C - Création d'un système de physique avec SDL

La création d'un système de physique implique que nous avons déjà un moyen de gérer des objets dans un espace. Cette note s'attarde sur la création de comportement lié à la physique (comme la gravité, les collisions et les rebonds) et non comment représenter des composante physique virtuellement.
Pour cela voici la note correspondante : C - Créer un objet physique avec SDL (pas fini)

La gravité

Pour créer de la gravité, c'est très simple, il suffit qu'à un intervalle régulier de temps (chaque image), la vitesse verticale d'un objet augmente.
Pour cela il suffit d'avoir une variable représentant cette vitesse, puis d'y ajouter une valeur définie tel que dans le code suivante :

```
speed += 0.2;
personnage->rect->y += speed;
```

Et voila ! Notre personnage tombe.
Il faut noter que cette approche dépend du notre de fois que cette fonction est exécuté par seconde. Le plus, le plus vite sera la chute. Pour palier à cela il nous faudrait une variable contenant le temps depuis la dernière image (connu sous le nom de DeltaTime), et multiplier notre augmentation de vitesse par celle-ci (afin que si le nombre d'image par seconde est élevé, la distance parcouru soit petite, et inversement).
Cependant obtenir un tel nombre a ses propres défie technique, et c'est pour cela que dans la suite de ce document nous assumerons que le nombre d'image par seconde est fixe.

Les collisions

Le premier code permettant de gérer des collisions était celui-ci :

```
speed += 0.2;
if (personnage->rect->y >= 500) {
    personnage->rect->y = 500;
    speed = 0;
}
personnage->rect->y += speed;
render(personnage);
```

Il se basait sur une variable speed (qui représentais la vitesse verticale) qui était ajouté au personnage à chaque frame, le faisant ainsi tomber en accélérant.
En guise de sol, nous avions les coordonnées Y=500 auquel le personnage était ramené si il dépassait.

Cependant cette méthode amenais un léger problème :
Quand le personnage tombe de haut, sa vitesse faisait qu'il dépassait visiblement la barrière des 500, et était ramené à l'image d'après, donnant un effet de rollback.

La correction à ceci à été d'anticiper la prochaine position du personnage à l'image d'après pour le placer directement à la bonne position (nous aurions pu vérifier de nouveau la position Y du personnage avant de le render, mais cela reviens un peu au même)
Ce concept d'anticipation de la prochaine position sera un fondement dans la suite du développement de ce système de collision.

Voici le code implémentant cette idée :

```
speed += 0.2;
if (personnage->rect->y + speed >= 500) {
    personnage->rect->y = 500;
    speed = 0;
}
personnage->rect->y += speed;
render(personnage);
```

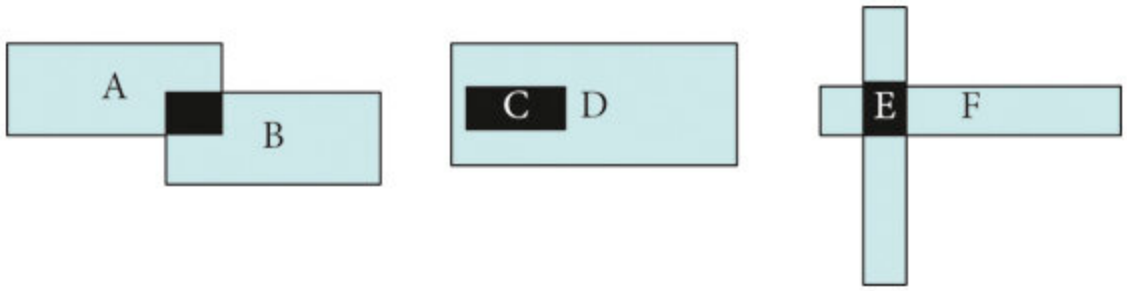
(Le "+ speed" après la position Y du perso, dans le if, modification discrète mais très efficace)

A partir de là, nous pouvons améliorer la structure d'un object de manière à ce qu'elle puisse stocker les valeurs de vitesse X et Y de l'object.
Et nous créerons une fonction (GetNextPosition) utilisant cela afin de retourner le Rect de la prochaine position de notre object.

