Algorithmes PAF

Romain Darous, Louis Martinez, Lucas Thomasset, Arthur Toulouse

July 4, 2022

1 Arbres quaternaires

Pour implémenter le comportement des boids, ceux-ci ont besoin de connaître leurs voisins. Cependant cela nécessite à priori d'itérer sur tous les boids présents et ne retenir que ceux qui sont en dessous d'une distance seuil. Les arbres quaternaires permettent de réduire le nombre d'itérations pour trouver les voisins d'un boid. (On passe d'une complexité moyenne en O(n^2) à une complexité en O(n.log(n)))

Un arbre quaternaire **A** est défini récursivement. Chaque noeud stocke les données suivantes :

- Frontière **f** (rectangulaire) du plan à laquelle il est associé. Elle est définie par le 4-uplet (x, y, w, h) où (x,y) sont les coordonnées du centre du rectangle, (w,h) sa largeur et sa hauteur.
- Référence vers ses 4 fils **(nw, ne, sw, se)** s'ils sont définis
- Liste **L** des boids qui se trouvent dans la région associée au noeud
- Capacité **c** : nombre maximum d'boids au-delà duquel on définit les 4 fils du noeud (subdivision de la région associée en 4 sous-régions)

A chaque actualisation, les n boids (et leur position) sont stockés dans L. L'arbre A est recréé à chaque fois et on y insère chaque boid. Il faut itérer deux fois sur l'intégralité des boids : une première fois pour tous les insérer dans l'arbre et la deuxième pour identifier le voisinnage de chaque boid.

Algorihtme 1 Actualisation à chaque raffraichissement

```
pour tout boids b faire
    A.insérer(b)
fin pour

pour tout boids b faire
    définir une zone de recherche r centrée sur b
    A.requête(r,null)
fin pour
```

Algorihtme 2 insérer(b)

```
divisé ← faux
si f ne contient pas b alors
    renvoyer faux
fin si

si L.taille ≤ c alors
    L.ajouter(b)
    renvoyer vrai
sinon si A.divisé = faux alors
    subdiviser()
fin si

renvoyer ne.insérer(b) ou nw.insérer(b) ou se.insérer(b) ou sw.insérer(b)
```

Algorihtme 3 subdiviser()

$ne \leftarrow nouvelarbre((x + w/2, y - w/2, w/2, h/2), c)$
$nw \leftarrow nouvelarbre((x - w/2, y - w/2, w/2, h/2), c)$
$se \leftarrow nouvelarbre((x + w/2, y + w/2, w/2, h/2), c)$
$sw \leftarrow nouvelarbre((x - w/2, y + w/2, w/2, h/2), c)$
divisé ← vrai

Algorihtme 4 requête(r, T)

si T est nulle **alors**
  initialiser T comme une liste de boids vide
fin si

si f n'intersecte pas r **alors**
  renvoyer T
fin si

pour tout boids ∈ L **faire**
  ne.requête(r,t)
  nw.requête(r,t)
  se.requête(r,t)
  sw.requête(r,t)
fin pour

renvoyer T

2 Forces subies par les boids

On suppose que chaque boid connaît ses voisins grâce à la méthode décrite plus haut. Pour un boid, on définit donc la liste **V** de ses voisins. Les fonctions suivantes sont définies dans class boid. **self** désigne l'instance de boid dans laquelle les fonctions suivantes sont définies.

Algorihtme 5 Force de cohésion

Prérequis: Liste V des voisins du boid considéré
  position_désirée ← Vecteur(0,0)
  force ← Vecteur(0,0)
  diff_angle ← 0

  pour tout boids b ∈ V **faire**
    écart_position ← b.position - self.position
    diff_angle ← angle entre position_désirée et self.vitesse
    si diff_angle ≤ self.angle_de_vue **alors**
      position_désirée ← position_désirée + b.position
    fin si
  fin pour

  si V.taille ≥ 0 **alors**
    position_désirée ← position_désirée / V.taille
    position_désirée ← position_désirée - self.position
    position_désirée.norme ← self.vitesse_max
    force ← position_désirée - self.vitesse
    force.norme_maximale ← self.force_max
  fin si

  renvoyer force

Algorihtme 6 Force de séparation

Prérequis: Liste V des voisins du boid considéré
```
position_désirée ← Vecteur(0,0)
force ← Vecteur(0,0)
diff_angle ← 0
diff ← 0

pour tout boids b ∈ V faire
   écart_position ← b.position - self.position
   diff_angle ← angle entre position_désirée et self.vitesse
   si diff_angle ≤ self.angle_de_vue alors
      diff ← self.position - b.position
      diff.normaliser()
      position_désirée ← position_désirée + diff
   fin si
fin pour

si V.taille ≥ 0 alors
   position_désirée ← position_désirée / V.taille
   position_désirée.norme ← self.vitesse_max
   force ← position_désirée - self.vitesse
   force.norme_maximale ← self.force_max
fin si

renvoyer force
```

