

# Algorithme de détection de Collision entre deux rectangles orientés

Le 07/05/2010

## **Auteur :**

RADLO Valentin

[valentinradlo@hotmail.fr](mailto:valentinradlo@hotmail.fr)

## **Démonstration :**

THILLOU Damien

[thillou\\_damien@yahoo.fr](mailto:thillou_damien@yahoo.fr)

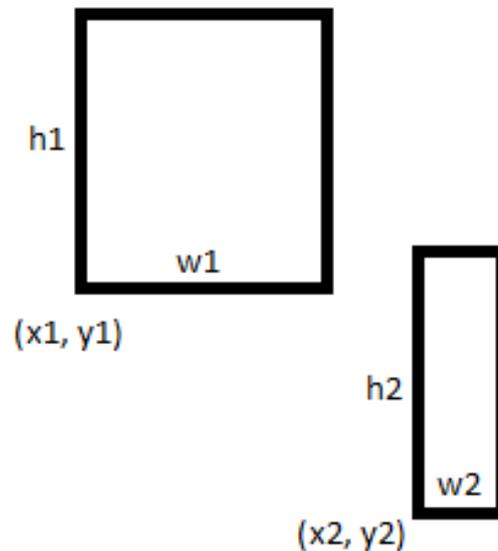
## **Introduction :**

La détection de collision intervient dans de nombreux domaines d'applications. C'est un problème qui prend toute son importance en informatique, dans la mesure où elle peut être exécutée des millions de fois par secondes dans une application en temps réel. Plusieurs algorithmes permettent déjà de détecter la collision entre deux rectangles orientés ; je propose dans cette publication une nouvelle méthode qui permet de voir le problème sous un autre angle. Cette publication a pour but de protéger tous les utilisateurs de cette méthode d'un éventuel brevet.

## **Méthode :**

Avant de rentrer dans le vif du sujet, un rappel :

Le cas de deux rectangles non-orientés :



L'algorithme de collision (le plus simple à l'heure actuelle) s'écrit en pseudo code:

*Soit deux rectangles non-orientés de positions respectives  $(x1, y1)$  et  $(x2, y2)$  de largeurs respectives  $w1$  et  $w2$  et de hauteurs respectives  $h1$  et  $h2$ .*

***Si*** (  $x2 > x1 + w1$  **OU**  $x1 > x2 + w2$  **OU**  $y2 > y1 + h1$  **OU**  $y1 > y2 + h2$  )

***Alors*** « il n'y a pas de collision »

***Sinon*** « il y a collision »

À noter que j'ai choisi arbitrairement la position (x, y) en bas à gauche du rectangle, une position centrale aurait légèrement changé la formule.

Peu importe la méthode utilisée, celle-ci sera reprise par la suite dans la fonction TestCollisionNonOrienté ainsi définie :

**Si** A et B sont en collision **alors** TestCollisionNonOrienté(A, B) = VRAI

**Sinon** TestCollisionNonOrienté(A, B) = FAUX

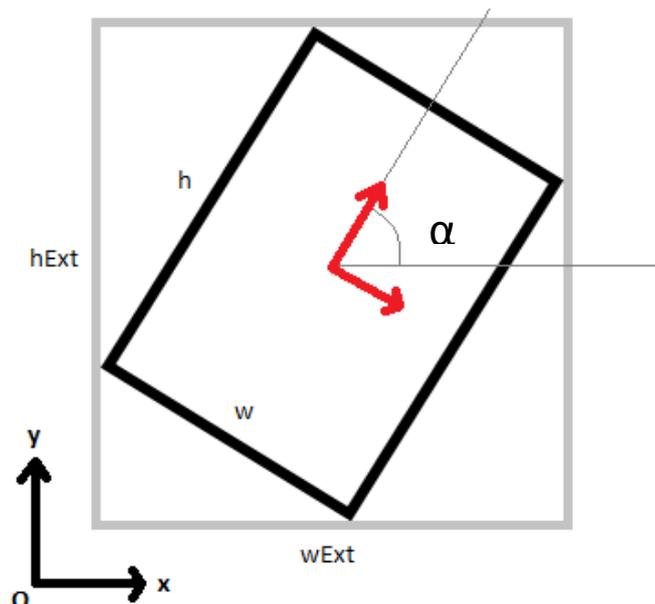
L'idée de la méthode, est de réutiliser ce que l'on sait très bien faire pour des rectangles non-orientés, afin de l'adapter aux rectangles orientés.