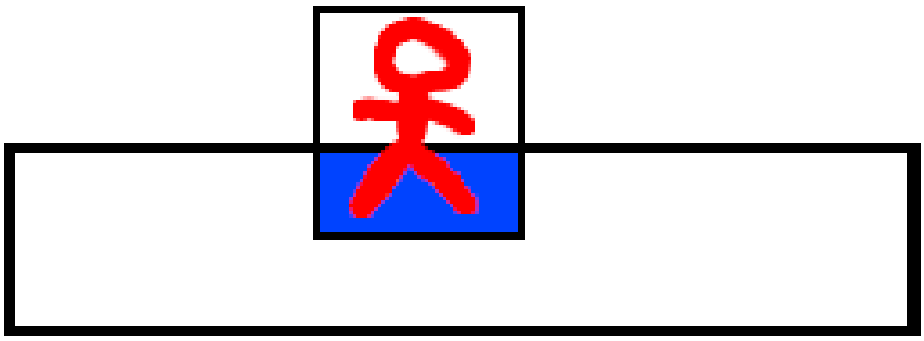
Maintenant, la prochaine étape sera d'améliorer ce avec quoi notre joueur a une collision. Pour l'instant nous utilisons simplement une hauteur prédéfinie dans le code, alors essayons d'utiliser un autre object du jeu !

Nous attaquons donc les collisions "Objet - Objet" :
Pour savoir si un objet est en collision avec un autre, nous devons savoir si ils se chevauche, en d'autres termes, s'il y a une intersection entre eux.
Comme nous n'utilisons uniquement des rectangles pour l'instant, nous pouvons utiliser de manière très pratique la fonction SDL nommée "SDL_IntersectRect" , qui permet de savoir s'il y a une intersection entre deux Rect, et si oui d'avoir le rectangle représentant cette intersection

comme montré dans le schéma suivant :



Dans notre cas, nous aurons un personnage, qui intersectionne avec un rectangle qui représente le sol, comme suit :



Nous constatons donc la collision entre ces deux objets, et la fonction SDL_IntersectRect nous retournerais bien TRUE, de plus récupérerions aussi l'équivalent du rectangle bleu ici, qui représente l'intersection de ces deux Rect.
Additionellement, ce shéma ne représente pas l'utilisation de la simulation de prochaine position que nous avons créé plus tôt. Dans les faits à, un état de repos le personnage serait situé sur le sol, et gagnerai à chaque frame de la vélocité verticale (du à la gravité). Cela déplacerai sa boite de prochaine position dans le sol, permettant ainsi de détecter la collision, et annulerai la vitesse verticale gagné -> faisant donc effectivement rester le personnage immobile sur le sol, l'empêchant de le traverser.

