UNIVERSITÉ
CAEN
NORMANDIE

Rapport Simulateur à N corps

Travail Personnel Approfondi

AGBODJAN Wilfried - 21914933
LETELLIER Guillaume - 21804030
MORLAY Antoine - 21803153
PIGNARD Alexandre - 21701890

L2 Informatique - Promotion 2019-2020

25 avril 2020

Table des matières

1 Introduction

Pour n'importe quel développeur informatique, la programmation orientée objet est importante pour factoriser du code et donc éviter la redondance, mais aussi pour associer des méthodes et attributs à un objet précisément. Cela permet donc d'encadrer l'utilisation du programme. Les étudiants en 2^{ime} année de licence informatique doivent donc réaliser un projet dans la langage de programmation Java afin d'utiliser cette connaissance dans le cadre de l'unité d'enseignement de Conception Logicielle 2.

Par groupe de 4, nous devions choisir un des projets proposés qui portaient sur des thèmes variés comme l'astrophysique, le jeu vidéo, etc.

Nous avons opté pour le projet portant sur le simulateur à N corps. L'énoncé est le suivant : « Le problème à N corps est un problème d'astronomie classique où plusieurs corps se déplacent dans l'espace en étant soumis à leur propre inertie et l'attraction des autres corps. L'équation différentielle qui modélise ce problème est en pratique inutilisable pour N $>$ 2. Le but de ce projet est dans un premier de simuler un espace newtonien où N corps interagissent et visualiser cette simulation. Il s'agira ensuite d'améliorer ce simulateur avec diverses propositions parmi les suivantes : instancier des chorégraphies à N corps, accélérer l'optimisation avec un découpage spatial récursif, intégrer un jeu de pilotage d'un corps au clavier, développer une IA pour optimiser les déplacements avec une trajectoire faible en énergie comme les orbites de transfert.

Nous avons choisi de découpé le projet en 2 parties :
- création de toute la partie mathématique et physique utile au projet ;
- conception de l'interface graphique.

2 Problème à N corps

Le problème à N corps consiste à résoudre les équations différentielles de mouvement établies par Newton de N corps interagissant gravitationnellement entre eux. L'équation principale pour simuler les N corps avec l'équation 1.

$$m_j \ddot{\overrightarrow{a_j}} = \sum_{i \in \{1,\dots,N\} \setminus \{j\}} -G \frac{m_i m_j (\overrightarrow{pos_i} - \overrightarrow{pos_j})}{\left\| \overrightarrow{pos_i} - \overrightarrow{pos_i} \right\|^3} \tag{1}$$

Pour des raisons d'optimisation de calculs, nous allons utiliser l'équation 2. Comme on peut le voir, c'est exactement la même équation que la 1, mais celle-ci permet de réaliser moins de calculs. Nous avons calculé de notre côté, qu'il y a une optimisation de 20% environ.

$$\overrightarrow{a_j} = -G \sum_{i \in \{1,\dots,N\} \setminus \{j\}} \frac{m_i (\overrightarrow{pos_i} - \overrightarrow{pos_j})}{\left\| \overrightarrow{pos_i} - \overrightarrow{pos_i} \right\|^3} \tag{2}$$

L'équation 2 nous permet donc de calculer l'accélération de chaque corps en fonction des autres. Néanmoins, le travail n'est pas terminé, il nous faut encore calculer la nouvelle vitesse et la nouvelle position. L'équation 3 permet de calculer la vitesse d'un objet en fonction de la position initiale $\overrightarrow{v_i}$ et de l'accélération subit $\overrightarrow{a}$.

$$\overrightarrow{v_f} = \overrightarrow{v_i} + \overrightarrow{a}t \tag{3}$$

Pour calculer la position, nous appliquons l'équation 4. Elle est en fonction de la position initiale de l'objet $\overrightarrow{pos_i}$, de la vitesse actuelle de l'objet $\overrightarrow{v}$ et de l'accélération $\overrightarrow{a}$.

$$\overrightarrow{pos_f} = \overrightarrow{pos_i} + \overrightarrow{v} \times t \tag{4}$$

Toutes ces équations utilisent la notation vectorielle. Cela signifie que l'on applique l'équation sur toutes les lignes des vecteurs présents dans l'équation. De manière physique, on applique les formules sur chaque axes représentés (dans notre cas, sur les 3 axes d'espace).

Nous allons donc voir comment nous allons appliquer ces différentes formules dans la section 5.

3 Manuel d'utilisation

3.1 Les scripts disponibles

A partir du dossier `nbodysimulator`, nous avons un sous-dossier `scripts` possédant des scripts afin d'exécuter différentes actions comme télécharger les librairies externes, compiler, ou encore lancer l'application. Pour des raisons de compatibilité, nous avons fait des scripts `.sh` pour Linux/MacOS, et des scripts `.bat` pour Windows. Ces scripts lance pour certains, des scripts `ant` comme Mr Bonnet a pu nous conseiller d'utiliser. Il faut donc qu'il soit préalablement installé sur la machine utilisée.

Note importante : Pour commencer à utiliser notre application, nous devons lancer les scripts en étant dans le dossier `nbodysimulator`.

Le premier script à lancer est celui qui va permettre d'installer toutes les librairies externes (JavaFX). Nous n'avons qu'à lancer `sh scripts/install.sh` (resp. `scripts\install.sh` sous Windows). Celui-ci va télécharger les librairies, mais aussi désarchiver le fichier `.zip`, copier les fichiers contenus dans celui-ci dans les dossiers `bin` et `lib` afin de permettre un lancement simple de l'application.

Pour lancer l'application, nous pouvons faire simplement un `sh scripts/run.sh` (resp. `scripts\run.bat`). Ce script va de lui-même lancer le script pour compiler l'ensemble du projet et installer les librairies si elles ne sont pas installées. Ceci permet de faire un lancement rapide de l'application. Nous arrivons donc sur la page d'accueil de l'application montrée à la figure 1.

N.B. : Nous pouvons aussi lancer ce script sans avoir lancer préalablement celui pour l'installation des librairies externes.