Algorihtme 7 Force d'alignement

Prérequis: Liste V des voisins du boid considéré
```
vitesse_désirée ← Vecteur(0,0)
force ← Vecteur(0,0)
diff_angle ← 0

pour tout boids b ∈ V faire
   écart_position ← b.position - self.position
   diff_angle ← angle entre position_désirée et self.vitesse
   si diff_angle ≤ self.angle_de_vue alors
      vitesse_désirée ← vitesse_désirée + b.vitesse
   fin si
fin pour

si V.taille ≥ 0 alors
   vitesse_désirée ← vitesse_désirée / V.taille
   vitesse_désirée.norme ← self.vitesse_max
   force ← position_désirée - self.vitesse
   force.norme_maximale ← self.force_max
fin si

renvoyer force
```

Pour calculer la force d'évitement d'obstacles, le boid cherche la première direction dans laquelle il n'y a plus d'obstacle. Pour cela il balaye son champ de vision sur un nombre déterminé de directions (**nombre_points**). Le **facteur d'anticipation** permet de faire varier la distance à partir de laquelle un boid modifie sa trajectoire.

Algorihtme 8 Force d'évitement d'obstacle

Prérequis: Liste O des obstacles du terrain, liste V des boids, facteur_anticipation, nombre_points
vitesse_évitement_désirée ← Vecteur(0, 0)
champ_vision_orienté ← Vecteur(0, 0)
force ← Vecteur(0, 0)
obstacles_voisins ← 0

// Paramètres angulaires pas ← 1/nombre_points theta ← 0

pour tout obstacle ∈ O **faire**
  tant que d ≤ facteur_anticipation × obstacle.rayon **faire**
    theta ← $2 \times \pi \times pas + 1$
    champ_vision_orienté ← self.position + self.vitesse.rotation(theta).norme(obstacle.rayon)
    d ← distance(champe_vision_orienté, obstacle.position)
    si i ≤ 0 **alors**
      i ← -i+1
    sinon
      i ← -i
    fin si
  fin tant que
fin pour

si theta ≠ 0 **alors**
  vitesse_évitement_désirée ← vitesse_évitement_désirée + self.vitesse.rotation(theta)
  obstacles_voisins ← obstacles_voisins + 1
fin si

i ← 0
theta ← 0

si obstacles_voisins ≥ 1 **alors**
  vitesse_évitement_désirée ← vitesse_évitement_désirée / obstacles_voisins
  vitesse_évitement_désirée.norme ← self.vitesse_max
  force ← vitesse_évitement_désirée - self.vitesse
  force.norme_maximale ← self.force_max
fin si

renvoyer force

3 Détection des groupes

En entrée, on prend boids la liste de tous les boids (n = nombre de boids) et groups la liste de tous les groupes. L'idée de l'algorithme est la suivante : A chaque itération de draw(), il y a une phase d'initialisation où on ne garde qu'un boid par groupe. Ensuite, on détermine pour chaque boid la liste de ses voisins (seuil de distance et/ou d'angle de vitesse). Puis on répartit les boids non-leader dans les groupes déjà existants et on en crée de nouveau si besoin. Enfin, on traite un cas particulier du à la phase d'initiation (boid avec plusieurs voisins mais seul dans leur groupe). L'algorithme a une complexité moyenne en $O(n^2)$.

Algorihtme 9 Initialisation de groupes

Prérequis: Liste G des groupes
pour tout groupe ∈ G **faire**
  ne garder que le premier boid (leader) du groupe
  attribuer à tous les autres boids le groupe 0
fin pour

Algorihtme 10 Détermination des voisins

Prérequis: Liste V des boids
  pour tout boid b *in* V **faire**
    liste des voisins ← liste vide
    nombre total de voisins ← 0
    pour tout boid autre ∈ V **faire**
      si autre est assez loin du bord **alors**
        si distance(b, autre) ≤ seuil et b ≠ autre et angle(b.vitesse, autre.vitesse) ≤ seuil **alors**
          liste des voisins.ajouter(b)
          nombre total de voisins ← nombre total de voisins + 1
        fin si
      sinon
        si distance(b, autre) ≤ seuil et b ≠ 0 **alors**
          liste des voisins.ajouter(b)
          nombre total de voisins ← nombre total de voisins + 1
        fin si
      fin si
    fin pour
  fin pour

Algorihtme 11 Formation des groupes

Prérequis: List V des boids, liste G des groupes

```
id_group ← prochain identifiant disponible // max des id dans G + 1
pour tout boid b ∈ V faire
  si b.groupe.id = 0 alors
    si b n'a pas de voisin alors
      Créer un nouveau groupe grp dans G avec l'id id_group
      Ajouter b dans grp
      id_group ← id_group + 1
    fin si
    midID ← 0
    res ← [0] // Liste des id des groupes des voisins de b
    pour tout boid autre ∈ liste des voisins de b faire
      si autre.id n'est pas dans res alors
        Ajouter autre à res
      fin si
      midID ← min(res)
    fin pour
    si midID = 0 alors
      On crée un nouveau groupe group dans groups avec l'id id_group, et on ajoute boid dans ce groupe
      pour tout voisin de b faire
        group.ajouter(voisin)
      fin pour
    fin si
    si midID ≥ 1 alors
      finalID ← élément de res privée de 0 qui correspond au group avec le plus de boids
      On prend le groupe avec l'id finalID (déjà existant), et on y ajoute boid
      pour tout autres éléments de res privé de 0 faire
        On enlève le groupe de group
        ON prend le groupe avec l'id finalID, et on y ajoute chaque membre des groupes ayant un voisin
        de b
      fin pour
    fin si
  fin si
fin pour
```

