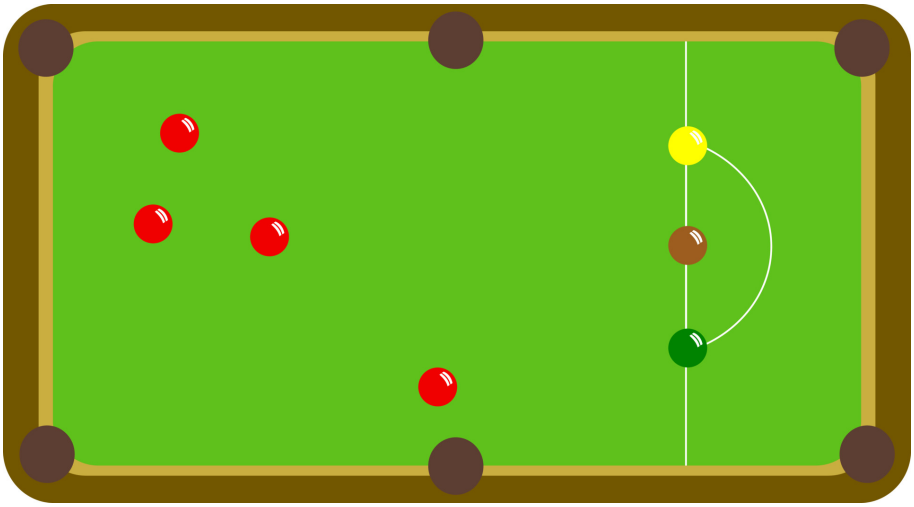
SAÉ 1.02 - Comparaison d'approches algorithmiques

Optimisation d'algorithmes pour une simulation de snooker



Sports Vectors Images by Vecteezy

Descriptif détaillé de la SAÉ

En quoi consiste cette SAÉ ?

En partant d'un besoin exprimé par un client, il faut réaliser une implémentation, comparer plusieurs approches pour la résolution d'un problème et effectuer des mesures de performance simples. Cette SAÉ permet une première réflexion autour des stratégies algorithmiques pour résoudre un même problème.

Quelles sont les productions de cette SAÉ ?

— Code de l'application.
— Présentation du problème et de la comparaison des différentes approches.

Quelles sont les compétences développées ?

Appréhender et construire des algorithmes :
— AC 1 Analyser un problème avec méthode (découpage en éléments algorithmiques simples, structure de données, ...)
— AC 2 Comparer des algorithmes pour des problèmes classiques (tris simples, recherche, ...)

Projet de 12h, travail **en binôme**.

Contexte

Vous êtes embauché en tant que stagiaire chez **Rockstar Games** pour travailler sur le prochain GTA qui doit sortir dans environ 18 mois. Le chef de projet du jeu souhaite y introduire une simulation de snooker. En effet, une partie du scénario se déroulera dans un bar où le joueur devra défier d'autres clients au snooker. Vous êtes donc responsable du développement de cette partie du jeu et donc des algorithmes de simulation de rebond des billes entre elles.

Vous avez développé une interface graphique pour vérifier le bon fonctionnement de vos algorithmes. Rapidement, vous vous rendrez compte que votre jeu utilise beaucoup trop de ressources et qu'un processeur Core i9 et une carte graphique RTX 3080 sont le strict minimum pour lancer votre chef-d'oeuvre !

Vous ne voulez pas vous l'avouer mais vous êtes bien obligé de le constater : vous codez comme un pied ! Il va donc falloir passer par une phase d'optimisation de vos algos : optimisation algorithmique mais également mathématique...

Description de l'application

L'application (fichier `pool.py`) consiste en une interface graphique rectangulaire modélisant une table de snooker avec les billes correctement réparties au lancement du programme.

La table est composée d'une bille blanche que vous allez taper en direction d'autres billes pour les faire rebondir. Les règles du snooker ne sont pas respectées et ce n'est absolument pas le but de ce projet.

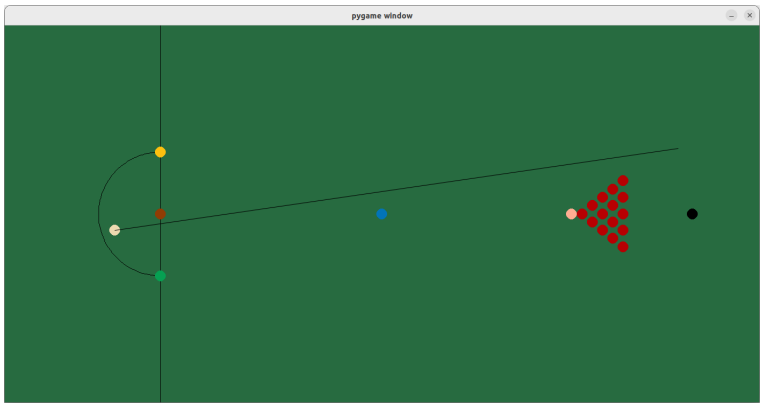


FIGURE 1 – Capture d'écran de la simulation de snooker.

