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.

C'est pourquoi j'ai décidé :

1) d'optimiser et de refactor (pour lire plus facilement le code) les choses "triviales" (facile et sans ambiguïté)
2) de chercher d'autres algorithmes de matchmaking pouvant mieux résoudre le problème (mais je n'en ai pas trouvé)
3) de 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) de paralléliser les calculs (ou la recherche dans ***MatchMake()*** ou permettre de lancer plusieurs ***MatchMake()*** en parallèle.

Note :

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.

Le code RandomFloat32() dans la version `USE_PREDICTABLE_RANDOMNESS` équivaut à :

```
(float) ((((unsigned int) rand()) << 16) | rand()) / (float) 0xFFFFFFFF
```

Or `rand() retourne des valeurs entre 0 et` 32767 (ou 0x7FFF). Et donc, RandomFloat32() retourne des valeurs entre 0 et (float)0x7FFF7FFF / (float)0xFFFFFFFF) c'est-à-dire 0.499992371 au lieu des valeurs entre 0 et 1. De plus, certaines valeurs entre 0 et 0.499992371 ne sont jamais renvoyées.

Dans les tests, pour simuler la probabilité que 5% des joueurs sont connectés on fait le test RandomFloat32() < 0.05f. Avec ce RandomFloat32(), on obtiendrait plutôt une probabilité de 10% des joueurs connectés.

J'ai aussi fait une version qui génère un jeu de tests (données en entrée et données en sorties par ***MatchMake()***) avec le code originale. Dans cette version, il n'est plus question de lancer les tests avec 16 threads simultanément mais de lancer les tests l'un après l'autre à fin de pouvoir les rejouer dans le même ordre avec la version optimisée et vérifier qu'on obtient toujours les mêmes résultats.

Optimisation « triviale » et sans changer l'algorithme

Le code est dans Matchmaking_Basic. Le temps : 5 à 6 heures.

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***.

   a) 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.

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

3) 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;`.

4) 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. Or, std::sort() est souvent un quicksort et le quicksort a une très mauvaise performance quand le tableau est déjà presque trié)

```
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().

5) Déplacer le test suivant avant le calcul de ***dist*** pour éviter de faire ce calcul inutilement :

```
if(!myPlayers[i]->myIsAvailable)
  continue;
```

6) Supprimer le code suivant de la boucle for car il faut le faire une fois sorti de la boucle :

```
for(int j = 0; j < 20; j++)
  aPlayerIds[j] = matched[j]->myId;
```

7) Dans le code suivant :

```
int index = -1;
for(int j = 19; j >= 0; j--)
{
  if(matched[j]->myDist < dist)
  break;

  index = j;
}
```

la ligne

```
if(matched[j]->myDist < dist) break;
```

signifie qu'on continue d'itérer tant que dist est inférieur ou **égal** à l'élément du tableau. Or on peut s'arrêter dès que dist est supérieur ou égal à un élément tableau. On remplace donc ce code par :

```
if(matched[j]->myDist <= dist) break;
```

> Note : ce changement peut donner des résultats différents aux résultats de la version non optimisée tout en étant un bon résultat. Dans la version non optimisée, en cas d'égalité, c'est le dernier évalué qui sera prioritaire. C'est l'inverse pour la version optimisée.

8) Remplacer le code suivant :

```
for(int j = 19; j > index; j--)
{
  matched[j]->myDist      = matched[j - 1]->myDist;
  matched[j]->myId  = matched[j - 1]->myId;
}

matched[index]->myDist    = dist;
matched[index]->myId      = myPlayers[i]->myPlayerId;

  par

Matched* newItem = matched[19];
newItem->myDist = dist;
newItem->myId = player->myPlayerId;