Algorihtme 12 Cas particulier

Prérequis: Liste G des groupes

```
pour i allant de 0 à G.taille faire
  si G[i] n'a qu'un seul boid b alors
    si b a plusieurs voisins alors
      Enlever group de G et ajouter b au groupe du premier voisin qu'on a trouvé
    fin si
  fin si
fin pour
```

4 Enveloppe convexe

Algorihtme 13 Algorithme de Jarvis

Prérequis: Liste V des boids
q ← premier boid de la liste

pour tout boib b ∈ V et qui n'est pas le premier **faire**
  si b.y ≤ q.y **alors**
    q ← v
  fin si
fin pour

u ← q
p ← q

tant que p ≠ u **faire**
  q ← premier point de V

  pour tout v ∈ v après q **faire**

    si pvq tourne dans le sens contraire des aiguilles d'une montre **alors**
      q ← v
    fin si
  fin pour

  p.suivant ← q
fin tant que

5 Prédiction de trajectoire

Le but est de calculer une estimation de la trajectoire du barycentre d'un groupe de boids.

On dispose d'un objet ayant les attributs suivants :

- La position courante du barycentre du groupe **position_barycentre_courante**, Une liste de positions estimées **trajectoires_prédites**, initialisée avec des vecteurs nuls. Une liste de vitesses estimées textbfvitesses_prédites, initialisée avec des vecteurs nuls
- Trois tableaux de vecteurs : **position_pred**, **vitesse_prec**, **accélération_prec**, de même taille, initialisés avec des vecteurs nuls. Plus l'indice dans le tableau est grand, plus le vecteur associé désigne un pas de temps récent.
- Un booléen **est_plein** qui permet de vérifier si ces tableaux ont bien été remplis une fois. Vaut False initialement.
- Un booléen **erreur_trop_haute** pour décider de quand recalculer la trajectoire en fonction de l'erreur. Vaut False initialement.
- Un entier **erreur_max** pour quantifier l'erreur tolérée.
- Un entier **nombre_données_prédites** qui donne la taille de la fenêtre de prédiction.
- Un entier **positions_prédites_passées** qui permet de compter le nombre de données prédites exploitées par le processus. Elle permet de recalculer la trajectoire prédite lorsque tous les pas de temps pour lesquels la position a été prédite sont écoulés. Elle est initialisée à **nombre_données_prédites - 1**

V désigne une liste de vecteurs

Algorihtme 14 Calcul du vecteur moyen

Prérequis: Liste V de vecteurs

vecteur_moyen ← Vecteur(0, 0)

pour tout vecteur v ∈ V **faire**
  vecteur_moyen ← vecteur_moyen + v
fin pour

vecteur_moyen ← vecteur_moyen / V.taille

renvoyer vecteur_moyen

Algorihtme 15 Estimation de l'erreur de position : valeur_erreur()

erreur ← min(distance(position_barycentre_courante, position_prédite))
renvoyer erreur

L'algorithme permet de mettre à jour les valeurs des positions passées du barycentre du groupe, qui serviront à calculer les vecteurs accélération et vitesse moyen pour appliquer le modèle de la chute libre (voir la présentation du projet pour plus de détails).

Algorihtme 16 Mise à jour des valeurs précédentes : maj_valeurs_précédentes()

On translate les valeurs des tableaux position_prec, vitesse_prec et acceleration_prec d'un indice vers la gauche
Pour le plus grand indice on prend :

- Pour position_prec : position_barycentre_courante
- Pour vitesse_prec : différence entre la position au plus grand indice (qui vient d'être calculée) et celle à l'indice précédent.
- Pour acceleration_prec : différence entre la vitesse au plus grand indice et celle à l'indice précédent

Si la fonction est exécutée au moins autant de fois que la taille des trois tableaux ci-dessus, alors est_plein ← Vrai

Algorihtme 17 Calcul de la trajectoire

```
maj_valeurs_précédentes()

si est_plein alors
  si prositions_prédites_passées = nombre_données_prédites -1 ou erreur_trop_haute alors
    erreur_trop_haute ← Faux
    positions_prédites_passée ← 0
    accélération_moyenne ← vecteurMoyen(accélérations_prec)

    pour i allant de 0 à nombre_données_précédentes faire
      vitesses_prédites[i] ← vecteurMoyen(vitesse_prec) + accélération_moyenne × i
      trajectoire_prédites[i] ← 0.5 × i² × accélération_moyenne + vecteurMoyen(vitesse_prec) × i + position_barycentre_courante
    fin pour

  sinon
    position_prédites_passées ← positions_prédites_passées + 1
  fin si

  si valeur_erreur() ≥ erreur_max + 1 alors
    erreur_trop_haute ← Vrai
  fin si
fin si
```