Lors du lancement de l'application, vous pouvez viser depuis la bille blanche avec votre souris. Une ligne affiche la direction que prendra la bille blanche si vous cliquez pour lancer la simulation. Vous pouvez mettre l'application en pause en appuyant sur espace.

Les billes sont décrites par une position, une direction de déplacement qui est nulle lorsque la bille ne bouge pas et une couleur.

Les paramètres, tels que la couleur des billes, leur taille, le coefficient de ralentissement, etc. se trouvent dans le fichier `constants.py`.

Le fonctionnement général de cette application est très simple et est décrit grâce aux algos simplifiés 1, 2, 3 et 4.

Algorithm 1: Programme principal

```
1 Création de la zone de jeu
2 balls ← création des billes
3 while Fenêtre non fermée do
4     if clic gauche then
5         Mettre à jour la direction de déplacement de la bille blanche
6     if une bille au moins bouge then
7         collisions ← Calculer les collisions entre p et persons
8         Traiter collisions
9         Mettre à jour la position et la direction de déplacement de toutes les billes.
```

Algorithm 2: Calculer les collisions entre `p` et la liste `balls`

```
1 collisions ← []
2 for toutes les billes q de balls do
3     if intersection entre les cercles représentants p et q then
4         collisions ← collisions + (p, q)
5 return collisions
```

Algorithm 3: Traiter les collisions

```
1 for toutes les collisions c de collisions do
2     a ← c[0]
3     b ← c[1]
4     Mettre à jour les attributs dx et dy de a et b
```

Lorsque vous lancerez la bille blanche, vous constaterez que la simulation semble lente et saccadée. En regardant de plus près, on peut même s'apercevoir que les collisions ne sont pas parfaites.

Pour évaluer la performance de vos algorithmes, un programme de benchmark est fourni. Ce programme (fichier `bench.py`) permet d'évaluer réellement les performances de la simulation en ne lançant que les calculs nécessaires et sans interface graphique. En effet, lors de

Algorithm 4: Mettre à jour une bille

1 Mettre à jour la position de la bille
2 Mettre à jour la direction de déplacement de la bille
3 Gestion du rebond de la bille sur les bords

l'exécution de la version graphique, le programme doit calculer l'image à afficher, et ceci, de nombreuses fois. Ce calcul est très gourmand en ressource et ralentit donc la simulation complète.

Le fichier de benchmark reprend donc exactement la même chose que la simulation graphique mais sans affichage et pour un tir prédéfini. Ce programme affiche le temps de calcul nécessaire au traitement du tir.

Pour conclure, le programme graphique vous permettra de vérifier le bon fonctionnement de vos algos et le programme benchmark vous permettra d'évaluer les performances.

TODO

Le but du projet

Ce qui nous intéresse ici n'est pas la bonne gestion de la simulation en intégrant les règles du snooker mais les performances des algorithmes. Plus vos algorithmes seront efficaces, plus des ressources nécessaires au vrai jeu seront disponibles!

Vous allez donc optimiser vos algorithmes de plusieurs manières.

Ce que vous allez et rendre!

Le sujet est découpé en plusieurs étapes qui, toutes combinées, offriront les meilleures performances à vos algorithmes. Ces différentes étapes vous demanderont de réaliser des optimisations en modifiant l'existant (le code de départ ou le code de l'une des étapes précédentes). Toutefois, toutes ces étapes sont **indépendantes**. Si vous n'arrivez pas à répondre à l'optimisation demandée, vous pourrez passer à la suivante sans résoudre la précédente.

Lors de ce projet, vous devrez produire, pour chaque étape, du code commenté, lisible, etc. mais également un compte-rendu (PDF) dans lequel vous expliquerez, en quelques lignes, comment vous avez modifié le code existant (code de départ ou de l'une des étapes précédentes) pour optimiser les algorithmes.

Lorsque vous aurez terminé toutes les optimisations (ou au moins celles que vous aurez réussi à faire), vous lancerez le benchmark avec les différentes optimisations pour faire une

comparaison (encore un PDF !).

