Zeste de savoir

# Artillerie et physique dans un jeu vidéo

12 août 2019

# Table des matières



Créer des armes pour un jeu vidéo est une entreprise complexe. Les armes doivent être efficaces, agréables à jouer et suffisamment crédibles, sans pour autant rendre leur coût en performances trop élevé. La plupart des jeux vidéo prennent donc des raccourcis :

— les balles traversent souvent un niveau en une seule *frame* [1], permettant de simuler le tir en un seul test de collision ;
— les dégâts dits *de zone* se contentent de causer des dégâts dans un rayon bien défini ;
— la localisation des dégâts sur la victime se borne habituellement à des zones *bonus* qui causent plus de dégâts.

Est-il possible de faire quelque chose de plus réaliste avec les contraintes d'un jeu vidéo ? C'est le pari que nous avons fait ! Voyons ensemble quelques techniques pour contourner les limites des moteurs physiques.

## 1. Un peu de contexte

Cet article est rédigé par l'équipe du jeu vidéo Helium Rain et a initialement été publié en anglais sur notre Devblog. Notre jeu est une *space sim* qui inclut du combat dans l'espace, avec un objectif de réalisme. Nous avions besoin d'une arme antiaérienne [2] efficace pour défendre les vaisseaux majeurs des attaques de chasseurs, ce qui nous a conduit à développer un canon antiaérien en nous inspirant des technologies modernes.

---

Élément externe (Video) —
Consultez cet élément à l'adresse `https://www.youtube.com/embed/xY4crKSvLTE?feature=oembed`.

---

## 2. L'artillerie antiaérienne à l'époque des guerres mondiales

Il y a historiquement deux options pour se débarrasser de chasseurs : utiliser des chasseurs en riposte, ou embarquer des canons antiaériens sur les cibles à protéger. La seconde option est celle qui nous intéresse ici. Ces canons répondent à des contraintes lourdes : il faut toucher une cible mouvante de quelques mètres de large qui évolue à plusieurs kilomètres d'altitude avec un obus qui met dix secondes à toucher sa cible, et une technologie de visée qui pendant les deux guerres mondiales se bornait essentiellement à l'estimation humaine.

Au cours de la Première Guerre mondiale, on comprend donc bien vite que les simples obus ne sont pas une solution et on développe des obus qui explosent près de la cible, projetant des fragments métalliques dans toutes les directions. A l'époque, il n'y a que deux façons de déterminer l'instant de l'explosion : soit un seuil de temps, soit un seuil d'altitude. Les deux méthodes exigent un réglage avant le tir.

Tout change avec la Seconde Guerre mondiale qui voit l'introduction des **fusées de proximité**, des obus explosant quand ils détectent une cible près d'eux. Ces projectiles détectent les objets métalliques et rendent les batteries antiaériennes extrêmement efficaces ; à l'époque, les Alliés prennent toutes les dispositions nécessaires pour que cette technologie reste secrète, quitte à limiter son déploiement . C'est cette technologie que nous simulons dans la vidéo ci-dessus - comme vous pouvez le voir, son efficacité est certaine.


FIGURE 2. – Détonateur de proximité des années 50 (Wikimedia)

Contrairement à l'essentiel des jeux, nous avons décidé de simuler physiquement ces obus d'artillerie. La première étape est de recréer une fusée de proximité. Notre version est perfectionnée : l'obus explose quand il entre dans un **rayon de détonation**, mais il peut aussi exploser quand il détecte qu'il a atteint sa **distance minimale à la cible**.

On met donc en place une **distance d'activation** en dessous de laquelle l'obus est armé. Une fois armé, si l'obus détecte que sa distance à la cible augmente, il explose. Nos obus ont une distance d'activation de **50m** et une **distance de détonation** de **3m**.