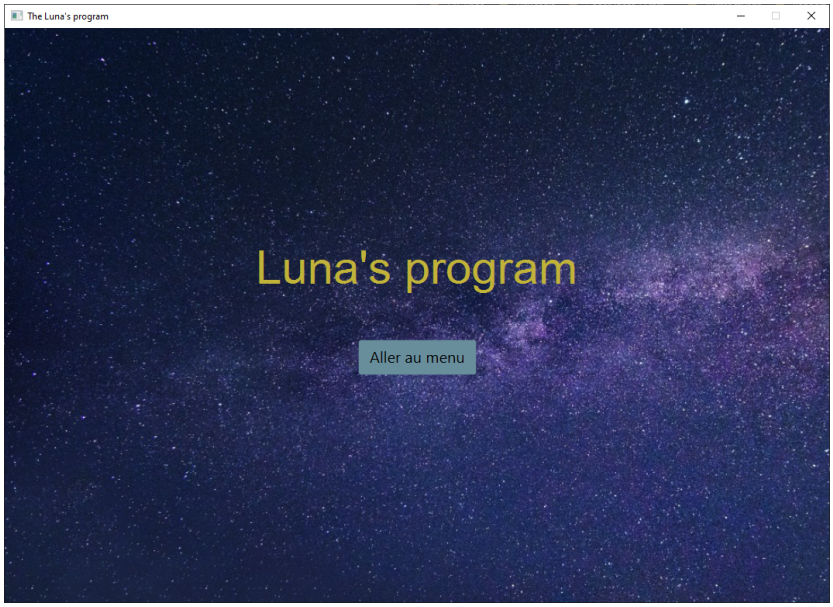

FIGURE 1 – Page d'accueil de l'application

Enfin, nous avons d'autres scripts (toujours dans le dossier `scripts`) pour compiler (`compile.sh`, resp. `compile.bat`), faire la JavaDoc (`makedoc.sh`, resp. `makedoc.bat`), faire un fichier `.jar` (`makejar.sh`, resp. `makejar.bat`), supprimer tous les dossiers créés (`clean.sh`, resp. `clean.bat`, ou encore, pour lancer les classes tests de l'application (`test.sh`, resp. `test.bat`). Nous avons pour terminer, un script `dist` à la racine du dossier qui permet de distribuer le code source simplement et rapidement sous forme d'archive `.tar`.

3.2 L'application

Après la figure 1, nous arrivons au menu (figure 2. Comme nous pouvons le voir, nous avons différents choix :

— créer une simulation aléatoire ;

— créer une simulation à partir d'un fichier ;
— créer une simulation personnalisée.

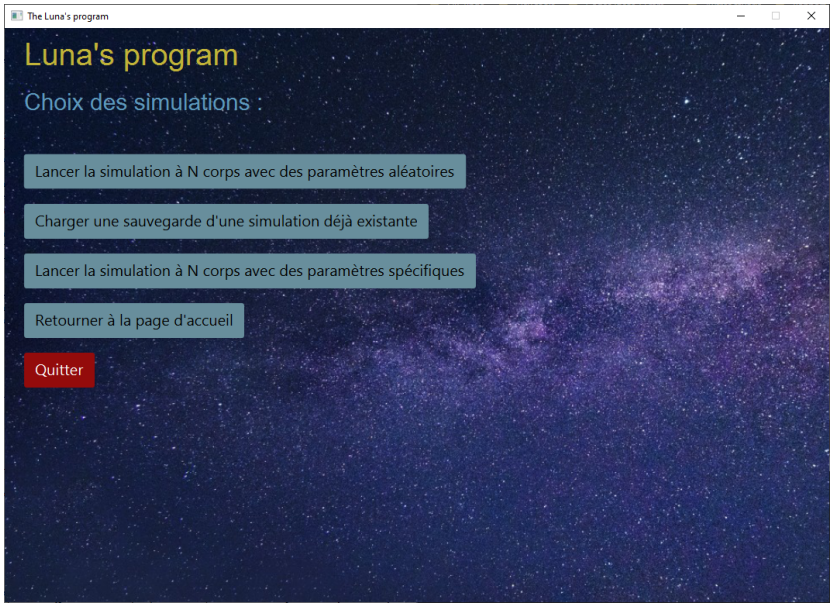

FIGURE 2 – Menu principal de l'application

En remplissant l'une des trois possibilités, nous arrivons à la simulation elle-même (figure 3). Pour la partie de gauche, nous pouvons faire tourner les objets à partir du centre et zoomer/dézoomer (avec le scroll de la souris). Pour la partie de droite, nous avons un slider pour gérer la vitesse de simulation, un slider pour gérer la précision de la simulation utilisée pour l'algorithme de Barnes-Hut (plus c'est haut, moins c'est précis) et un checkbox (paramètre "A une vache près) qui quand il est coché, active l'utilisation de l'algorithme de Barnes-Hut et sinon, utilise l'algorithme classique. Pour la liste des corps, nous pouvons sélectionner un corps et une boîte de dialogue s'ouvre pour modifier ses attributs.

4 Organisation du projet

4.1 Répartition des tâches

Le projet étant découpés en deux parties principales, nous avons tout d'abord commencé par l'élaboration d'un plan de travail dès les premières séances. Nous avons ainsi commencé par cerner les grands axes du projet et avons ainsi procédé à l'élaboration d'une modélisation du projet en UML (Unified Modeling Language). Afin d'optimiser le temps de travail et d'être plus prolifique, nous nous sommes dans un premier temps séparés en binômes. Guillaume s'est

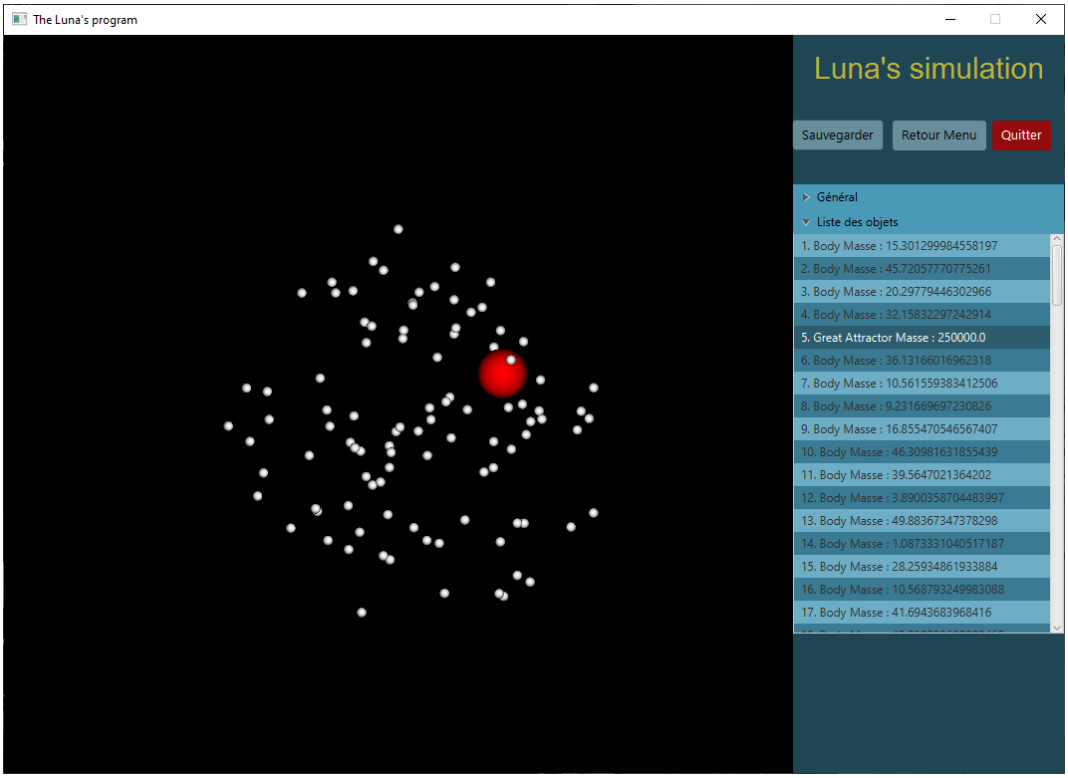

FIGURE 3 – Scène de simulation de l'application

concentré sur l'intégration de toute la partie mathématique et physique utile au projet, tandis qu'Alexandre et Antoine ont oeuvré pour la conception de l'interface graphique.

