Projet Elements Logiciels pour le Traitement de Données Massives

Crowd Behavior Analysis

Bonnet Louis, Vonin Cédric

2021

Table des matières

1 Introduction

Ces dernières années, les foules virtuelles sont devenues un élément de plus en plus commun de notre expérience cinématographique. Qu'il s'agisse d'une foule photo-réaliste de passagers numériques dans Titanic, d'une légion de soldats animés dans le Seigneur des Anneaux ou d'une armée de droïdes dans Star Wars, les foules de personnages générées par informatique ont considérablement renforcé l'impact des films qui les utilisent, permettant de visualiser des scènes qui n'étaient pas possibles il y a quelques années seulement. Les foules de personnages humains et non humains en images de synthèse aident à surmonter les coûts prohibitifs et la complexité du travail avec un grand nombre de figurants, se substituent aux cascadeurs virtuels dans les plans dangereux et s'intègrent bien aux scènes virtuelles.

Le but de notre projet est de construire et d'optimiser un algorithme de "Pedestriand Crowd Evacuation Dynamics". Pour ce faire on va dans un premier temps expliquer les grandes lignes du paradigme dans lequel on se place pour réaliser notre algorithme, nous expliciterons par la suite notre modèle et le fonctionnement de celui-ci, enfin on se penchera sur les améliorations du logiciel pour permettre un temps de calcul plus performant afin de pouvoir considérer les utilisations dans des cadres de véritables foules virtuelles.

2 Le Problème

2.1 Idée

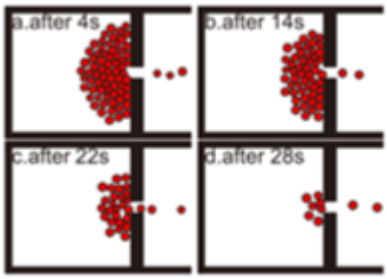
Comment mettre en place un modèle permmetant la simulation de l'évacuation de piétons confinés dans un espace comportant un obstacle ? Nous nous placerons dans un point de vue aérien de l'espace, visualisant ainsi l'espace de confinement comme un environnement 2D et les piétons comme des points de ce plan. Le but de l'algorithme est alors, à partir d'une distribution initiale des individus aléatoire, de recréer une évacuation vers un objectifs tout en évitant à chaque instant les collisions entre les individus.

L'évitement des collisions est un problème fondamental en simulation de mouvements de foules. Le problème peut être défini en général dans le contexte d'un individu mobile autonome naviguant dans un environnement comportant des obstacles et/ou d'autres entités mobiles, où le piéton utilise un cycle continu de détection et d'action. Dans chaque cycle, une action pour le piéton doit être calculée sur la base d'observations locales de l'environnement, de telle sorte que le piéton reste à l'abri des collisions avec les obstacles et les autres entités mobiles, tout en progressant vers un objectif.

Plusieurs travaux dans différents domaines ont exploré des questions liées au domaine des simulations de foule. Dans son travail de pionnier, Reynolds a décrit un modèle de comportement distribué pour simuler le mouvement global d'une volée d'oiseaux. Bouvier et Guilloteau ont utilisé l'association de systèmes de particules et de réseaux de transition pour modéliser des foules humaines dans la visualisation d'espaces urbains. Brogan et Hodgins ont simulé des comportements de groupe pour des systèmes ayant une dynamique importante. Aubel et Thalmann ont introduit des imposteurs générés dynamiquement pour rendre les humains virtuels. Tecchia et al. a proposé des méthodes basées sur l'image pour le rendu en temps réel de foules animées dans des villes virtuelles. O'Sullivan et al. a décrit la simulation de foules et de groupes avec des niveaux de détail pour la géométrie, le mouvement et le comportement. McPhail et al. a étudié les actions individuelles et collectives dans des rassemblements temporaires. Still a utilisé des automates cellulaires mobiles pour la simulation et l'analyse des évacuations de foule. Cependant, seuls quelques travaux ont tenté d'explorer des modèles de foule plus généraux, intégrant plusieurs sous-composantes telles que l'évitement des collisions, la planification des trajets, les comportements de haut niveau, l'interaction ou le rendu. Nous proposons notre idée sur la question, tout d'abord en posant les délimitations de notre problème simplifié (nous n'essaierons pas de donner d'autres instructions aux piétons que d'éviter les collisions avec les obstacles/autres entités mobiles et de rejoindre l'objectif) puis en expliquant le contenu de notre algorithme pour enfin proposer des optimisations de celui-ci en termes de performances.