## 3. 16.7ms chrono

16.7 millisecondes, c'est le temps qui se déroule entre deux images (ou *frames*) d'un jeu vidéo quand ce dernier a un débit d'image de 60fps, la valeur habituellement visée par les développeurs. C'est un intervalle de temps très court pendant lequel on doit mettre à jour toute la logique du jeu, transmettre une scène au moteur de rendu 3D, effectuer un rendu et l'afficher. Le budget de temps pour les projectiles, lui, est de l'ordre d'une seule milliseconde : pas question de passer toute une *frame* à calculer des positions de projectiles !

Et pourtant, 16.7 millisecondes c'est énorme : pendant ce temps-là, un obus a le temps de parcourir près de *quinze mètres*, la taille d'un petit vaisseau. Chaque mise à jour du moteur physique va déplacer un obus de cette distance. On comprend mieux pourquoi les balles de jeux vidéo ne sont jamais simulées physiquement !

L'implémentation de notre mécanisme est donc très difficile. La simulation de la physique est faite pas à pas, chaque pas de simulation correspondant au calcul de tout ce qui s'est passé dans le monde du jeu entre le temps présent et le temps de la prochaine *frame*. Comme nos obus sont très rapides (près de 800 m/s), on ne peut pas se contenter de vérifier la proximité à la fin de chaque pas, puisqu'on pourrait traverser une cible pendant ce temps !


FIGURE 3. – Trajectoire d'un obus vis-à-vis d'une cible

C'est ce qu'on appelle **l'effet tunnel** dans les jeux vidéo. Cette anomalie peut permettre à des balles de traverser des murs, par exemple. Les moteurs physiques pour le jeu vidéo ne permettent simplement pas de simuler de telles vitesses.

Pour corriger le problème, il nous faut calculer la distance minimale entre la cible et la trajectoire de l'obus, ainsi que le point associé sur la trajectoire. Si ce point n'est pas sur le segment (S0S1) correspondant à la *frame* courante , cette distance minimale n'est pas utilisable.


FIGURE 3. – Différentes positions de l'obus pendant un tick

Si la distance minimale est inférieure à la distance de détonation, l'obus explose. La position initiale de l'explosion n'est en revanche *pas* le point de distance minimale D, car le projectile doit toujours exploser à la distance de détonation et jamais en dessous. En effet, ce point de distance minimale peut parfaitement être situé à l'intérieur du vaisseau - un cas typique d'effet tunnel. Une telle explosion causerait des dégâts irréalistes.

Pour éviter cela, on calcule la position où la distance à la cible est égale à la distance de détonation, pour exploser à cet endroit précis. Une conséquence amusante de cet ajustement est qu'il faut parfois faire revenir un obus en arrière pour exploser.

FIGURE 3. – Explosion à la position précédente

Si on n'est pas encore à la distance de détonation, mais déjà en dessous de la distance d'activation, l'obus est armé. Dans ce pas de simulation ou dans les pas ultérieurs, si l'obus a été armé et que la distance augmente, l'obus explose.


FIGURE 3. – Explosion au plus près

## 4. Pluie de métal

Dans la plupart des jeux vidéo, les obus explosent avec des dégâts dans un rayon fixe et tous les ennemis proches sont touchés. Habituellement, ils reçoivent plus de dégâts s'ils sont proches du centre. Dans le monde réel, l'explosion du projectile en elle-même ne cause pas de dégâts, c'est le nuage de fragments métalliques qu'elle projette qui cause le plus de dégâts. Plus une cible est proche du centre de l'explosion, plus la *probabilité* d'être touché est grande. Notre simulation reproduit ce comportement.

Dans Helium Rain, les vaisseaux n'ont pas de points de vie. Ils sont constitués de dizaines de composants individuels, chaque composant ayant son modèle de collision. Les obus simulent des centaines de fragments touchant les vaisseaux alentour, pour rendre les effets du tir aussi réalistes que possible. Une explosion sur le côté gauche d'un vaisseau endommagera donc des composants sur sa gauche, sans affecter son côté droit. La méthode la plus simple pour implémenter cette simulation serait de créer autant de *traces* [3] physiques qu'il y a de projectiles, mais elle serait bien trop coûteuse en performances, et inefficace, car très peu de rayons toucheraient leur cible.