4.2 SVN

Afin de pouvoir travailler tous sur le projet, nous avons utilisé un service rendu disponible par l'université nommé Forge et plus précisément, SVN [1].

Forge est une plateforme qui permet de faire circuler des informations relatives à des projets, de recenser les anomalies et les tâches à réaliser, de disposer d'un dépôt de sources du projet géré par SVN ou Git.

SVN est un gestionnaire d'arborescence de fichiers qui :

— garde en mémoire toutes les modifications effectuées ;

— permet de revenir à une version antérieure ;

— gère les conflits d'écriture sur un même fichier.

1. SubVersioN permet de centraliser les projets sur un serveur

Partie du projet	Tâche effectuée	Membre qui l'a réalisée
Mathématiques	Système matriciel et vectorielle	Guillaume
	Tests	
Physique	Application des formules basiques	
	Base du simulateur	
	Tests	
Barnes-Hut	Algorithmie de Barnes-Hut	
	Tests	
Package *nbody*	Amélioration des corps	Alexandre
	Parser de fichier (chargement et écriture)	
	Amélioration simulateur	Guillaume
Interface	Création initiale de l'espace de simulation	Alexandre
	Ajout de certaines fonctionnalités (zoom, rotation)	
	Création de tout le reste de l'interface (page d'accueil, menu, etc)	Antoine
	Arrangements de l'interface	
	Stylisation	
Expression	Rédaction du rapport	Tous les membres
	Rédaction du diaporama de la soutenance	

4.3 Librairies utilisées

Pour implémenter l'interface graphique de notre application, nous nous sommes servis de la bibliothèque d'interface **JavaFX**. Qu'est-ce que JavaFX ? JavaFX est un framework et une bibliothèque d'interface utilisateur issue du projet OpenJFX, qui permet aux développeurs Java de créer une interface graphique pour des applications de bureau, des applications internet riches et des applications smartphones et tablettes tactiles.

Créé à l'origine par Sun MicroSystems, puis développé par Oracle après son rachat et ce, jusqu'à la version 11 du JDK, c'est depuis lors à la communauté OpenJFX que revient la poursuite de son développement. Cette bibliothèque a été conçue pour remplacer Swing et AWT, qui ont été développés à partir de la fin des années 90, afin de pallier les défauts de ces derniers et fournir de nouvelles fonctionnalités (dont le support des écrans tactiles).

Le cycle de sortie d'une nouvelle version de JavaFX correspond à celui de Java, soit tous les 6 mois.

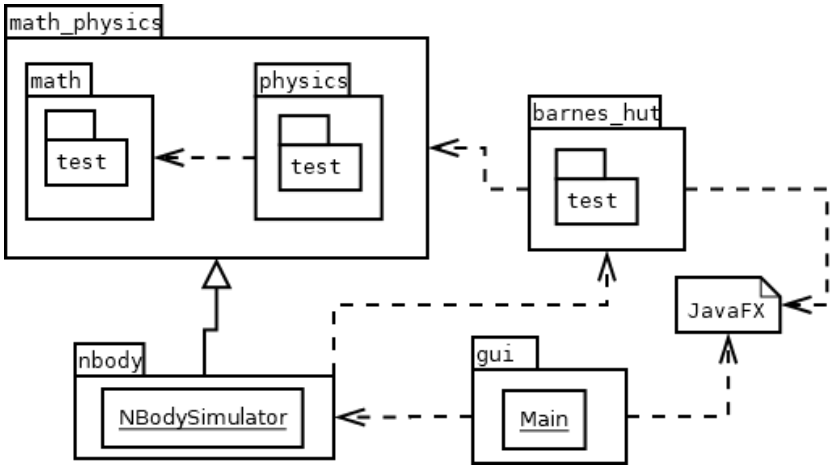

FIGURE 4 – Diagramme des packages

4.4 Architecture du programme

Le programme a été séparé en plusieurs packages en voulant répondre au mieux au sujet. Nous avons essayé aussi de suivre le modèle MVC que nous avons pu voir en Compléments de POO. Nous avons séparé le programme dans les packages suivants :

math_physics contient 2 sous-packages, `math` qui contient toute la base mathématiques utile au projet, et `physics` qui contient toute la logique physique de base et un simulateur basique.

barnes_hut contient les classes implémentant l'algorithme de Barnes-Hut (section 5.3), qui permet d'accélérer les calculs.

nbody contient les classes qui permettent de relier l'interface graphique et la partie arrière de l'application. C'est ici que le simulateur utilisé est créé (permet l'utilisation de l'algorithme N corps classique et celui de Barnes-Hut).

gui (pour Graphic User Interface) contient les classes de l'interface graphique.

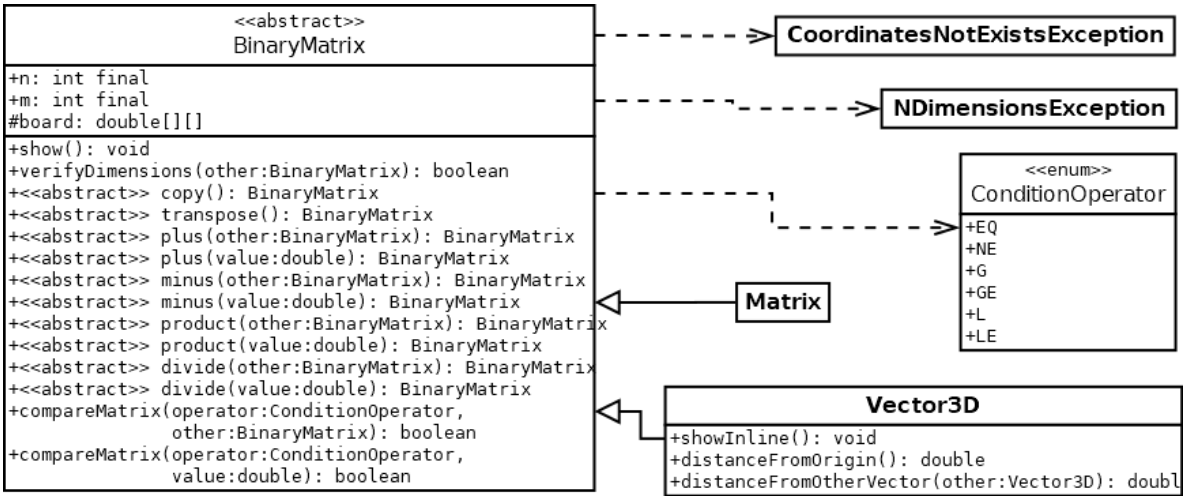

FIGURE 5 – Diagramme du package **math_physics.math**

5 Spécificités techniques

5.1 Ordre d'exécution du programme

Nous avons décidé de créer différents types de simulation. En intégrant ces différents types de simulation, l'ordre d'exécution du programme a donc du être revu. La figure 10 montre l'ordre d'exécution de notre application de manière schématique.

Nous avons voulu distinguer différents types de simulation :

- la simulation aléatoire ;
- le chargement d'une simulation déjà effectuée (ou créée par un logiciel externe) ;
- et la création/instanciation manuelle d'une simulation.