Au final, vous devrez produire :

— **Pour chaque étape, un code source** respectant toutes les normes que vous avez vues en cours (nommage des variables, indentation, commentaires, etc.).
— **Pour chaque étape, un document PDF** contenant une explication de quelques lignes des modifications faites à l'existant.
— **Un dernier pdf** comparant les performances des différentes optimisations.

Sur moodle, vous rendrez une archive contenant des répertoires pour chaque étape de l'optimisation. Chaque répertoire sera nommé **etape_i** avec i le numéro de l'étape. Chaque répertoire contiendra tous les fichiers sources (n'oubliez pas d'y mettre également `bench.py`).

Étape 1 : Optimisation algorithmique 1

Vous pourrez constater que la simulation est saccadée. Cela est dû au fait que, dès que la bille blanche est lancée, un test de collision est réalisé entre toutes les billes.

Or, il ne peut y avoir de collision entre des billes qui ne bougent pas. Modifiez le code existant pour faire en sorte que le test de collision ne soit effectué que lorsque la bille parcourue est en mouvement.

Une fois la modification effectuée, vérifiez qu'il y a bien une amélioration de la performance en comparant les résultats du programme benchmark avant et après optimisation.

(code) - Modifiez le moteur de la simulation pour que le test de collision ne soit réalisé que lorsque la bille parcourue est en mouvement.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte.

⚠N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 2 : Optimisation mathématique

Vous allez maintenant optimiser la simulation d'un point de vue mathématique. Votre simulation passe le principal de son temps à tester des intersections entre des cercles. Cette partie du code doit donc être la plus efficace possible. Ce code se trouve dans la fonction `circleCollision` du fichier `engine.py`. Elle prend le centre de deux cercles en paramètre. Les rayons sont connus (`BALL_RADIUS`) et présents dans le fichier `constants.py`.

Appelons `c1` et `c2` les deux cercles passés en paramètre. L'idée principale pour vérifier si `c1` intersecte `c2` (ou inversement) est de tester si un point `p` contenu dans `c1` se trouve également dans `c2` (voir figures 2 et 3). Si un tel point peut être trouvé alors les deux cercles s'intersectent.

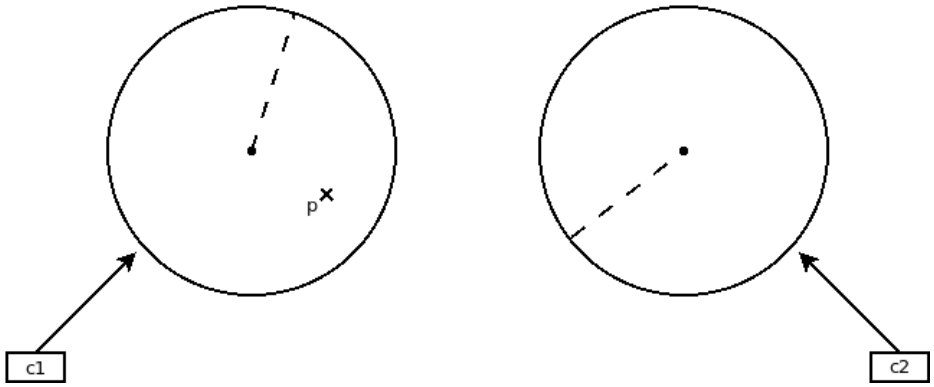


FIGURE 2 – Pas d'intersection entre `c1` et `c2`. Aucun point `p` appartenant à `c1` et `c2` ne peut être trouvé.

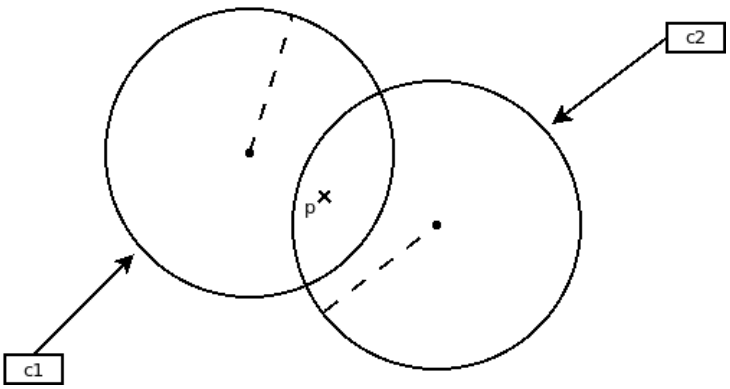


FIGURE 3 – Intersection entre `c1` et `c2`. On a trouvé au moins un point `p` de `c1` appartenant également à `c2`.