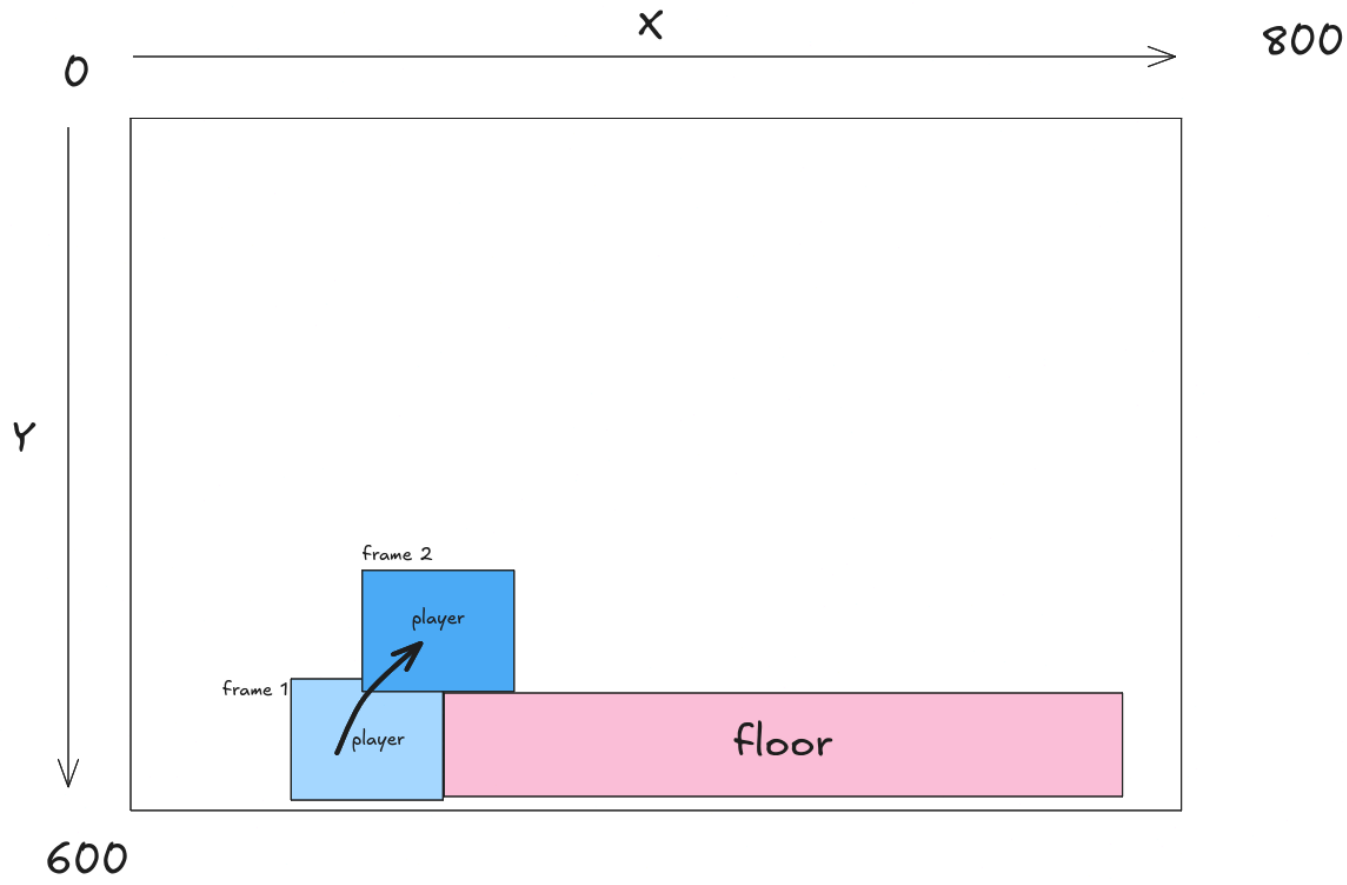
Dans notre code, tout ce que nous aurons à faire c'est détecter avec la fonction SDL_IntersectRect s'il y à une collision entre la future position de notre personnage, et l'objet collisioné. Et si c'est le cas, déplacer le joueur au dessus de celui-ci.
(personnage->y = obstacle->y - personnage->height)
OU (personnage->y = personnageNextPos->y - IntersectRect->height)

```
SDL_Rect* intersect = (SDL_Rect*) malloc(sizeof(SDL_Rect));
SDL_Rect* colliderNextPos = GetNextPos(collider); //N'oublions pas l'astuce
d'utiliser la prochaine position de l'objet.

SDL_bool hasCollided = SDL_IntersectRect(colliderNextPos, collideePos,
```

```
intersect);

if (hasCollided == SDL_TRUE) {
    collider->rect->y = collidee->rect->y - collider->rect->h;
}
```

Maintenant, nous avons enfin un système permettant au personnage de tomber sur un objet, et d'y rester sans bouger !

Cependant, nouveau problème :)
Le code présenté ci-dessus engendrerais des situation comme celle-ci :