for(int j = 19; j > index; --j)
{
  matched[j] = matched[j - 1];
}

matched[index] = newItem;
```

Ainsi, au lieu déplacer les éléments dans le tableau en échangeant leur valeur, on échange leur pointeur (puisque le tableau contient les pointeurs des éléments).

9) Supprimer le `printf("num players in system %u\n", myNumPlayers);` dans `AddUpdatePlayer()`.

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

Le code est dans Matchmaking_MaxHeap. Le temps : +1h à 2h (à partir de Matchmaking_Basic).

Dans MatchMake(), pour chaque Player, on détermine si l'on ajoute ce Player dans les résultats (et on retire le dernier du tableau ***matched***) en parcourant le tableau ***matched*** (qui est trié) depuis la fin puis en l'y insérerant. C'est un algorithme de recherche en O(n) et de même pour l'insertion (dans le pire des cas ; en pratique, le pire des cas n'est pas systématique).

Le « max-heap » qui est une structure d'arbre binaire dont les éléments fils sont inférieurs ou égaux au père et donc, l'élément racine est l'élément le plus grand de l'arbre. De plus, le max-heap s'implémente avec un tableau et avec la propriété suivante : les enfants de l'élément d'indice **i** sont les éléments d'indice **i * 2** et **i * 2 + 1** avec **i commençant à 1**.

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

Dans la version originale, les résultats retournés sont triés car l'algorithme trie les résultats tout au long de la recherche. Or, il ne semble pas être demandé par la spécification.

Avec le max-heap, les résultats ne sont pas triés. Si cela est nécessaire, il suffit d'exécuter std::sort_heap() sur le tableau avant de recopier les résultats pour les renvoyer.

Au final, par rapport à la version Matchmaking_Basic, cette version semble apporter un gain très faible mais le temps d'exécution de MatchMake() est plus stable que la version.

Note : même si je connais le fonctionnement de max-heap, je n'en ai jamais implémenté et j'ai peur de prendre trop de temps d'en implémenter un alors j'ai utilisé les fonctions make_heap() et push_heap() de la STL.

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

Le code est dans Matchmaking_Map.

Le temps : +1h (à partir de Matchmaking_Basic).

En remplaçant un tableau par une map, le temps de recherche est beaucoup plus rapide (en contre-partie, ça demande plus de mémoire). On gagne du temps aussi lors des ajouts et modifications des Players.

Mais, dans ***MatchMake()*** on ne fait qu'une fois cette recherche pour retrouver les information du Player qui cherche son matchmaking ; alors que parcourir tous les éléments d'une map est plus lent que de parcourir un tableau. Au final, ça s'est avéré plus lent que la version Matchmaking_Basic.

La solution est de combiner d'une map et d'un tableau pour la liste des Players (heureusement, que nous n'avons pas de suppression de Player).L'occupation de la mémoire est encore plus importante.

Mais finalement, cette solution ne semble pas être plus rapide que Matchmaking_Basic.

Optimiser en parallélisant la recherche

Le code est dans Matchmaking_MultiThread.

Le temps : +2h (à partir de Matchmaking_MaxHeap).

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

Une fois les worker threads terminés, on fusionne les résultats et on prend les 20 meilleures.

Cette version apporte un gain visible mais décevant car si on gagne du temps dans la recherche des Players qui matchent, on y reperd lors de la fusion des résultats. En plus, l'exécution du premier appel de MatchMake() est bien plus lente ; très probablement à cause de la création des threads (les suivantes sont plus rapide car il semble que Windows a un système de cache pour la création des threads).

Optimiser en permettant l'éxécution de plusieurs MatchMake() en même temps

Le code est dans Matchmaking_SRWL_Basic (basant sur Matchmaking_Basic) et Matchmaking_SRWL_MaxHeap (basant sur Matchmaking_MaxHeap).

Le temps : +1 heures (à partir de Matchmaking_Basic / Matchmaking_MaxHeap).

Il s'agit d'utiliser la méthode « n readers – 1 writer » car ***MatchMake()*** ne modifie pas la liste des Players. Seules les méthodes ***AddUpdatePlayer(),SetPlayerAvailable()*** et ***SetPlayerUnavailable()*** qui modifient la liste des Players.

Pour cela, j'utilise « Slim Reader/Writer (SRW) Locks » de Windows à la place d'un simple mutex :

- dans ***MatchMake()***, on fait un « Reader Lock »
- dans ***AddUpdatePlayer(),SetPlayerAvailable()*** et ***SetPlayerUnavailable()***, on fait un « Writer Lock »

Note : les SRW Locks de Windows nécéssitent au moins Windows 7 et surtout compiler le code avec `_WIN32_WINNT` défini avec 0x0600 dans stdafx.h. (J'utilise Visual Studio 2015)

On obtient de bon gain par rapport à leurs versions de base.

Géneration des jeux de tests

Le code est dans Matchmaking_GenTests.

Le temps : 3 à 4 heures.

Conclusion

Comme je ne suis pas vraiment limité par le temps alors j'ai implémenté plusieurs versions d'optimisation mais si je ne devais choisir qu'une, je choisirais la version avec les « Slim Reader/Writer (SRW) Locks » car c'est simple et efficace.