La génération aléatoire de corps est effectuée en fonction de paramètres choisis par l'utilisateur avant de générer les corps tels que le nombre de corps à créer, la masse maximale autorisée pour les corps, la distance maximale du centre de la simulation ou encore le rayon des corps.

Pour le choix du fichier, n'importe quel fichier peut être ouvert tant que les informations se trouvant dans le fichier respecte les normes de lecture de notre application. L'utilisateur peut choisir de modifier le fichier chargé et de le sauvegarder avant de commencer la simulation.

Enfin, la création manuelle des corps laisse pleinement le choix à l'utilisateur des différents attributs des corps. Il peut créer n'importe quel corps ayant un nom, une masse, une position initiale, une vélocité initiale, un rayon, ou encore, une couleur pour afficher le corps que l'utilisateur pourra choisir. L'utilisateur peut continuer à créer des corps jusqu'à ce qu'il ne

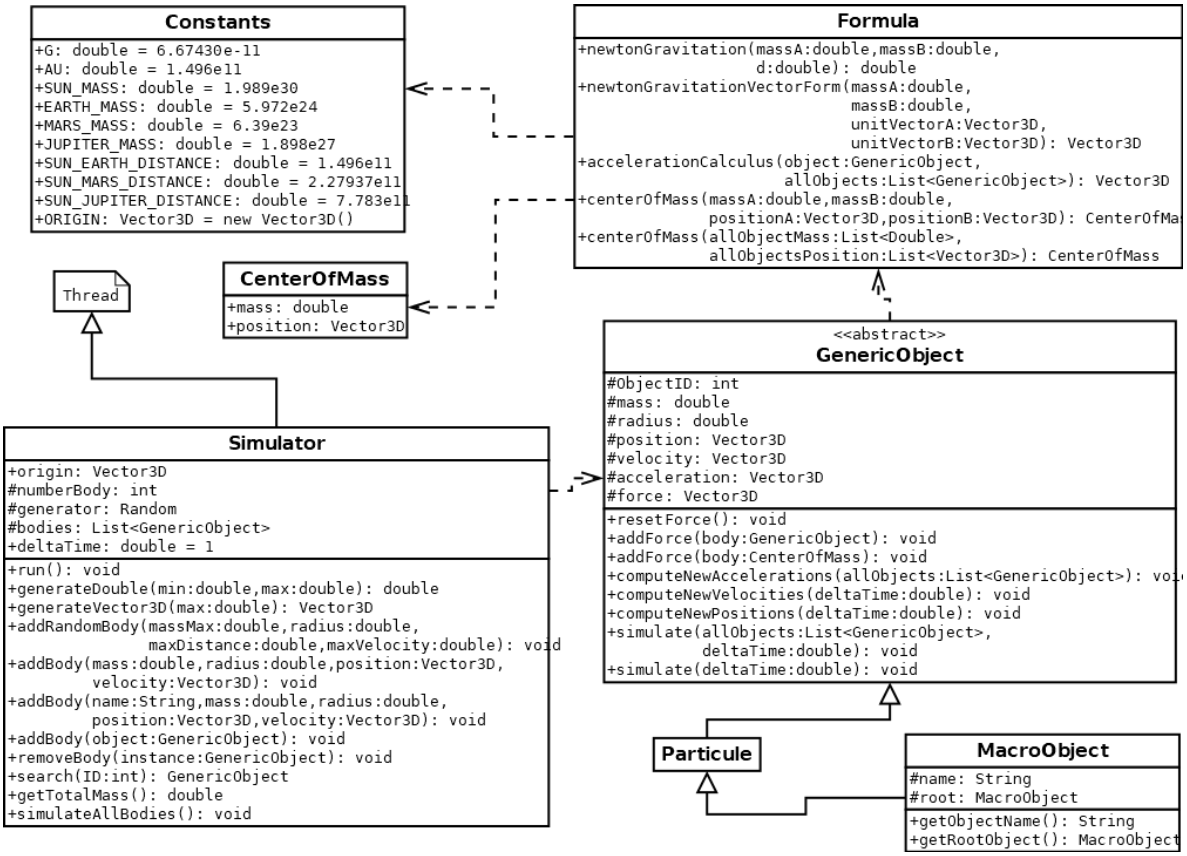

FIGURE 6 – Diagramme du package **math_physics.physics**

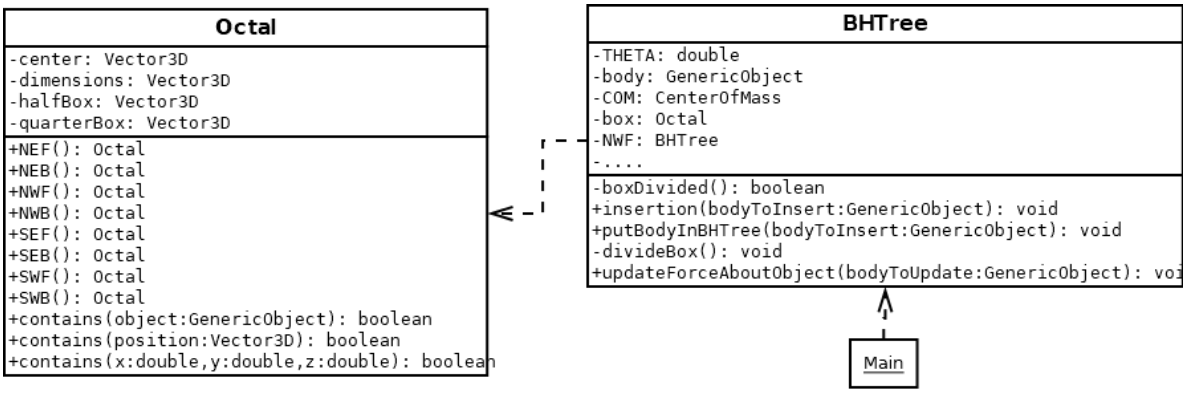

FIGURE 7 – Diagramme du package **barnes_hut**

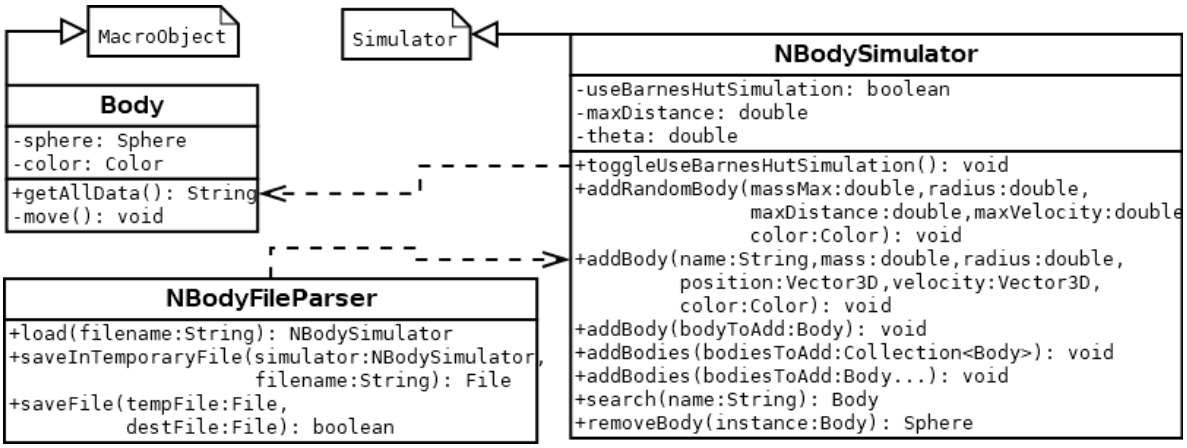

FIGURE 8 – Diagramme du package **nbody**

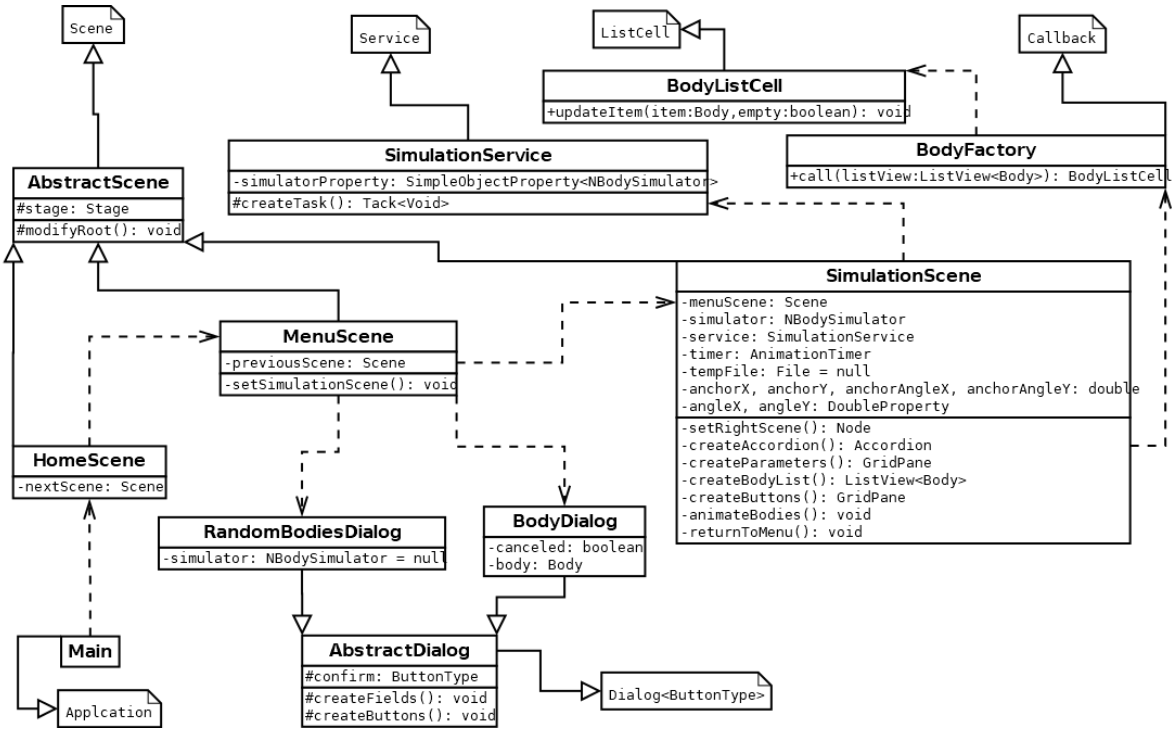

FIGURE 9 – Diagramme du package **gui**

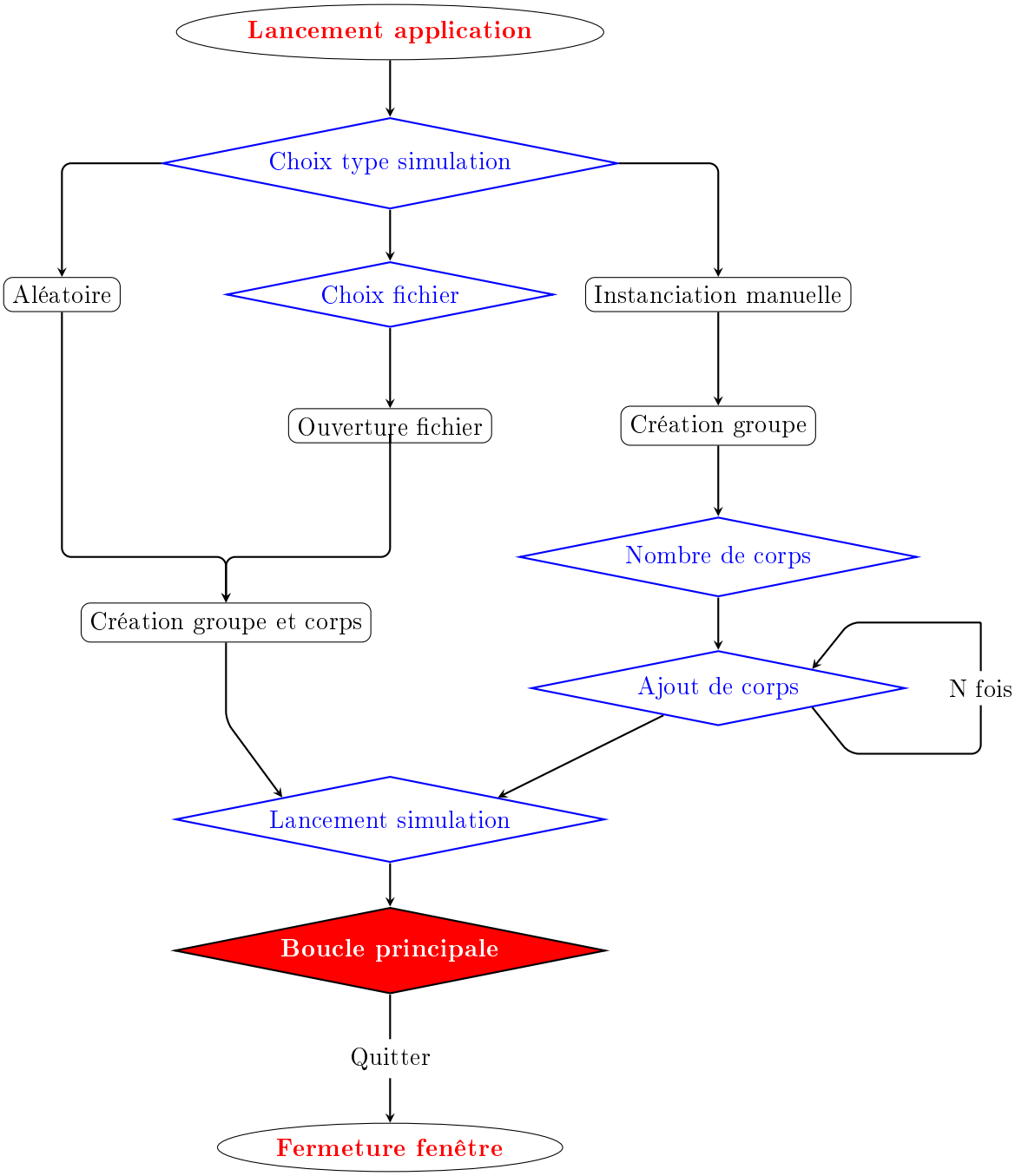

FIGURE 10 – Ordre d'exécution de notre programme

veut plus en créer. Il peut choisir aussi de sauvegarder dans un fichier pour reprendre plus tard la création de sa simulation.

