TP2 – Neige

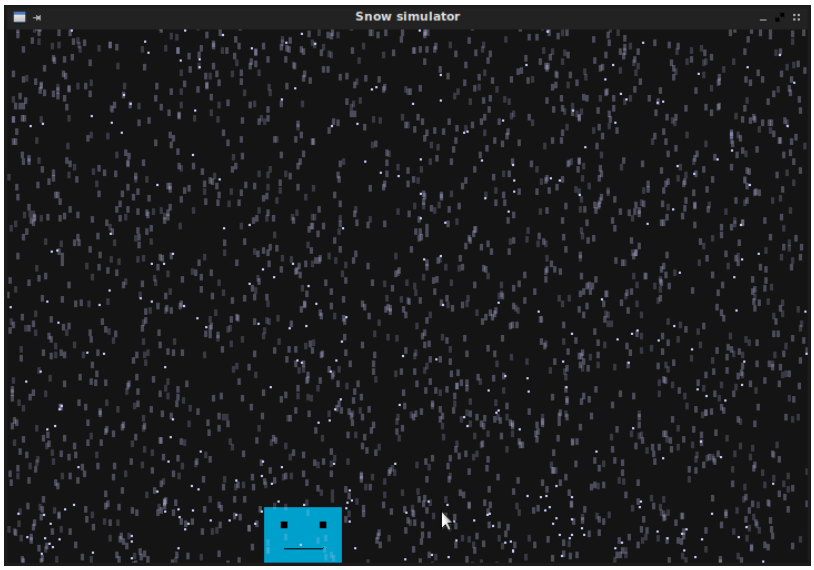
Programmation Avancée de Jeux Vidéos – Nicolas Hurtubise

Contexte

C'est l'hiver! Notre petit personnage se promène dans la neige. Il n'y a **pas beaucoup de flocons** pour le moment...

Votre tâche sera d'implanter **trois optimisations** dans le code pour ajouter plus de magie hivernale avec un nombre maximal de particules de neiges.

Vous devrez également répondre à des **questions de compréhension** dans le rapport que vous me remettrez.



Interface graphique

Le jeu est affiché dans une fenêtre SDL. Les contrôles sont les suivants :

- *Flèches (haut/bas/gauche/droite)* : déplacer/sauter/pencher le personnage
- *Barre espace* : sauter
- *Numéros 1 à 9* : relance le jeu avec un nombre prédéfini de particules

Structure du code

La *game loop* de base est fournie dans la classe `Game`, la classe `GameSnow` implante le code spécifique au jeu.

Le code représente différents types de flocons avec de l'héritage :

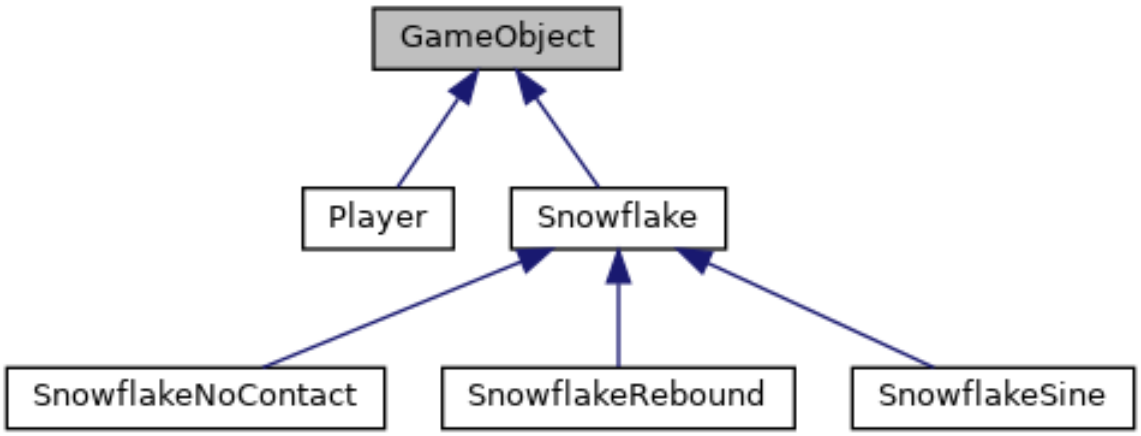


Figure 1: Hiérarchie de `GameObjects`

0 – Avant de commencer

Commencez par comprendre ce qui compose chaque objet

Q1 – Quelle est la taille (en bytes, avec `sizeof()`) des objets suivants si vous compilez le code en 64 bits?