FIGURE 4. – Tracé de rayon pour des fragments

Pour augmenter la densité du nuage de fragments tout en diminuant la charge du processeur, il devient nécessaire d'optimiser. On sélectionne d'abord toutes les cibles dans un rayon raisonnable (*en rouge sur les images suivantes*), et pour chacune d'entre elles on calcule l'intersection entre leur *sphère circonscrite* [4] (*en magenta*) et une sphère centrée sur l'obus (*en jaune*) qui représente la distance à la cible. Le ratio de surface entre la sphère jaune et l'intersection des deux sphères correspond au ratio entre le nombre de fragments que devrait émettre la munition et le nombre de fragments que l'on doit vraiment simuler (*en bleu*). Un dessin pour comprendre...


FIGURE 4. – Tracé de rayon optimisé pour des fragments

Il suffit ensuite de simuler les fragments utiles et de propager les dégâts correspondants aux composants touchés. Voici le résultat en 3D, dans le jeu, avec la visualisation correspondant à l'image précédente.


FIGURE 4. – Tracé de rayon pour des fragments, vu en 3D

## 5. Pour aller plus loin...

Ces obus sont encore très dangereux à utiliser, car ils ne font pas la différence entre alliés et ennemis, puisqu'ils se comportent comme des détecteurs de métaux. Plus simplement, tirer sur un ennemi proche peut causer des dégâts à son propre vaisseau. Pour pallier ce problème, les obus sont équipés d'un *timer*, qui ne permet à l'obus de s'armer que quand il est à une certaine distance du canon. Ce timer permet aussi de désactiver un obus qui aurait dépassé sa cible.

La vidéo au début de l'article montre l'efficacité de ces obus. Bien sûr, on pourrait encore développer le sujet en parlant des tourelles qui tirent ces projectiles, puisqu'elles doivent anticiper les mouvements des adversaires et compenser ceux du joueur ; ou encore de l'intelligence artificielle qui régit les pilotes des vaisseaux pour leur permettre d'échapper à ces obus... Ce qu'il faut retenir, c'est qu'il est possible de simuler très précisément des mécanismes réels dans un jeu vidéo.

En bonus, voici un petit extrait du code du jeu, issu du calcul du nombre de fragments à simuler.

```
1     float ApparentRadius =
          FMath::Sqrt(FMath::Square(CandidateDistance) +
          FMath::Square(CandidateSize));
2     float Angle = FMath::Acos(CandidateDistance/ApparentRadius);
3     float ExposedSurface = 2 * PI * ApparentRadius *
          (ApparentRadius - CandidateDistance);
4     float TotalSurface = 4 * PI * FMath::Square(ApparentRadius);
5     float ExposedSurfaceRatio = ExposedSurface / TotalSurface;
6
```

```
7     int FragmentCount =
          ShellDescription->GunCharacteristics.AmmoFragmentCount *
          ExposedSurfaceRatio;
```

On peut noter que le niveau de mathématiques requis pour ce genre de travail est très accessible. Les notions basiques de géométrie sont suffisantes.

Merci de votre lecture !

1. Une *frame* est une image faisant partie d'une vidéo. On parle de *frames per seconds* pour désigner le débit d'images à l'écran, par exemple.
2. On parle d'*antiaérien* parce que le terme est courant, il est inutile de préciser qu'il n'y a pas d'air dans l'espace.
3. Une *trace*, dans le jeu vidéo, est un segment entre deux points qui teste l'existence d'objets physiques. Imaginez une balle dans un jeu de tir.
4. La *sphère circonscrite* ou *bounding sphere* est la plus petite sphère qui contient tout un objet, en 3D.