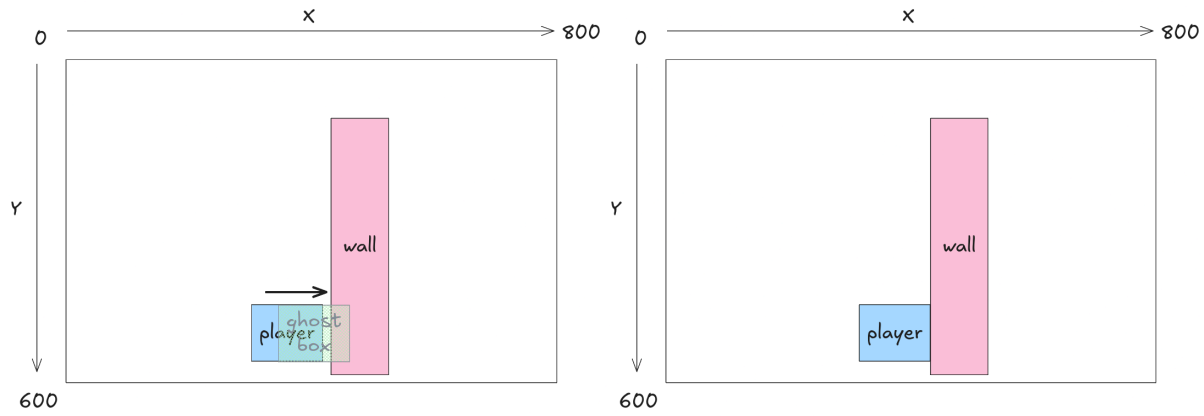


Si notre joueur entre en collision depuis le côté avec ce qui représente le sol, il se fait téléporter au dessus de celui-ci.
Ce n'est de toute évidence absolument pas le comportement que nous désirons, nous devons donc adapter notre code afin qu'il puisse être plus généraliste.
L'objectif est de permette de gérer proprement les collisions, qu'elles viennent aussi bien du côté, dessus ou dessous !

Mais pour commencer cantonnons nous à faire fonctionner les collisions pour 1 axe.
Notre code actuel fonctionne pour une collision sur 1 seul axe, ET en venant d'un seul côté de

cet axe. En effet, si nous entrons en collision avec le sol que nous avons codé juste au dessus, nous nous faisons téléporter dessus ce dernier.

Pour commencer la résolution de ce problème, changeons d'abord d'axe. L'axe Y que nous utilisions jusqu'à présent était intuitif du point de vu de la gravité, cependant le fait que sa valeure augmente en déscendant, ne l'est pas.
Prenons donc l'exemple d'un mur, et définissons les termes de l'explication :
Dans les exemples qui suivent, nous ferons référence à la prochaine position du joueur en tant que "GhostBox", celle-ci est simplement une projection de la position du joueur à partir de sa vélocité actuelle.



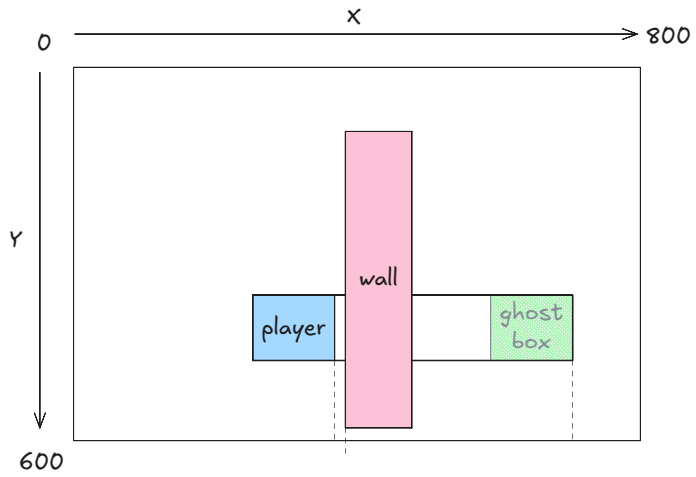
Très bien, grace à ce que nous avons définie précédemment, cette collision serai résolue en plaçant le player juste collé au mur.

Voici le code complet implémentant les collisions de tous côtés, avec plusieurs objets simultanément :

```
WindowElement* collider = personnage;
SDL_Rect* colliderNextPos = GetNextPosition(collider);

// Sachant que obs représentes les objets du monde
for (unsigned i = 0; i < obs->lenght; ++i) {
    WindowElement* collidee = obs->objects + i;
    //mieux que &obs->objects[i]
    SDL_Rect* collideeNextPos = GetNextPosition(collidee);

    SDL_Rect* intersect = (SDL_Rect*) malloc(sizeof(SDL_Rect));
    SDL_bool hasCollided = SDL_IntersectRect(colliderNextPos,
collideeNextPos, intersect);

    if (hasCollided == SDL_TRUE) {

        if (collider->rect->y + collider->rect->h <= collidee->rect->y)
        {
```

```
            collider->vY = 0;

            colliderNextPos->y -= intersect->h;
            //collider->rect->y = colliderNextPos->y;
        }
        else if (collider->rect->y >= collidee->rect->y + collidee->rect->h)
        {
            collider->vY = 0;
            //collider->rect->y = colliderNextPos->y + intersect->h;

            colliderNextPos->y += intersect->h;
            //collider->rect->y = colliderNextPos->y;
        }

        if (collider->rect->x + collider->rect->w <= collidee->rect->x)
        {
            collider->vX = 0;

            colliderNextPos->x -= intersect->w;
            //collider->rect->x = colliderNextPos->x;
        }
        else if (collider->rect->x >= collidee->rect->x + collidee->rect->w)
        {
            collider->vX = 0;

            colliderNextPos->x += intersect->w;
            //collider->rect->x = colliderNextPos->x;
        }
    }

    collidee->rect->x = collideeNextPos->x;
    collidee->rect->y = collideeNextPos->y;
}
collider->rect->x = colliderNextPos->x;
collider->rect->y = colliderNextPos->y;
```