- `GameObject`
- `Player`
- `Snowflake`
- `SnowflakeNoContact`
- `SnowflakeRebound`
- `SnowflakeSine`

1 – Object Pool

Le code dans son état actuel génère des flocons avec la fonction

```
void GameSnow::addSnowflake(double x, double y)
```

qui crée des flocons au hasard d'un des quatre types

- `Snowflake`
- `SnowflakeNoContact`
- `SnowflakeRebound`
- `SnowflakeSine`

et les ajoute au vecteur `std::vector<std::unique_ptr<Snowflake>> snowflakes;`

Le code fait donc beaucoup de `new` et de `delete` indirectement, à travers l'utilisation de `unique_ptr<>` (chaque `make_unique<>` fait un `new` derrière).

Ces `new` et `delete` sont une source de ralentissement du programme.

Pour accélérer le jeu, on va **préallouer un nombre fixe** de particules de neige (à vous de choisir combien, vous pourrez augmenter le nombre à mesure que les optimisations vous le permettent) et **incrémenter** ou **décrémenter** un compteur du nombre de particules actives au lieu de faire des `new`/`delete`.

On pourrait se faire un tableau dans ce style :

```
std::unique_ptr<Snowflake[]> snowflakes;
size_t nbSnowflakes;
```

Au moment de *créer* un flocon, on peut simplement *incrémenter* le nombre de flocons utilisés, et réinitialiser le prochain flocon disponible :

```
// Valider qu'on ne dépasse pas le MAX disponible
if(nbSnowflakes < MAX_SNOWFLAKES) {

    // Réinitialiser le prochain snowflake inactif
    snowflakes[nbSnowflakes].init( ... );

    nbSnowflakes++; // Le nombre total augmente de +1
}
```

Au moment de supprimer un flocon, on peut plutôt *déplacer le dernier flocon utilisé* à la place de celui qu'on souhaite enlever

```
size_t dernierIdxDuTableau = nbSnowflakes - 1;

// On déplace les valeurs du dernier snowflake de la liste
// par-dessus l'index à supprimer
snowflakes[deleteIdx].init( {valeurs des attributs de : snowflakes[dernierIdxDuTableau]} );

nbSnowflakes--; // Le nombre total diminue de 1
```

Esquisse de code

Pour gérer tout ça, faites-vous une nouvelle classe `SnowflakePool`, du genre :

```
#include <cstddef> // pour size_t
#include <memory> // pour unique_ptr

class SnowflakePool
{
public:
    SnowflakePool() : nbParticles{0}
    {
        // Initialiser le pool d'objets
    }

    ... spawn(...) // À compléter
    {
        // "Créer" un snowflake de plus en activant
        // le prochain snowflake inactif
    }

    void destroy(size_t deleteIdx)
    {
        // "Supprimer" le snowflake demandé en déplaçant
        // le dernier snowflake utilisé par-dessus l'objet
        // à supprimer
    }

    Snowflake& get(size_t idx)
    {
        // Accès à un des snowflakes
    }

    size_t getNbSnowflakes() const
    {
        return nbParticles;
    }
private:
    std::unique_ptr<Snowflake[]> pool;
    size_t nbParticles;
};
```

Commencez par **ignorer les trois sous-types de `Snowflake`** et faites un *Object Pool* pour la classe principale seulement : dans la fonction `GameSnow::addSnowflake()`, générez uniquement des `Snowflake` ordinaires.

Généraliser aux différents types

Seulement une fois que c'est fait, essayez de mettre différents sous-types de `Snowflake` dans votre `ObjetPool`...

> **Q2 – Vous allez avoir un problème si vous essayez d'ajouter différents types de `Snowflake` au tableau de `Snowflake[]`.**
>
> **Expliquez pourquoi ça ne peut pas fonctionner en C++, alors que ça fonctionnerait sans problème en C#.**
>
> Vous pouvez me faire un dessin si ça vous aide à expliquer

Pour résoudre le problème, faites **quatre pools différents**, un pour chaque type/sous-type de `Snowflake`. Ne copiez-collez pas la classe `SnowflakePool`... Utilisez des **templates** pour adapter la classe!