Nous introduisons pour cela deux nouvelles notions :

- La notion de rectangle externe qui sera définie selon un repère orthogonale Oij
- La notion de repère local qui aura pour origine le centre du rectangle et orienté comme ce dernier

La figure suivante nous montre donc :

- En noir, le rectangle orienté
- En gris, le rectangle externe selon le repère orthogonale Oxy
- En rouge, le repère local



Dans cet exemple, on obtient :

$$wExt = w * |\cos(\alpha)| + h * |\sin(\alpha)|$$

$$hExt = w * |\sin(\alpha)| + h * |\cos(\alpha)|$$

Détaillons maintenant l'algorithme en pseudo code que démontrerons ensuite:

**Algorithme :**

*Soit A et B deux rectangles orientés, et notons Aext et Bext leurs rectangles externes.*

**Si**

*dans le repère local de A, TestCollisionNonOrienté(A, Bext) = VRAI*

**ET**

*dans le repère local de B, TestCollisionNonOrienté(Aext, B) = VRAI*

**Alors** « il y a collision »

**Sinon** « il n'y a pas collision »

## Démonstration (Par Damien THILLOU) :

A et B sont en collision est équivalent à écrire :  $A \cap B \neq \emptyset$

### Première implication:

Démontrons que :

$$A \cap B \neq \emptyset \Rightarrow (A \cap Bext_{Ref(A)} \neq \emptyset) \& (Aext_{Ref(B)} \cap B \neq \emptyset)$$

où  $Aext_{Ref(B)}$  désigne le rectangle externe de A lorsque l'on se place dans le repère local de B

$$\text{Soit } A \cap B \neq \emptyset, \text{ or on a évidemment } \begin{cases} A \subset Aext_{Ref(B)} \\ B \subset Bext_{Ref(A)} \end{cases}$$

Par conséquent, on a bien :  $(A \cap Bext_{Ref(A)} \neq \emptyset) \& (Aext_{Ref(B)} \cap B \neq \emptyset)$

### Deuxième implication:

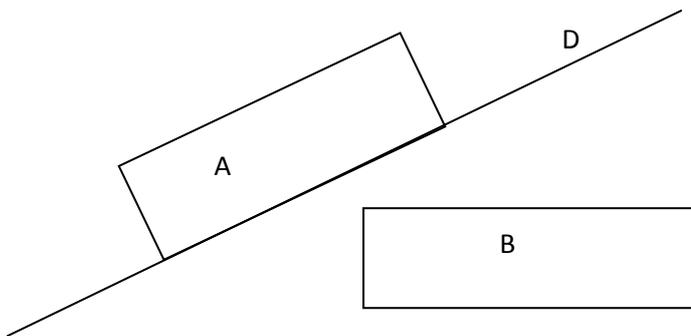
Démontrons que :

$$(A \cap Bext_{Ref(A)} \neq \emptyset) \& (Aext_{Ref(B)} \cap B \neq \emptyset) \Rightarrow A \cap B \neq \emptyset$$

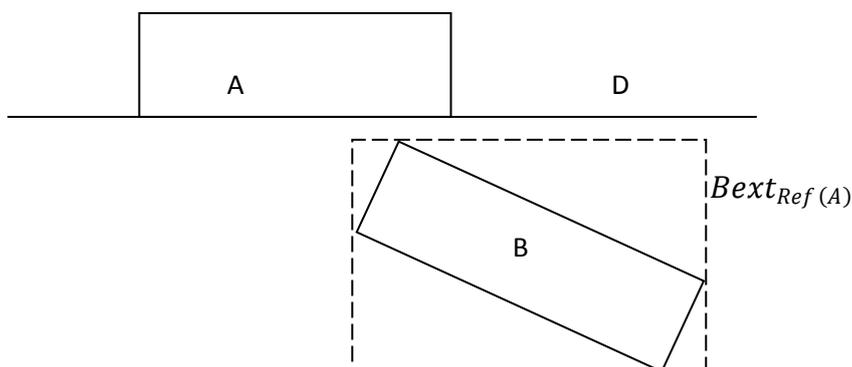
Par l'absurde, on suppose qu'il n'y a pas collision. Montrons que l'une des deux intersections est vide.

Nous manipulons des rectangles, polygones convexes, nous pouvons donc appliquer la méthode des axes séparateurs qui nous dit que A et B ne sont pas en collision si et seulement s'il existe un axe séparateur défini par l'un des côtés des polygones, à savoir ici l'un des côtés de A ou de B.

Prenons le cas suivant : l'axe séparateur porté par un côté de A (Cas n°1).



Ce qui donne dans le repère local de A :



A noter que la droite Delta pourrait également être verticale, ce qui ne change pas le raisonnement suivant.