Lorsque la simulation commence, nous ne pouvons pas l'arrêter à moins de mettre le temps à 0. Pour arrêter la simulation, nous avons juste simplement besoin de fermer la fenêtre ou appuyer sur le bouton "Quitter".

5.2 Premier algorithme implémenté (version naïve)

Cet algorithme a été implémenté en premier. C'est l'algorithme naïf d'une simulation à N corps. Il ne décrit qu'un seul tour de simulation (chaque corps ne bouge qu'une seule fois).

Algorithm 1: simulateAllBodies

Input: *listeCorps* : liste des corps de la simulation
Δt : temps relatif écoulé lors de chaque tour

for $o \in listeCorps$ **do**
  $o.acceleration \leftarrow Vector3D()$
  for $o2 \in listeCorps$ **do**
    if $o != o2$ **then**
      $o.acceleration \leftarrow o.acceleration + \frac{i.mass \times (i.position - o.position)}{\|i.position - o.position\|^3}$
    end
  end
  $o.acceleration \leftarrow -o.acceleration \times G$
  $o.velocity \leftarrow o.velocity + o.acceleration \times \Delta t$
  $o.position \leftarrow o.position + o.velocity \times \Delta t + 0.5 \times o.acceleration \times \Delta t^2$
end
return *null*

Comme nous pouvons le voir, cet algorithme applique seulement les équations que nous avons vu à la section 2. Cet algorithme est néanmoins plus évolué que certains autres algorithmes car des simplifications de calculs ont été réalisées. Nous utilisons l'équation 2 au lieu de l'équation 1. Les simplifications permettent de réaliser 20% [2] de calcul en moins ce qui permet d'accélérer l'exécution de notre programme. Nous avons préféré utiliser cette équation pour des soucis de rapidité/optimisation des calculs mais nous aurions tout aussi pu utiliser la loi universelle de la gravitation et la deuxième loi de mouvement de Newton. Cela aurait permis de faire moins de redondance dans notre code !

Nous pouvons considérer que le simulateur est prêt. Mais le nombre de calcul à réaliser reste encore très important ! En effet, la complexité en temps de cet algorithme est quadratique ($O(n^2)$), ce qui n'est pas très bon en terme d'optimisation ! Nous avons donc besoin d'un algorithme qui soit moins gourmand en calcul et donc en temps d'exécution.

2. Ce chiffre résulte d'une comparaison du nombre d'opérations effectuées à chaque tour.

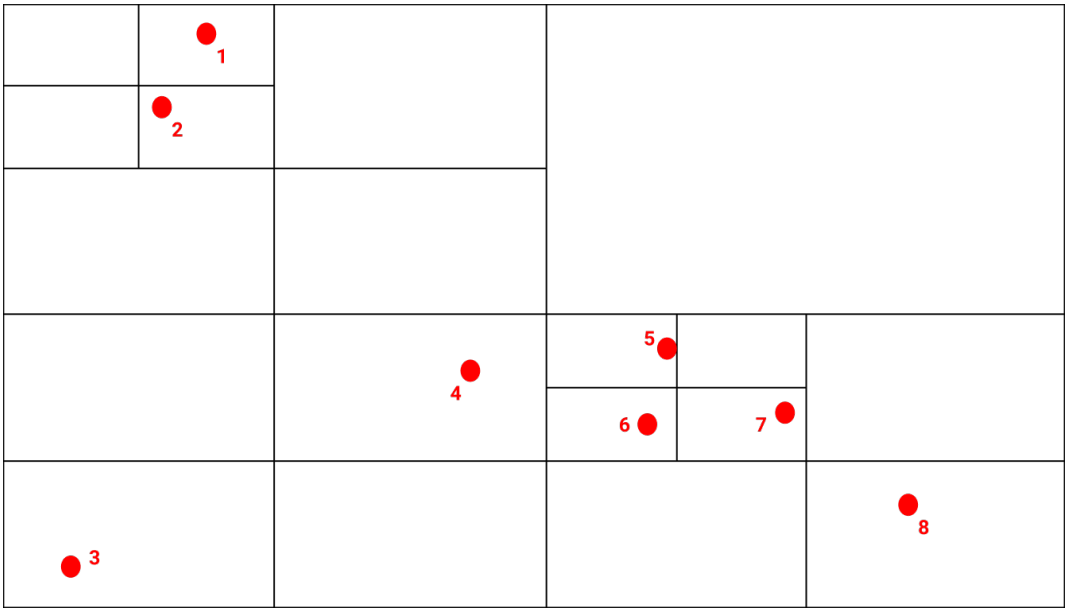

FIGURE 11 – Exemple de découpement récursif d'un espace 2D

5.3 L'algorithme de Barnes-Hut

5.3.1 Principe et exemple

Josh Barnes et Piet Hut ont créé un algorithme qui permet de simuler N corps de manière plus rapide que l'algorithme naïf. Pour cela, ils utilisent la spécificité des QuadTree (ou OcTree en 3D) qui permet de découper récursivement l'espace en carré (resp. cube).

On utilise donc des arbres à 4 branches (resp. 8). L'utilisation des arbres explique la rapidité de l'algorithme. En effet, la complexité en temps de cet algorithme est quasi-linéaire ($O(n\ log\, n)$) ce qui est meilleur que celui de l'algorithme naïf qui était quadratique ($O(n^2)$). Pour générer l'arbre, on découpe récursivement jusqu'à qu'il ne reste plus qu'un seul ou aucun corps dans le noeud/carré (resp. cube) où l'on se trouve. La figure 12 montre l'arbre généré à partir de l'exemple de la figure 11.

5.3.2 Algorithme d'insertion d'un corps dans l'arbre

L'algorithme 2 montre comment nous avons fait pour insérer un corps dans l'arbre. Les fonctions *boxDivided* et *divideBox* ne sont pas montrées dans l'algorithme. *divideBox* permet de générer les fils du noeud considéré de l'arbre et *boxDivided* permet de vérifier si le noeud considéré contient des fils (ou dire que le noeud considéré est une feuille). Les variables NWB, NWF, SWB, etc sont les fils du noeud considéré (resp. Nord-Ouest Arrière, Nord-Ouest

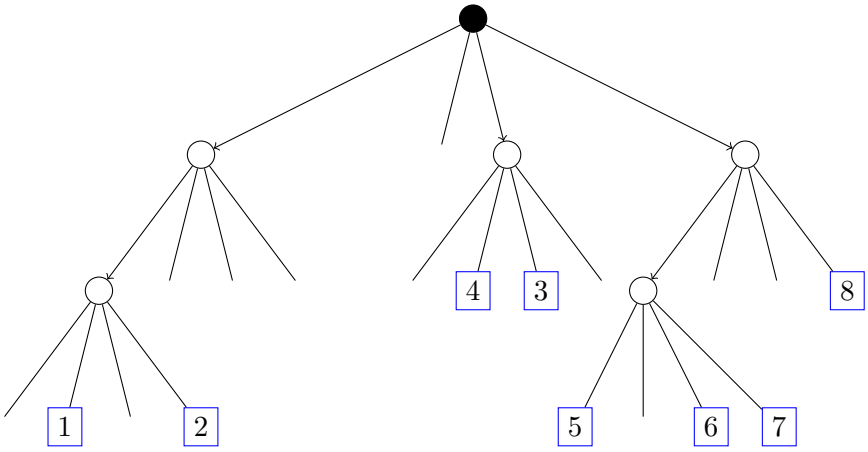

FIGURE 12 – Exemple de découpage récursif (en utilisant l'**algorithme du QuadTree**) généré d'après l'image 11

Avant, Sud-Ouest Arrière, ...).