Notez : vous **devez** finir avec du code équivalent au code d'origine. Dans le code d'origine, chaque snowflake qui tombe est automatiquement remplacé par un **type de snowflake au hasard** (quand un `SnowflakeSine` disparaît, il pourrait être remplacé par un `SnowflakeNoContact`, ou par un `SnowflakeRebound`, etc).

Vous ne pouvez **pas** faire respawner en haut le même flocon qui vient de tomber au sol.

2 – Retirer les allocations inutiles

Une section du code fait encore beaucoup d'allocations inutilement. Utilisez le profileur pour trouver de quelle section il s'agit. Corrigez le problème.

Indice: Pensez au fait que des `new/delete` sont implicitement faits par certains objets.

> **Q3 – Donnez une capture d'écran du profileur qui montre que la section du code que vous avez modifiée valait la peine d'être optimisée.**

3 – Réduire la taille des objets

Dans l'état actuel du code, chaque `GameObject` contient des tableaux relativement gros et non-utilisés :

```
double bloop[256];
...
double blaap[256];
...
double bleep[256];
...
double bliip[256];
...
double bluup[256];
...
double blyyp[256];
```

Ces tableaux font partie de la structure `GameObject` directement, ce qui rend la taille d'un `GameObject` assez élevée.

Supprimez ensuite tous ces tableaux inutiles et répondez à la question suivante :

Q4 – Comment expliquez-vous le fait que retirer des tableaux qui n'étaient pas utilisés accélère le programme?
(Si jamais ça n'a pas d'impact chez vous, je vous confirme que ça me permet d'ajouter pas mal de particules sans lag sur mon ordinateur!)

Est-ce...

1. À cause de l'ordre Big-O de l'algorithme après modification
2. À cause du nombre d'allocations faites pendant l'exécution du programme
3. À cause de l'impact sur l'utilisation de la cache
4. À cause de la prédiction de branches du processeur
5. À cause des *smart pointers*
6. À cause des garçons

Expliquez votre réponse

Un peu de réflexion...

Q5 – Pourriez-vous ajouter l'optimisation de Grille d'accélération que vous aviez implantée dans le TP1? Est-ce que ça serait utile ici?

Ne l'implantez pas, répondez seulement théoriquement!

Note sur les pointeurs

Vous **devez** mettre en pratique l'utilisation de *smart pointers*.

Il ne devrait **pas** y avoir de `new` ni de `delete` explicites dans votre programme. Vous n'avez **pas** besoin de `shared_ptr`, vous allez remarquer que les `unique_ptr` sont suffisants.

Rapport

En plus de votre code optimisé, vous devez remettre un rapport qui répond à la question : combien de particules de neige arrivez-vous à ajouter avant d'obtenir le premier message de lag dans la console? (Donnez le chiffre **seulement en mode `Release`**)

1. **Avant de modifier le programme**
2. **Après avoir ajouté l'Object Pool**
3. **Après avoir retiré les allocations inutiles**
4. **Après avoir réduit la taille des objets**

Répondez également aux questions **Q1 à Q5** dans votre rapport.

Notez que je ne vous demande qu'une seule capture d'écran du profileur pour ce travail

Remise

Vous devez remettre sur Léa :

- Votre code **dans un fichier .zip**, qui ne contient **pas** les fichiers temporaires de Visual Studio (un dossier nommé `.vs`)
  - Votre fichier `.zip` ne devrait pas prendre plus que ~1Mo, si ça pèse plusieurs dizaines de Mo, il y a un problème
- Votre rapport **au format PDF**

Barème

- 50% : Fonctionnalités demandées implantées correctement
  - Trois optimisations (grande majorité des points accordés sur votre implantation de l'Object Pool)
- 40% : Rapport
  - Réponses aux questions
  - Discussion/analyse des optimisations
- 10% : Qualité du code
  - Code bien commenté, bien découpé en fonctions au besoin, ...

Note sur le plagiat

Le travail est à faire **individuellement**. Ne partagez pas de code, ça constituerait un **plagiat**, et vous auriez *zéro*.