Alors dans ce cas, comme les 4 sommets de B sont du même côté de l'axe séparateur Delta, sa boîte englobante est elle aussi de l'autre côté de l'axe et donc l'intersection  $A \cap B_{\text{ext}_{\text{Ref}(A)}} = \emptyset$

Si jamais c'est une arête de B qui porte l'axe séparateur (Cas n°2), on arrive à  $A_{\text{ext}_{\text{Ref}(B)}} \cap B = \emptyset$

D'où la conclusion par l'absurde.

### Application :

Le code suivant écrit en C++, implémente nos deux types d'objet, les rectangles Orientés et Non-Orientés ainsi que deux méthodes permettant de savoir si deux rectangles du même type sont en collision. La méthode pour les rectangles orientés utilise évidemment la méthode décrite au dessus.

A noter que le code ci-dessous n'est pas optimisé pour plus de compréhension.

### Fichier rect.h :

```
#ifndef _RECT_
#define _RECT_

#define DEG_TO_RAD 0.0174532925
#define RAD_TO_DEG 57.29577957

class rect
{
public:

    double oX, oY;    //coordonnée du centre du rectangle
    double w, h;      //largeur et hauteur

    rect(double oX = 0, double oY = 0,
          double w = 1, double h = 1);
    friend bool Collision(rect* pRect1, rect* pRect2);
};

#endif
```

### Fichier rect.cpp :

```
#include "rect.h"
#include "stdio.h"
#include <cmath>

rect::rect(double oX, double oY, double w, double h)
: oX(oX), oY(oY), w(w), h(h) {}

bool Collision(rect* pRect1, rect* pRect2)
{
    double offsetX1 = pRect1->oX - pRect1->w/2.;
    double offsetY1 = pRect1->oY - pRect1->h/2.;
    double offsetX2 = pRect2->oX - pRect2->w/2.;
    double offsetY2 = pRect2->oY - pRect2->h/2.;
    return !(offsetX1 > offsetX2 + pRect2->w
            || offsetX2 > offsetX1 + pRect1->w
            || offsetY1 > offsetY2 + pRect2->h
            || offsetY2 > offsetY1 + pRect1->h);
}
```

## Fichier orect.h :

```
#ifndef _ORECT_
#define _ORECT_

#include "rect.h"

class orect : public rect
{
public:

    rect outLineRect; //boite englobante
    double angle;     //en degré

    orect(double oX = 0, double oY = 0,
           double w = 1, double h = 1, double angle = 0);

    friend bool Collision(orect* pRect1, orect* pRect2);

private:
    void UpdateOutLineRect_private();
    void RotateAround_private(const orect* base);
};

#endif
```

## Fichier orect.cpp :

```
#include "orect.h"
#include "stdio.h"
#include <cmath>

orect::vm_orect(double oX, double oY, double w, double h, double angle)
:rect(oX,oY,w,h),angle(angle){}

bool Collision(orect* pRect1, orect* pRect2)
{
    orect rectA(*pRect1);
    orect rectB(*pRect2);

    rectB.RotateAround_private(&rectA);

    if(!Collision(&rectA,&rectB.outLineRect))
        return false;

    rectB = *pRect2;

    rectA.RotateAround_private(&rectB);

    if(!Collision(&rectA.outLineRect,&rectB))
        return false;

    return true;
}
```

```

/*****
PRIVATE FUNCTION
*****/

void orect::UpdateOutLineRect_private()
{
    double absCosA = abs(cos( angle * DEG_TO_RAD));
    double absSinA = abs(sin( angle * DEG_TO_RAD));
    outLineRect.w = w * absCosA + h * absSinA;
    outLineRect.h = w * absSinA + h * absCosA;
    outLineRect.oX = oX;
    outLineRect.oY = oY;
}

void orect::RotateAround_private(const orect* base)
{
    double cosA = cos( -base->angle * DEG_TO_RAD);
    double sinA = sin( -base->angle * DEG_TO_RAD);

    double oXDif = oX - base->oX;
    double oYDif = oY - base->oY;

    double oXDif2 = oXDif * cosA - oYDif * sinA;
    double oYDif2 = oXDif * sinA + oYDif * cosA;

    oX = base->oX + oXDif2;
    oY = base->oY + oYDif2;
    angle -= base->angle;
    UpdateOutLineRect_private();
}

```

### Généralisation à 3 dimensions :

De la même manière, cette méthode est facilement transposable en 3 dimensions, pour des parallélépipèdes rectangles. Il suffira de prendre des boîtes englobantes à la place de nos rectangles externes dont nous savons facilement déterminer s'ils sont en collision ou non, puis à l'aide des formules de changement de repère, réutiliser l'algorithme.