Dans cet algorithme, on remarque qu'nous mettons le corps à insérer dans le noeud et nous créons un centre de gravité dans le noeud considéré si aucun corps n'est stocké. Cela correspond donc à une feuille qui ne contient aucun corps. Dans tous les autres cas, on se trouve dans le premier niveau du **sinon**. Dans cette partie, si le noeud ne contient pas de fils (c'est donc une feuille avec un corps), nous créons les 8 fils et nous insérons dans le corps stocké dans le noeud, dans la bonne feuille. Ensuite, dans tous les cas de ce niveau, nous recalculons le centre de gravité avec la formule du centre d'inertie

$$\overrightarrow{OG} = \frac{1}{\sum m_i} \sum_{i=1}^{n} m_i \overrightarrow{OM_i}$$

où $\overrightarrow{OG}$ est la position du centre de gravité (ou d'inertie) et $(\overrightarrow{OM_i}, m_i)_{1 \leq i \leq n}$ l'association de la position et de la masse d'un point i. Enfin, nous appelons la fonction récursivement seulement pour le fils qui pourra contenir le corps que l'on insert.

5.3.3 Algorithme de calcul de l'interaction newtonienne dans l'arbre

Après avoir séparé l'espace en arbre, il encore faut ajouter le système d'interaction entre les corps. Tout d'abord, on calcule le centre de gravité de chaque noeud en partant de la racine (cette partie a déjà été réalisée dans l'algorithme 2). Enfin, pour que les corps interagissent réellement, on utilise la formule classique de Newton. Une particularité apparaît, on décide d'utiliser une valeur seuil notée θ permettant de définir si les 2 objets sont éloignés ou pas grâce à la formule $\theta = \frac{s}{d}$ où s est la largeur de la l'espace représenté par le noeud considéré

Algorithm 2: Algorithme d'insertion dans un OcTree

Input: *bodyToInsert* : corps à ajouter dans l'arbre
box : boîte représentant l'espace
body : corps qui se trouve dans le noeud
COM : centre de gravité du noeud (dictionnaire avec 2 clés : *mass* et *position*)

```
1  Function insertion(box, body, COM, bodyToInsert) :
2      if body == null AND NOT boxDivided() then /* Aucun corps dans le noeud */
3          body ← bodyToInsert
4          COM ← [body.mass, body.position]
5      else /* S'il y a déjà un corps à la racine de l'arbre */
6          if NOT boxDivided() then Si le noeud-arbre ne contient pas encore de fils
7              divideBox()
8              if bodyToInsert ∈ box.NWB() then /* Vérifie si la partie Nord-Ouest
                   Arrière pourrait contenir le corps */
9                  insertion(NWB, NWB.body, NWB.COM, bodyToInsert)
10             else if bodyToInsert ∈ box.NWF() then/* Vérifie si la partie Nord-Ouest
                   Avant pourrait contenir le corps */
11                 insertion(NWF, NWF.body, NWF.COM, bodyToInsert)
12             ...
13         end
14         COM ← [COM.mass +
               bodyToInsert.mass, (COM.position×COM.mass)+(bodyToInsert.mass×bodyToInsert.position) / COM.mass+bodyToInsert.mass]

15         if bodyToInsert ∈ box.NWB() then /* Vérifie si la partie Nord-Ouest Arrière
               pourrait contenir le corps */
16             insertion(NWB, NWB.body, NWB.COM, bodyToInsert)
17         else if bodyToInsert ∈ box.NWF() then/* Vérifie si la partie Nord-Ouest
               Avant pourrait contenir le corps */
18             insertion(NWF, NWF.body, NWF.COM, bodyToInsert)
19         ...
20     end
21     return null
```

et d, la distance entre ces 2 objets. Si $\theta < \frac{s}{d}$, alors nous continuons d'explorer l'arbre, sinon, nous considérons le centre de gravité du noeud. Cela permet de ne pas calculer des forces ridiculement faibles qui n'auraient pas d'impacts importants sur l'objet.

L'algorithme 3 montre son implémentation dans notre code. La fonction *boxDivided* n'est pas montrée dans l'algorithme. *boxDivided* permet de vérifier si l'arbre contient des fils. Les variables NWB, NWF, SWB, etc sont les fils du noeud considéré (resp. Nord-Ouest Arrière, Nord-Ouest Avant, Sud-Ouest Arrière, ...). Dans cet algorithme, nous ne faisons rien s'il n'y a

Algorithm 3: Algorithme de mise à jour des forces

Input: *bodyToUpdate* : corps à ajouter dans l'arbre
box : boîte représentant l'espace
body : corps qui se trouve dans le noeud
COM : centre de gravité du noeud (dictionnaire avec 2 clés : *mass* et *position*)
θ : valeur seuil représentant l'éloignement entre 2 objets

```
1  Function updateForce(box, body, COM, bodyToUpdate, θ) :
2      if body == null then /* Aucun corps dans le noeud */
3          if boxDivided() then /* L'arbre a des fils */
4              if body != bodyToUpdate then /* Si les deux corps sont différents */
5                  dimension ← box.getDimensions()
6                  distance ← ||bodyToUpdate.position − body.position||
7                  if dimension/distance ≤ θ then /* Objet trop loin */
8                      bodyToUpdate.force ← bodyToUpdate.force +
                         G (bodyToUpdate.mass × COM.mass × (bodyToUpdate.position − COM.position)) / ||bodyToUpdate.position − COM.position||^3
9                  else /* Objet assez près */
10                     updateForce(NWB, NWB.body, NWB.COM, bodyToInsert, θ)
11                     updateForce(NWF, NWF.body, NWF.COM, bodyToInsert, θ)
12                     ...
13                 end
14             end
15         else /* S'il y a déjà un corps à la racine de l'arbre */
16             if body != bodyToUpdate then /* Si les deux corps sont différents */
17                 bodyToUpdate.force ← bodyToUpdate.force +
                     G (bodyToUpdate.mass × body.mass × (bodyToUpdate.position − body.position)) / ||bodyToUpdate.position − body.position||^3
18             end
19         end
20     end
21     return null
```

aucun corps dans le noeud car le vide ne peut exercer de force sur un objet. Lorsque qu'il y a un corps dans le noeud, nous vérifions tout d'abord si le noeud où l'on se trouve contient des fils. S'il n'en contient pas, nous vérifions juste que le corps du noeud et le corps que l'on calcule la force soient différents et s'ils le sont, on utilise la loi sur la gravitation énoncée par Newton. Si le noeud considéré contient des fils, nous vérifions si les deux corps (celui du noeud et celui que l'on essaie de calculer la force qui est exercée sur lui) sont bien différents. S'ils sont bien différents, nous appliquons la formule du seuil mesurant l'éloignement ($\theta < \frac{s}{d}$). Si $\theta >= \frac{s}{d}$, nous appliquons la loi de la gravitation de Newton sur le centre de gravité du noeud, sinon, nous appelons la fonction `updateForce` récursivement sur tous les fils du noeud considéré.