2.2 Modélisation

Plaçons nous dans un cadre concret pour comprendre notre démarche. Supposons que l'on veuille évacuer un espace rempli de piétions qui comporterait une menace directe pour leur bien-être (comme un feu par exemple). On peut alors trouver un exemple de mouvement de foule exprimant non pas une evacuation optimale mais bien une évacuation en panique avec une seule sortie. Naïvement, la Figure 1 nous montre le résultat d'une telle évacuation



Pour éviter ces interactions naïves, nous nous inscrirons dans les délimitations de modèles de foules dits à "force sociale" comme définis par Dirk Helbing et Anders Johansson :

Soit un ensemble de n individus confinés dans un même espace. Pour simplifier, nous supposons que les piétons ont la forme d'un disque et se déplacent dans le plan $\mathbb{R}^2$. Chaque piéton a :

— une position actuelle p_i (le centre du disque)
— une vitesse actuelle v_i
— un rayon de collision r_i

Ces paramètres constituent l'état externe du piéton, c'est à dire que les autres individus peuvent les observer et décider en conséquent de leurs actions.

De plus, chaque robot a :

— une vitesse maximale $vmax_i$
— une vitesse préférée $vpref_i$ (la vitesse à laquelle le piéton irait si aucun autre individu ne croisait son chemin, typiquement dirigée directement vers la sortie)

Ces paramètres sont internes au piétons et ne sont donc pas observés par les autres entités mobiles du plan.

L'algorithme propose un objectif à chaque individu, il consiste pour ceux ci en le choix indépendant et simultané d'une nouvelle vitesse $vnew_i$ pour que tous les piétons soient assez éloignés afin d'éviter les collisions. De plus, cette nouvelle vitesse doit se rapprocher au plus posssible de la trajectoire prférée du piéton vers la sortie. Les piétons ne peuvent utiliser dans leur choix que la position et la vitesse actuelle des autres entités mobiles mais tous connaissent la position de l'objectif commun et suivent la même stratégie.

2.3 Modélisation

En pratique, nous avons écrit une première version de l'algorihtme sous forme simplifiée, avec une partie "back" et "front". La version "naive" est la première version à avoir été programmée. Elle est composée des programmes suivants :

- constants.py : Contenant des constantes de classes. Ces constantes permettent d'omettre l'utilisation de variables globales afin de pouvoir faciliter l'implémentation des algoirhtmes futurs
- main.py : Le fichier main permet de faire un test en "back" de l'algorithme pour les performances. Il effectue le déplacement de tous les points, un nombre de fois décidé par l'algorithme.

- main-window.py : Permet de visualiser le résultat fenêtre de l'ensemble des points simulés par le "back". Il est codé en Tkinter
- crowd-init.py : Inclut assez d'instructions pour pouvoir simuler les points de façon aléatoire qui n'entrent ni en collision avec le mur, ni entre eux. Ces points seront dans la première version des listes d'array. Dans la version optimisée, ce seront des set de tuples que nous utiliserons pour optimiser la performance des codes. Ce programme contient notamment des fonctions de :

* Vérification : "point-location-available" en 0(n) qui sera exécuté un très nombreux nombre de fois.

- "crowd-computation.py" : Contient toutes les fonctions nécessaires au déplacement des points dans l'ensemble des individus. Ce code contient l'intelligence du programme. Il contient notamment des algorithmes de :
- Vérification en 0(n) "set-already-contain-latter-array". Nous constatons dans les versions intermédiaire que la vérification des listes de "np.array" est très coûteuse. En transformant les listes d'array en liste de liste de coordonnées, nous gagnions déjà un facteur deux dans le temps de l'exécution du déplacement des points. C'est l'évolution progressive de l'algorithme de vérification qui nous a permi de gagner un facteur 25 en utilisant un set de tuples de points

La version finale se distingue de la version naïve par l'utilisation de set de listes à la place de liste d'arrays pour la représentation des points dans le mouvement de foules. Les set ont un coût de recherche de l'ordre de O(1). En enlevant l'utilisation de "array", on a pû gagner de nombreuses secondes grâce à la possibilité d'utiliser la méthode naturelle de python "in" pour vérifier la non-collision des inidivuds dans la foule (nous supposons un espace de r=1).

On comprend alors que l'utilisation des librairies Python non-natives est bien pratique pour économiser des lignes de code mais sont très fastidieuses à manipuler pour une réelle optimisation des algorithmes utilisés.

