Bonjour,

Voici comment j’ai procédé pour optimiser le programme **matchmaking**.

Après une première lecture et une analyse rapide du code, j’ai constaté qu’effectivement, le code a été volontairement écrit de manière le moins optimisé possible.

Remarque : il semble que le code de MatchMake() a un bug : ça prend toujours les 20 premiers (dans l’ordre du tableau des Players ou d’ordre d’ajout) sans tenir compte du status ***myIsAvailable*** de ***Player***.

**Pour le code optimisé tient compte de *myIsAvailable* dans tous les cas.**

C’est pourquoi j’ai décidé :

1. d’optimiser (et de refactor aussi pour lire plus facilement le code) les choses "triviales" (facile et sans ambuguité)
2. chercher d’autres algorithmes de matchmaking pouvant mieux résoudre le problème (mais je n’en ai pas trouvé)
3. chercher si on peut remplacer le tableau ***matched*** et ***myPlayers*** par d’autres type de structures de données (arbre, heap, map…) ou remplacer les méthodes d’accès de ces tableaux
4. paralléliser les calculs (ou la recherche dans ***MatchMake()*** ou permettre de lancer plusieurs ***MatchMake()*** en parallèle.

# Optimisation « triviale » et sans changer l’algorithme

1. Utiliser le mutex de Windows et les fonctions ***WaitForSingleObject*** et ***ReleaseMutex*** à la place d’un ***volatile LONG*** et de ***while( \_InterlockedCompareExchange…)*** et ***\_InterlockedExchange***. Car avec le mutex de Windows, le thread est endormi par Windows tant que le mutex n’est pas disponible => moins de consommation du CPU (le ***while( \_InterlockedCompareExchange…)*** consomme 100% du CPU)
2. Remplacer les allocations dynamiques et recopies inutiles des objets ***Players*** et ***Matched***.
   1. Par exemple, remplacement du tableau ***matched*** :

Matched\*\* matched = new Matched\*[20];

for(unsigned int i = 0; i < 20; i++)

{

matched[i] = new Matched();

matched[i]->myDist = -1.0f;

matched[i]->myId = -1;

}

Par

Matched matchedItems[20];

Matched\* matched[20];

Matched\* pIter = matchedItems;

Matched\* pEnd = matchedItems + 20;

Matched\*\* p = matched;

for(; pIter < pEnd; ++pIter, ++p)

{

\*p = pIter;

}

Pourquoi un tableau de ***Matched*** ***matchedItems[20]*** et un de ***Matched\* matched[20]*** ?

Chaque éléments de ***matched*** est un pointer vers un élément de ***matchedItems*** évitant ainsi l’allocation dynamique et lors du tri (par std::sort()), on échange les pointeurs au lieu d’échanger des objets Matched.

* 1. Supprimer la copie d’objet ***Player*** pour ***playerToMatch*** puisqu’on ne modifie pas l’objet et pendant l’éxécution de ***MatchMake()***, la modification de la liste des Players n’est pas possible à cause du lock du mutex ***myLock***.

1. Remplacer, la fonction ***Dist***, le calcul de la distance euclidienne par un calcul de distance euclidienne au carré car ce qui nous importe ici est la comparaison < entre 2 distances. On évite ainsi ***sqrt()*** qui est très couteux tout comme pow((aA[i] - aB[i]), 2.0f) est plus lent que float d2 = aA[i] - aB[i]; d2 \*= d2;.
2. Remplacer ce code de la boucle principale for() : (Ce code qui permet de remplir ***matched*** avec les 20 premiers Players tout en triant ***matched*** à chaque ajout)

if(matchCount < 20)

{

matched[matchCount]->myId = myPlayers[i]->myPlayerId;

matched[matchCount]->myDist = Dist(myPlayers[i]->myPreferenceVector, playerToMatch->myPreferenceVector);

matchCount++;

using std::sort;

sort(matched, matched + matchCount, MatchComp);

continue;

}

Par un code qui remplit ***matched*** avec les 20 premiers Players puis trier ***matched*** avec std::sort() avant d’entrer dans la boucle for().

1. Déplacer avant le calcul de ***dist*** pour éviter de faire ce calcul inutilement.

if(!myPlayers[i]->myIsAvailable)

continue;

1. Supprimer ce code de la boucle for car il faut le faire une fois sortie de la boucle :

for(int j = 0; j < 20; j++)

aPlayerIds[j] = matched[j]->myId;

# Optimiser en remplaçant la méthode d’accès au tableau matched

Dans MatchMake(), pour chaque Player, on détermine si ajoute ce Player dans les résultat (et on retire le dernier du tableau ***matched***) en parcourant le tableau ***matched*** (qui est trié) depuis la fin puis l’y insérerant. C’est un algorithme de recherche en O(n) et de même pour l’insertion. Néanmoins, on y gagne en utilisant une recherche dichotomique.

# Optimiser en remplaçant le tableau matched par un « max heap »

Le « max heap » qui est une structure d’arbre binaire dont les éléments fils sont inférieurs ou égaux au père et l’élément racine est l’élément le plus grand de l’arbre. De plus, le max heap s’implémente en avec un tableau en interne.

En remplaçant le tableau ***matched*** par, on peut déterminer si on doit ajouter le Player ou pas en comparant avec le premier élément du tableau (à l’index 0) donc une recherche en O(1) et l’ajout d’élément est en O(ln n).

# Optimiser en remplaçant le tableau des Players par std::map

Il est clair qu’en remplaçant un tableau par une map, le temps de recherche est beaucoup plus rapide. Mais, dans ***MatchMake()*** on fait une fois cette recherche pour retrouver les information du Player qui cherche son matchmaking  et parcourir tous les éléments d’un map est plus lent que de parcourir un tableau.

# Optimiser en parallélisant la recherche

Ici on crée n worker threads (ou n est le nombre de cœurs ou de « hardware thread » du processeur) et on subdivise le tableau Player en n portions et on fait calculer, en parallèle, 20 matchmaking sur chaque portion par un worker thread.

Une fois les worker threads terminés, on fusionne les résultats pour en obtenir que les 20 meilleures.