Cette aproche est parfaitement fonctionnelle pour une utilisation classique, cependant elle a certaines limites dans des cas extremes. Imaginons que le Player aille à une vitesse très élevé, il serait possible que sa "GhostBox" passe directement derrière l'obstacle et par conséquent que le joueur le traverse.
Voici une schéma illustrant ceci :



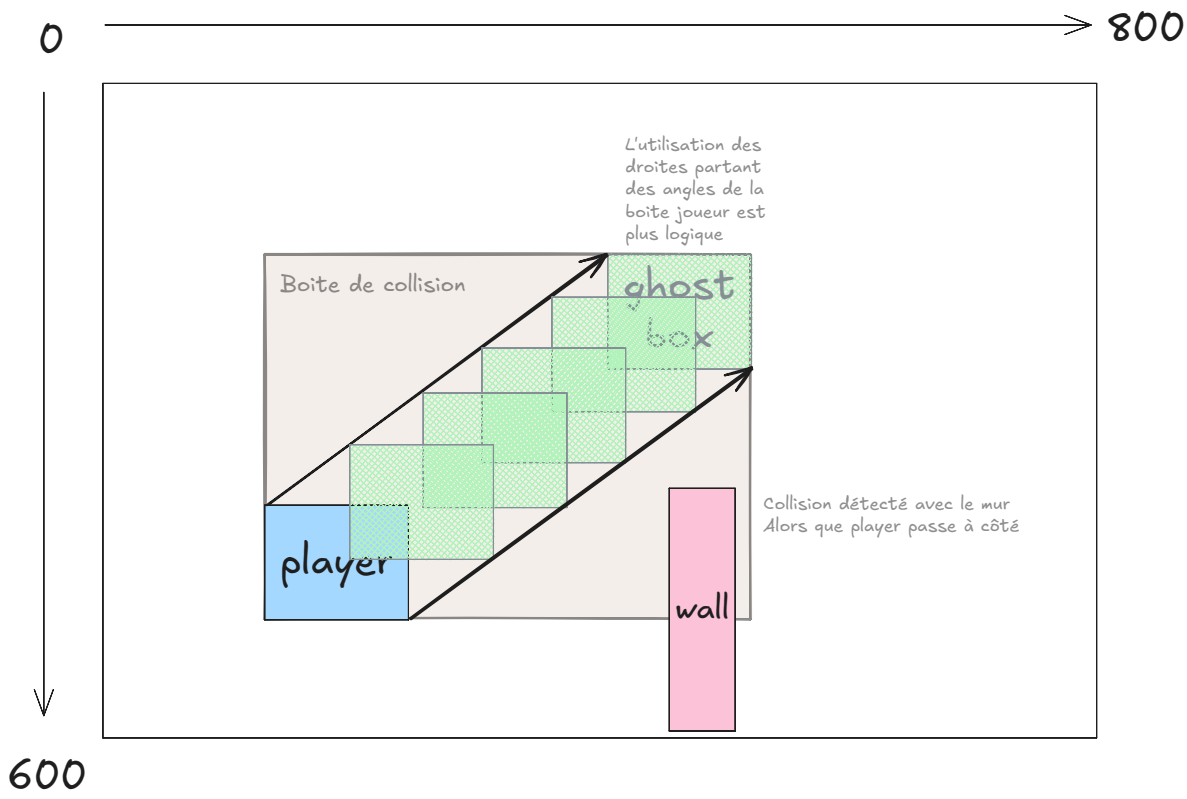
Nous pouvons essayer d'implémenter une solution avec SDL_UnionRect permettant de récupérer le rectangle contenant le joueur jusqu'à sa GhostBox (position future).