3 Modèle de Gradient

La modélisation du mouvement de foule (avec modèle de gradient) s'est fait en pratique de la façon suivante : - Le nombre d'individus est variable - Le nombre d'étapes de la simulation est variable (dans le fichier constants.py) - Chaque perosnne a personne va se déplacer de façon autorisée, dans une fenêtre rectangulaire (exemple de simulation disponible dans main-window.py). Les individus sont bien entendu capables de se déplacer , mais uniquement sur un ensemble de coordonnées etnières positives qui seront décidés en fonction de règles décrites dans le programme. - On définit des vecteurs unitaires de coordonnées élémentaires (HAUT, BAS, GAUCHE, DROIE). Pour obtenir une direction possible de déplacement, on choisit soit l'un des vecteurs unitaires dans l'ensemble, soit une combinaison de deux veccteurs distincts dans la liste.(8 directions posisbles pour une personne) - Personne ne peut entrer en collision l'un de l'autre par les règles du programme. Les coordonnées sont écrits en fonction du repère cartésien du module Tkinter (avec l'axe des abcisses conservé et l'axe des ordonnées oreinté vers le bas). - Personne ne peut quitter le terrain - Il existe un nombre de murs N dont aucun des piétons ne peut franchir - A chaque étape, les personnes se déplacent au tour par tour. Ils ne se déplaceront que dans les directions autorisés - Aussitôt un piéton se déplace, les coordonnées de celui-ci seront mis à jour dans la liste des coordonnées ("list d'array" qui deviendra un set de tuples)

Nous avions naturellement recouru à l'utilisation de variables globale pour inscrire explicitement les règles du programme. Cependant, cette pratique nous a fait perdre beaucoup de temps (à cause de "l'effet spaghetthi"). Ainsi, nous avons eu à recommencer le développement en remplaçant ces variables par des variables globales. Toutes ces variables globales sont, rappelson-le, disponibles dans le fichier constants.py.

L'initialisation des coordonnées des individus de la foule se fait de façon aléatoire. Il s'agit du point fabile du programme, il n'est pas forcément optimal en fonction du nombre de coordonnées à créer. Nous avons fait nos simulations avec un nombre de piétons égal à 50

A chaque étape, les coordonnées des piétons est mise à jour, il y a plusieurs parties dans le programme :

- La position de chaque individu sera répertoriée dans une liste (ou set) de coordonnées. Dans un premier temps nous avons choisi de créer des listes d'arrays (à 2 coordonnées) pour stocker les coordonnées de tous les individus. Cette pratique a eu pour avantage de nous fournir une panoplie plus étendu de possibilités de calcul. Cependant, elle

n'est pas optimale car nous avions régulièrement eu besoin de vérifier le contenu de la liste. Un array est un type ambigû. Ce type oblige à itérer une liste entière pour vérifier l'existence d'une clé dans une liste non-ordonnée (et non ordonnable). Cela crée un coût de recherche en O(n), c'était excessif. En pratique, nous devions mettre 50 secondes pour lancer 1000 déplacemnet des piétons sur la partie back de notre programme. Les performances sur la partie front étaient bien médiocres.

En se contenant de changer le format en set de tuples, nous avons réussi à généraliser la recherche en O(1). nous avons donc réussi à gagner énroméméent de secondes. Le programme est passé d'une exécution en 50 secondes à une exécution en environ 3 secondes. On notera également que le fait de convertir les list d'array en liste de liste pour faire des vérifications a augmenté par deux la performance de l'algorithme. Ainsi, nous avons enlevé un maximum de référence à numpy tout en conservant certaines d'entre elles, pour faciliter certains calculs plus long (calcul du gradient).

Méthode du gradient en pratique dans le programme : - Nous avons crée une variable qui indique la porte de la sortie. En front, elle le dessinne. En back, nous prenons en compte la sortie afin d'attribuer à chaque utilisateur i à l'étape j quelle est le point A(i,j) préféré qu'il va choisir pour s'enfuir de la pièce. - Un utilisateur va chercher à se rapprocher du plus que possible de la sortie selon une fonction distance ua carré. Cette norme nous est pratique car elle est différentiable. Il va alors tenter du mieux qu'il peut de diminuer sa distance a la sortie la plus proche. - L'utilisateur ne souhaite pas calculer le résultat explicite mais chercher la direction éléemntaire la plus proche du gradient (avec des calculs d'angles etc.) - L'utilisateur ne va pas nécessairement suivre le chemin unitaire le plus proche du gradient (pas = 1), mais comparer cette valuers à autres directions possibles. Si cela est possible, il va alors choisir de préférence la direction que le gradient lui conseille.

La majorité des algoirthmes sont assez simples en pratiques. Etant donné que le gradient de chaque individu à l'étape i ne dépend pas des coordonnées des autres indiviuds , il est possible de paralléliser le calcul de l'ensemble des individu en créant des calculs à (nombre d'individus) blocs et un nombre de thread que l'on peut optimiser.