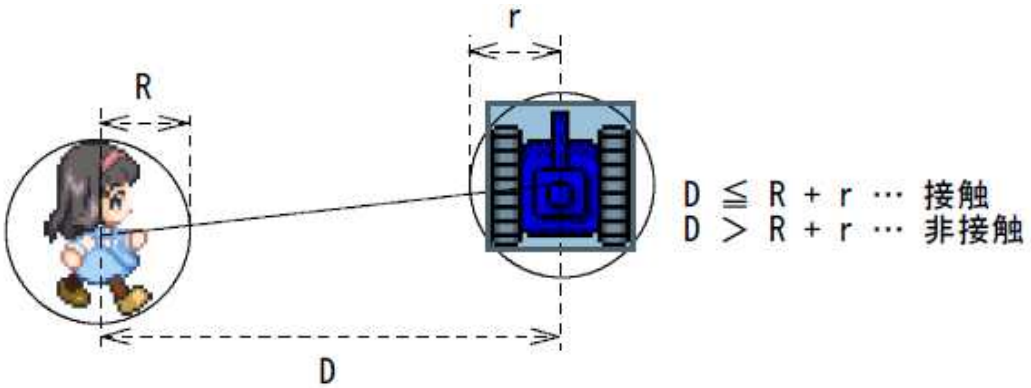
バウンディングボックスを円にすると、その中心と半径を定義するだけで扱うことができます。キャラクタの衝突を検出する場合には、それぞれの衝突範囲を定義している円の中心同士の距離を測定し、これが各半径の和以下であれば、このふたつのキャラクタは衝突しているとみなします。

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、D2 と $(R + r)^2$ の大小比較で判定を行うことができます。

$D2 \leqq (R + r)^2$ … 接触
$D2 > (R + r)^2$ … 非接触



▲円の衝突検出

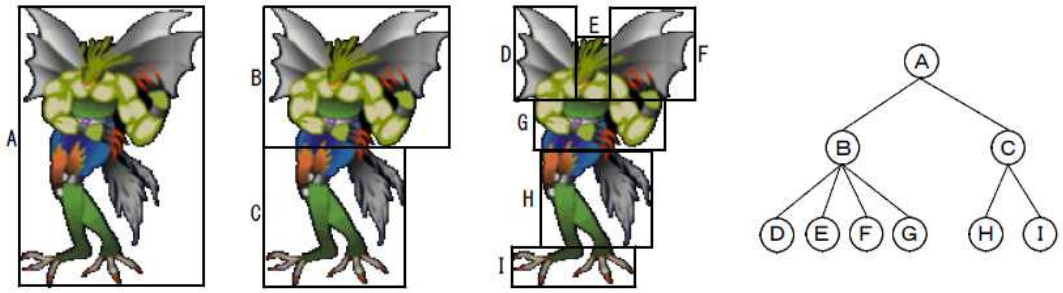
3.4　バウンディングボックスの細分化

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。

これを極力減らすには、バウンディングボックスの細分化を考えます。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