**TP4 C++ : Héritage et Polymorphisme**

**Performances de l’application**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Temps | 100 objets | 1 000 objets | 10 000 objets | 100 000 objets | 1 000 000 objets |
| Écriture dans un fichier (SAVE) | 0,258 ms | 0,618 ms | 2,69 ms | 29,312 ms | 289,018 ms |
| Lecture d’un fichier (LOAD) | 1,14 ms | 9,383 ms | 60,79 ms | 492,278 ms | 5151,1 ms |
| Lecture d’un fichier  (Entrée standard) | 4 ms | 21 ms | 120 ms | 1031 ms | 10874 ms |

*Test du LOAD à partir d'un fichier:*

Le fichier à charger est constitué de n commandes de création de Figures dont 1/4 de chaque type (C, R, L, PL).

On mesure le temps nécessaire à l'exécution de la méthode *Container.Load(«fichierLoad»);*

*Test du SAVE:*

On charge en mémoire un fichier de n commandes de création (LOAD) puis on mesure le temps nécessaire à l'exécution de la méthode *Container.Save(«fichierSave»)*;

*Test d'insertion à partir d'un fichier redirigé dans l'entrée standard:*

Le fichier à exécuter en entrée est constitué de n commandes de création de Figures dont 1/4 de chaque type (C, R, L, PL).

On redirige la sortie vers un fichier temporaire afin de ne pas subir les temps d'affichage de création (OK et #)

On mesure le temps global de l'exécution de l'application avec la commande *time*. Le résultat est la somme des *time user* et *time system.* Cela comprend donc la destruction de tous les objets.

*Méthodes d'Optimisation :*

Afin de réduire les temps d'exécution, nous avons développé quelques optimisations :

-La ligne de commande saisie par l'utilisateur est analysée, découpée et stockée une seule fois afin d'en extraire les mots-clés. Le découpage se fait dans Tools::Split() en utilisant une stringstream. Le stockage se fait dans une liste qui est plus optimale (mémoire et temps de création) que le vector si on n'utilise pas l'accès aléatoire.

-La détection du type de commande dans le main se fait par une succession de comparaisons en commençant par les cas les plus probables (C, R, L, PL, S, LIST...LOAD, SAVE, CLEAR). Ceci afin de trouver le bon type de commande le plus rapidement possible.

-Pour le stockage des Graphics, une map est utilisée afin d'optimiser l'insertion (log n), la suppression (log n) et l'accès (log n). Cela améliore la rapidité d'affichage car la map est déjà triée.

-La détection de présence d'un nom(donc d'un objet) dans la map étant l'opération la plus utilisée, et de complexité (log n), nous l'avons optimisé à l'aide d'un unordered\_set<string>. Celle-ci contient les noms des objets actifs et se comporte comme une table de hash permettant de réduire la complexité de cette opération à un temps constant au mieux. Nous sommes conscients de la duplication des données dans la mémoire mais considérons que le gain apporté en temps vaut ce sacrifice.

-Nous avons optimisé les temps d'exécution des UNDO/REDO (pour toutes les commandes d'insertion et de suppression) par deux moyens :

-La non-duplication de nos éléments graphiques et leur manipulation par les différentes classes (Container, Command, Selection...) via des pointeurs exclusivement.

-La gestion de la libération non-instantanée (par rapport aux DELETE) de la mémoire (cf spec : *Comment est gérée la mémoire ?*) évitant les appels successifs de *delete* et *new*.

-Le calcul du rectangle « fictif » dans le constructeur des Figures (cf spec *Comment savoir si une Figure est incluse dans une sélection ?*) permet de n'effectuer le calcul qu'une seule fois et d'optimiser les tests d'appartenance à une sélection lors de la création de celle-ci.