Implémentation de l'utilisation d'un SDL_UnionRect et changement de méthode réconciliation, + utilisation de Rect flottant :

```
WindowElement* collider = personnage;
SDL_FRect* colliderNextPos = GetNextPosition(collider);

for (unsigned i = 0; i < obs->lenght; ++i) {
    WindowElement* collidee = obs->objects + i;
    //mieux que &obs->objects[i]
    SDL_FRect* collideeNextPos = GetNextPosition(collidee);

    SDL_FRect* deltaBox = (SDL_FRect*) malloc(sizeof(SDL_FRect));
    SDL_UnionFRect(collider->rect, colliderNextPos, deltaBox);
    SDL_bool hasCollided = SDL_HasIntersectionF(deltaBox, collideeNextPos);

    if (hasCollided == SDL_TRUE) {

        if (collider->rect->y + collider->rect->h <= collidee->rect->y)
        {
            collider->vY = 0;

            colliderNextPos->y = collideeNextPos->y - collider->rect->h;
            //collider->rect->y = colliderNextPos->y;
        }
        else if (collider->rect->y >= collidee->rect->y + collidee->rect->h)
        {
            collider->vY = 0;
            //collider->rect->y = colliderNextPos->y + intersect->h;
```

```
            //colliderNextPos->y += intersect->h;
            colliderNextPos->y = collideeNextPos->y + collideeNextPos->h;
            //collider->rect->y = colliderNextPos->y;
        }

        if (collider->rect->x + collider->rect->w <= collidee->rect->x)
        {
            collider->vX = 0;

            //colliderNextPos->x -= intersect->w;
            colliderNextPos->x = collideeNextPos->x - collider->rect->w;
            //collider->rect->x = colliderNextPos->x;
        }
        else if (collider->rect->x >= collidee->rect->x + collidee->rect->w)
        {
            collider->vX = 0;

            //colliderNextPos->x += intersect->w;
            colliderNextPos->x = collideeNextPos->x + collideeNextPos->w;
            //collider->rect->x = colliderNextPos->x;
        }
    }

    collidee->rect->x = collideeNextPos->x;
    collidee->rect->y = collideeNextPos->y;
}
collider->rect->x = colliderNextPos->x;
collider->rect->y = colliderNextPos->y;
```

Après avoir testé cette méthode je vois un problème et en théorise un autre.
Le problème que je constate est que si le personnage est sur une plateforme qui monte, il ne peut plus sauter. Après investigation cela est du au fait que dans ce code la boite d'union comprend la position actuelle du personnage et n'est pas basé uniquement sur ses position future. Cela à pour effet de créer un collision alors que dans le futur il n'y en aurait pas eu.
Ce problème peut être mitigé en enlevant la largeur et hauteur du Rect du perso a cette box et la décallant dans la bonne direction.
Cependant ce problème m'a fait penser à un autre, ce système de boite d'union est présent pour avoir des collisions continue même si l'objet va à une haute vitesse. Cependant si l'objet va en diagonale la boite va s'étendre dans les deux diagonales tangente et potentiellement collisioner avec un mur alors que l'objet serait simplement passé à côté avec un calcul normal.
Pour régler cela il faudrait utiliser des raycast afin de vérifier si une intersection LINEAIRE existe. Voir les schéma suivant :



Le système initial étant déjà suffisant pour ce que nous faisons, et n'ayant pas un temps illimité pour expérimenter, nous allons revenir à la version précédente du système de collision qui fait déjà suffisamment l'affaire.
Cependant nous savons que si nous nécessitons éventuellement d'une version plus robuste, nous avons le modèle ici.

Le rebond

Implémenter un système de rebond une fois les bases physique posée est plutôt simple.
Il nous suffis de définir un coefficient de rebond pour les objets du jeu, et de multiplier la vitesse du joueur sur l'axe de la collision par le négatif de ce coefficient; inversant ainsi sa direction d'un facteur définie.
Exemple :
Le joueur avance de 5 vers la droite (mouvement sur X), il rencontre un obstacle avec un coefficient de rebond de 1.
Nous avons donc `vitesse_joueur * -(coeff_rebond) = new_vitesse_joueur`,
donc ici `5 * -1 = -5`, notre joueur ira donc dans l'autre sens sans perte de vitesse, soit le comportement attendu.

Voici ce que cela donne en code :

```
// ...
// Si collision :
```

```
// Coefficient de rebond :
// 0 -> Aucun rebond
// 1 -> Rebond total, aucune perte de momentum
float bounciness = 1;

// Si collision sur axe Y, venant du dessus
if (collider->rect->y + collider->rect->h <= collidee->rect->y)
{
    // Négation de la vitesse Y et multiplication par le coefficient de
rebond
    collider->vY *= -bounciness;

    // Gestion de la collision
}
// Gestion de l'axe Y venant du dessous et axe X similaire
// ...
```