Bomberman

Rapport du projet de programmation impérative

Quentin Januel
Clément Defrétière

8 décembre 2018

Résumé

Pendant toute la réalisation du projet, les collisions ont été la plus importante difficulté à laquelle nous avons été confronté. Il nous semble donc pertinent de leur consacrer ce rapport. Il y aura donc de nombreux points que l'on se contentera de taire afin de garder ce document aussi concis que possible, cependant lire le code source suffit à assimiler ces derniers.

Les collisions sont sources de beaucoup de problèmes dans les jeux vidéos. Dans le cadre de notre Bomberman, nous avions naïvement commencé par les coder de la façon suivante :

```
1: nextPosition ← getNextPositionFromInputs()
2: if playerCollidesAt(nextPosition) then
3:     while notplayerCollidesAt(onePixelToward(nextPosition)) do
4:         movePlayerTo(onePixelToward(nextPosition))
5:     end while
6: else
7:     movePlayerTo(nextPosition)
8: end if
```

C'est-à-dire que le programme empêchait formellement au joueur d'être en collision avec un obstacle : on vérifie d'abord s'il y a un obstacle là où le joueur est censé se rendre, si non, il peut bouger, si oui on le fait bouger dans cette direction pixel par pixel tant qu'aucune collision ne pose problème.
Pour être plus précis, notre code séparait ce pseudo-code en deux parties : le movement horizontal et celui vertical, de sorte qu'il n'y ait pas de trigonométrie impliquée.

Le joueur a une vitesse x et bouge donc de x pixels par frame. De plus, la hitbox du joueur est un carré de $n \times n$ pixels et ce dernier se déplace dans des couloirs ayant exactement ses dimensions, ce qui produit le problème suivant :

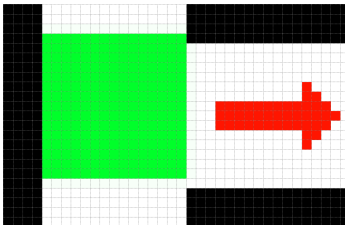

Fig. 1

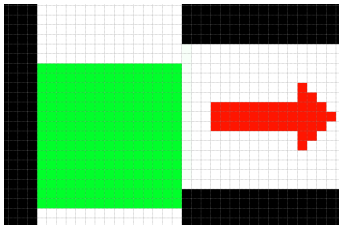

Fig. 2

Le joueur (le carré vert) se trouve en **Fig. 1** et cherche à rentrer dans le couloir indiqué par la flèche rouge. Il lui est impossible d'aller à droite car il serait alors en collision avec l'obstacle noir et le programme interdit ça. Cependant, s'il descend pendant une frame, sa coordonnée verticale augmente de $x = 3$ pixels (voir **Fig. 2**) ce qui ne lui permet toujours pas de rentrer dans le couloir. Le joueur ne pourra donc jamais y accéder. Pour y remédier, nous avons imaginé deux astuces.

La première consiste à ne pas bouger de x pixels par frame mais de seulement 1. Ensuite, répéter la procédure x fois par frame. Au final, le joueur aura bien bougé de x pixels mais ce en x fois et comme les inputs ne sont récupérés qu'une fois par frame, le jeu laissera passer le joueur dans le

couloir si ce dernier appuie sur droite et bas en même temps depuis **Fig. 1**.