Le but de cette fonction (`circleCollision` du fichier `engine.py`) est donc de trouver un point `p` de `c1` appartenant à `c2`. Si un tel point est trouvé alors les cercles s'intersectent, sinon il n'y a pas d'intersection. Il est évident que les points `p` testés ne doivent pas être pris au hasard. Les points `p` testés seront donc pris sur le bord du cercle `c1` à intervalle régulier et

seront assez rapprochés. Ainsi, si un point du bord du cercle appartient également à l'autre cercle alors nous aurons prouvé leur intersection.

Pour vérifier si le point `p` appartient au cercle `c2`, il suffit de calculer la distance entre `p` et le centre du cercle `c2`. Si cette distance est plus petite que le rayon du cercle alors le point appartient bien au cercle.

L'algorithme de test d'intersection est décrit sur l'algo 5.

Algorithm 5: Tester l'intersection de `c1` et `c2`

```
1 # c1 désigne le centre du cercle c1
2 # c2 désigne le centre du cercle c2
3 for angle ← 0 à 2 × π par pas de 0.01 do
4     p_x ← c1_x + cosinus(angle) * rayon de c1
5     p_y ← c1_y + sinus(angle) * rayon de c1
6     if distance(c2,p) < rayon de c2 then
7         return vrai
8 return faux
```

C'est cet algorithme que vous allez devoir optimiser! Cherchez une façon plus optimisée de réaliser ce test.

Une fois la modification effectuée, vérifiez qu'il y a bien une amélioration de la performance en comparant les résultats du programme benchmark avant et après optimisation.

(code) - L'algorithme de test d'intersection de cercles décrit précédemment peut être qualifié de totalement éclaté et loin d'être efficace! Trouvez un meilleur algorithme et remplacez ainsi la fonction toute pourrie!

Rappel : on essaie de supprimer un maximum de calculs et notamment les calculs lourds pour la machine comme les racines carrées. Si vous pouvez, évitez de les utiliser.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte.

⚠ N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 3 : Optimisation algorithmique 2

Si vous avez réussi à optimiser l'algorithme de test d'intersection des cercles à l'étape précédente, vous avez dû voir une forte hausse de la performance de la simulation et une nette amélioration de la fluidité.

Toutefois, votre programme effectue encore des calculs inutiles. En effet, à chaque pas de la simulation, la position et la direction de déplacement de toutes les billes sont mises à jour. Les billes qui ne bougent pas n'ont pas besoin d'être mises à jour. Faites donc en sorte de vérifier si la bille bouge avant de lancer les calculs de mise à jour. Voir la fonction `update` du fichier `engine.py`.

Une fois la modification effectuée, vérifiez qu'il y a bien une (petite) amélioration de la performance en comparant les résultats du programme benchmark avant et après optimisation. N'hésitez pas à lancer le benchmark plusieurs fois pour avoir une estimation moyenne de la performance.

(code) - Modifiez le code du moteur du jeu pour faire la mise à jour de position et direction de déplacement des billes qui bougent seulement.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte.

⚠N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Étape 4 : Optimisation algorithmique 2 bis

En partant du même principe que l'étape précédente, il est possible de réaliser cette optimisation bien plus tôt dans le programme. Dans le fichier `bench.py`, étudiez le fonctionnement du programme principal et notamment de la ligne :
`moving_balls = engine.how_many_moving_balls(balls)`

Il est possible de modifier cette fonction pour qu'elle nous retourne la liste des billes qui bougent et pas seulement leur nombre. Cette liste peut ensuite être utilisée pour ne lancer la fonction `update` que sur ces billes.

Une fois la modification effectuée, vérifiez qu'il y a bien une amélioration de la performance en comparant les résultats du programme benchmark avant et après optimisation.

(code) - Modifiez le code du moteur du jeu **et du benchmark** pour réaliser cette optimisation.

(code) - Faites de même pour le programme principal avec interface graphique.

(PDF) - Expliquez les changements que vous avez fait dans le code et argumentez pour montrer que la solution que vous proposez est correcte.

⚠N'oubliez pas de sauver le code de cette optimisation afin de l'inclure dans votre archive finale.

Conclusion

(PDF) - Comparez toutes les étapes d'optimisation en un seul graphique et commentez. Cette conclusion devra être faite **en anglais** en expliquant et comparant les différents algorithmes mis en place, le type d'optimisation et une quantification de leur impact sur le temps de calcul.

Partie optionnelle

Avez vous d'autres idées ? Expliquez les nous et, si vous avez le temps, codez les !