R5.04 - Qualité algorithmique

TD3 - Détection de collision et quadtree

2025-2026

Détecter les collisions

Dans le développement de jeux vidéo, il est essentiel de mettre en place des stratégies efficaces pour **détecter** les collisions entre les entités du jeu, afin de ne pas dégrader l'expérience utilisateurs.

Pour cela, une méthode consiste à découper les entités (personnages, objets, obstacles, ...) en sprit, qui correspondent à des formes géométriques. Cette technique se nomme **Axis-aligned bounding box**. Ici nous considérerons que chaque sprit correspond à un rectangle dont les côtés sont parallèles aux axes d'un plan cartésien. On notera également n le nombre total de sprit de la fenêtre. Pour simplifier, on supposera que chaque sprit est associé à une entité distinct.

1. Proposer une solution simple permettant de trouver toutes les collisions possibles à un instant t. calculer tte les posiblitées entre les sprit pour savoir s'ils se touchent
2. Quelle est la complexité de votre solution ? O(N&)
3. Proposez une implémentation en C de votre solution. Pour cela, il faut proposer :
    1. une structure afin de représenter un sprit
    2. une fonction qui prendra en paramètre un tableau de sprit de taille n et qui renverra le nombre de collisions total.

Nous allons maintenant voir comment optimiser le calcul des collisions. En effet, il n'est pas très utile de tester une collision entre deux objets très éloignés. Pour palier ce problème, une solution consiste à diviser la fenêtre en zones plus petites. Pour faire simple, on supposera qu'un sprit peut soit être contenu dans une seule zone, soit être à l'intersection de plusieurs zones voisines.

Une structure de données appropriée pour cela est le **quadtree**.

Présentation du quadtree

L'idée est de fixer le nombre k de sprits pouvant être contenu dans une zone. Lorsque le nombre de sprits dépasse k, on va subdiviser la zone en quatre zones de taille égale : nord-est (NE), nord-ouest (NO), sud-est (SE) et sud-ouest (SO). Chaque sprit est donc associé à sa nouvelle zone, qui aura une capacité de k.

On applique cette procédure pour chaque insertion d'un sprit dans le quadtree. Les feuilles de notre quadtree stockent donc des sprits, les noeuds internes quand à eux se contentent de pointer vers d'autres noeuds.

1. Proposez une structure pour représenter un noeud du quadtree. On nommera cette structure `noeudQT`. Un noeud doit contenir :
   — Les quatre pointeurs vers les noeuds "enfant"
   — les coordonnées de la zone qu'il représente (on utilisera des `float` pour les coordonnées)
2. Proposez maintenant une structure pour représenter un quadtree, qui contiendra la racine de l'arbre ainsi que le nombre k de sprites au maximum stockés par un noeud. Cette structure sera notée `quadtree`.

On rappelle que nous avons fait l'hypothèse (simplificatrice) qu'un sprit est contenu dans une seule zone. Les collisions ne peuvent donc avoir lieu qu'entre deux sprites d'une même zone.

3. En quoi cela est intéressant par rapport à la première solution qui comparer toutes les collisions possibles ?

Cependant, plusieurs problématiques se posent maintenant :

— il faut stocker en mémoire le quadtree que nous venons de construire.
— il faut maintenir cette structure de données
— il faut pouvoir accéder rapidement à toutes les zones du quatree contenant des sprites, autrement dit les feuilles de l'arbre.

Bref, nous devons faire des compromis !

Nous allons tout d'abord regarder le coût de maintenance et d'interrogation. Nous nous intéresserons dans un deuxième temps à la consommation mémoire de la structure.

-créer l'arbre de subdiv
-insérer les bullets
-vérifier si une zone est trop remplie
-diviser les zones remplie
-Vérifier les collisions

Utiliser un quadtree

Nous nous intéresserons surtout à la recherche des feuilles dans le quadtree. L'idée est de parcourir l'arbre récursivement depuis la racine et de s'arrêter lorsqu'une feuille est trouvée. On compte le nombre de collisions dans la zone et on renvoie cette valeur. On récolte ainsi les collisions de chaque feuille.

1. Supposons que l'arbre est bien équilibré, quel est sa hauteur ? Et son nombre de feuilles au maximum ?
2. Proposez une fonction en C que l'on nommera `int recolte(struct quadtree * qt)` qui permet de renvoyer le nombre de collisions total.
3. Déterminez asymptotiquement le temps nécessaire pour compter toutes les collisions.
4. Imaginez un cas où l'arbre n'est pas bien équilibré. Que dire des performances dans ce cas ?

Et en mémoire ca donne quoi ?

Nous nous sommes convaincus que l'utilisation d'un quadtree possédait un réel intérêt pour notre problème de calcul de collisions. Cependant quel est le coût en mémoire d'une telle structure de données ?

1. Pour commencer, nous allons nous placer du point de vue du théoricien et supposer que le coût d'un noeud de l'arbre est constant par rapport aux nombres de sprites total. Quel est le nombre de noeuds d'un arbre bien équilibré ? Et dans le pire cas ? On pourra faire des hypothèses simplificatrices, par exemple que n (le nombre de sprites total) est un multiple de k (le nombre maximum de sprite par noeud).
2. Cependant, en pratique, quelle est la taille en mémoire d'un noeud ? On s'intéressera uniquement au coût des différentes champs de la structure, sans prendre en compte les éventuels métadonnées de la structure.
3. En déduire le coût total d'un quadtree.