La seconde est un tout autre algorithme de collision :

```
1: tempPosition ← getNextPositionFromInputs()
2: movePlayerTo(tempPosition)
3: if playerIsColliding() then
4:     minDistance ← +∞
5:     angleOfMinDistance ← 0
6:     for angle = 0; angle < 360; angle ← angle + 45 do
7:         currentDistance ← 0
8:         while playerIsColliding()andcurrentDistance ≤ speed do
9:             newPosition ← tempPosition + vector(1, angle))
10:            movePlayerTo(newPosition)
11:            currentDistance ← currentDistance + 1
12:        end while
13:        if currentDistance < minDistance then
14:            minDistance ← currentDistance
15:            angleOfMinDistance ← angle
16:        end if
17:        movePlayerTo(tempPosition)
18:    end for
19:    newPosition ← tempPosition+vector(minDistance, angleOfMinDistance))
20:    movePlayerTo(newPosition)
21: end if
```

Le principe est de laisser le joueur se déplacer, et seulement ensuite le faire sortir de l'obstacle en le poussant dans la direction la moins couteuse en distance. Cependant, l'algorithme tel quel ne convient pas : en effet si le joueur se trouve sous un obstacle et tente de monter (voir **Fig. 3 & 4**), le jeu le repoussera dans la meilleure direction. Mais il n'y a aucune différence entre le pousser dans la direction bas-gauche, bas tout court et bas-droite. Si l'on décide de redéfinir la distance minimale seulement si l'on a trouvé une distance strictement plus court, alors c'est la première calculée (bas-gauche) qui sera prise en compte. Et si on laisse l'inégalité non stricte, alors c'est la dernière calculée (bas-droite) qui sera prise en compte. Au final, dans les deux cas, si le joueur monte alors qu'un obstacle l'en empêche, alors il observera son personnage se déplacer latéralement (voir **Fig. 5**).

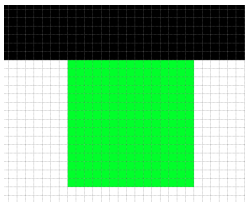

Fig. 3

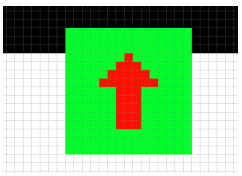

Fig. 4

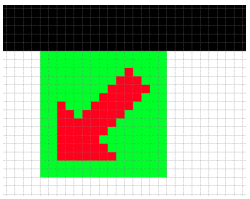

Fig. 5

Pour y remédier, on pourrait d'abord penser à supprimer les directions diagonales de l'algorithme mais ce serait retirer un bug pour en rajouter un autre : Si le joueur se trouve en **Fig. 6** et bouge en diagonale direction haut-gauche (**Fig. 7**), alors le jeu, incapable de le renvoyer d'où il vient sans diagonales, le repoussera comme indiqué sur **Fig. 8**.

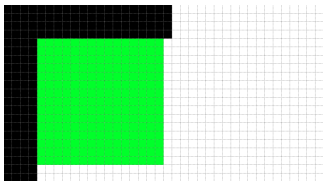

Fig. 6

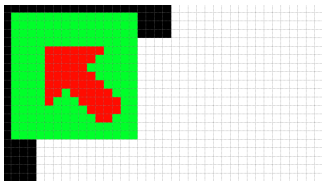

Fig. 7

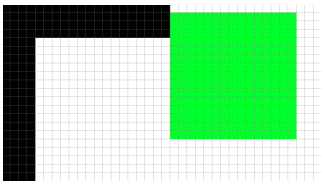

Fig. 8

On a alors simplement redéfini la distance minimale uniquement si une distance d est strictement plus courte, ou autant bonne mais non diagonale.

Il s'agit maintenant de choisir entre les deux solutions à notre disposition. Une analyse de complexité ne semble pas pertinente car, en supposant que playerIsColliding et playerCollidesAt soient $O(1)$ ce qui n'est pas le cas mais qui ne change rien, les deux solutions sont linéaires ($O(x)$ avec x la vitesse du joueur). Toutefois, même si la solution deux multiplie le nombre d'itérations par 8 (= nombre de directions), elle ne se produit que lorsque le joueur est en collision tandis que la solution 1 se trouve dans le pire cas à chaque frame. C'est pourquoi nous avons choisi la deuxième solution. Il y a possiblement des bugs que possède la première solution mais ne l'ayant pas implémentée nous n'avons pas eu l'occasion de les rencontrer.