5.3.4 Algorithme Barnes-Hut final

Pour terminer cette section, l'algorithme 4 nous montre l'implémentation finale avec :
- l'insertion dans l'arbre (l'arbre est supposé déjà construit) ;
- le calcul des forces exercées sur l'objet ;
- le calcul des autres attributs de l'objet (accélérations, vitesses et positions).

Algorithm 4: SIMULATEALLBODIES

Input: *listeCorps* : liste des corps de la simulation
Δt : temps relatif écoulé lors de chaque calcul
tree : racine de l'arbre déjà créé
θ : valeur seuil représentant l'éloignement entre 2 objets

1 **for** $o \in listeCorps$ **do**
2 | `insertion` ($tree$, $null$, $null$, $bodyToInsert$)
3 **end**
4 **for** $o \in listeCorps$ **do**
5 | $o.force \leftarrow Vector3D()$
6 | `updateForce` ($tree$, $tree.body$, $tree.COM$, o, θ)
7 | $o.acceleration \leftarrow \frac{o.force}{o.mass}$
8 | $o.velocity \leftarrow o.velocity + o.acceleration \times \Delta t$
9 | $o.position \leftarrow o.position + o.velocity \times \Delta t + 0.5 \times o.acceleration \times \Delta t^2$
10 **end**
11 **return** *null*

Comme nous pouvons le voir dans cet algorithme, nous faisons une première boucle servant à créer l'entièreté de l'arbre représentant les corps de la simulation en les insérant dans l'arbre (cf. algorithme 2). Nous faisons une seconde boucle qui permet de remettre à zéro les forces de l'ancien tour de simulation, calcule les forces avec l'algorithme 3 et ensuite, nous utilisons la deuxième loi de mouvement de Newton ($\vec{a} = \frac{1}{m}\sum_i \vec{F_i}$) et nous calculons les vitesses et positions de la même façon que l'algorithme 1.

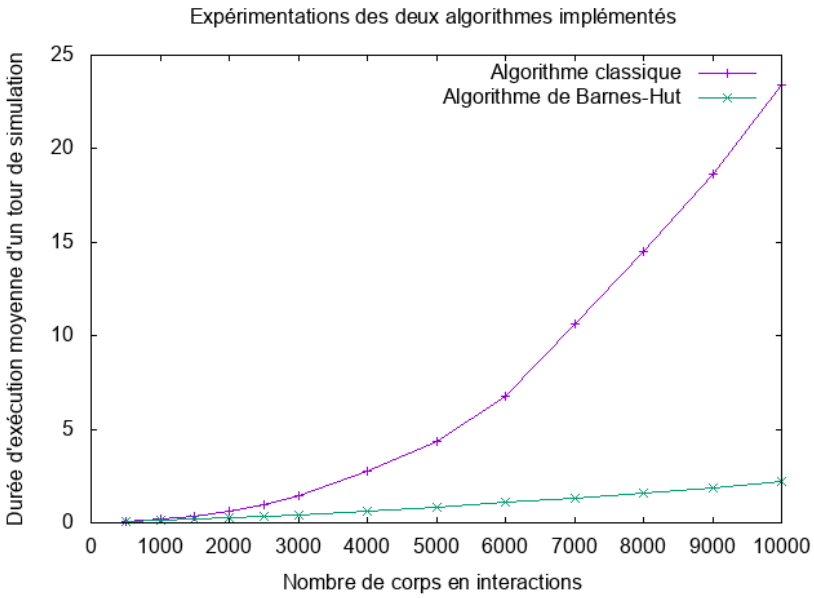

FIGURE 13 – Expérimentations des 2 algorithmes implémentés pour le N corps

5.4 Expérimentations des 2 algorithmes

Comme nous pouvons le voir sur le graphique[3] de la figure 13, on remarque qu'assez tôt, l'algorithme 3 est plus rapide à l'exécution que l'algorithme 1. Cette rapidité d'exécution entraîne néanmoins une baisse de précision. Cette baisse de précision est minime et les valeurs après calculs sont quasiment les mêmes. Nous avons donc décidé d'utiliser l'algorithme de Barnes-Hut à chaque début de simulation (nous pouvons changer d'algorithme à tout moment grâce à un paramètre disponible sur la partie droite de l'interface graphique).

6 Conclusion

6.1 Objectifs à réaliser

Le premier but du projet était de réaliser un simulateur à N corps, dans un espace newtonien avec une interface graphique avant de pouvoir visualiser le résultat des interactions entre les différents corps. Cette partie a été réalisée avec succès. Nous avons bien un nombre N de corps (où N est défini par l'utilisateur) qui interagissent gravitationnellement entre eux grâce aux équations de mouvement de Newton et que nous pouvons observer.

3. Les valeurs ont été mesurées sur 10 tours où nous avons pris la moyenne des tours de simulation effectués.

Ensuite, le second but du sujet était d'ajouter des fonctionnalités diverses parmi quatre propositions :

- instancier des chorégraphies à N corps ;
- accélérer l'optimisation avec un découpage spatial récursif ;
- intégrer un jeu de pilotage au clavier ;
- développer une IA pour optimiser les déplacements avec une trajectoire faible en énergie comme les orbites de transfert.

Pris par le temps, nous n'avons pu développer que la deuxième des propositions avec l'algorithme de Barnes-Hut (expliqué à la section 5.3). Malgré ceci, toutes les fonctionnalités qui ont été implémentées dans notre programme fonctionnent correctement.

6.2 Améliorations possibles

Pour ce projet, de nombreuses améliorations sont possibles. En premier lieu, les autres propositions que nous n'avons pas faîtes auraient été très intéressantes à implémenter et à utiliser dans notre programme.

Dans un deuxième temps, nous aurions pu utiliser des librairies externes qui auraient permis d'effectuer les calculs plus rapidement en utilisant la puissance des calculs vectorielles des CPU et GPU comme Aparapi. Nous avons aussi tenter d'ajouter un système de threading afin que l'interface et la simulation ne tourne pas sur le même thread mais nous n'avons pas pu pleinement l'intégrer sans qu'il n'y est des bugs.

Enfin, avec de meilleures connaissances en mathématiques et en algorithmique, nous aurions pu améliorer les algorithmes de calculs avec le P3M et la méthode multipolaire rapide.

6.3 Ressenti global du projet

Les problèmes de communication ont été la principale source des problèmes recensés au cours de la réalisation du projet. Ces problèmes ont surtout eu lieu entre l'un des membres du groupe et le reste du groupe. Ce membre n'a donc produit que très peu de code. Malgré cela, cela n'a pas entraîné d'énormes retards. Néanmoins, nous n'avons pas pu implémenté tout ce que nous voulions faire.

Nous avons trouvé ce projet très intéressant. Pouvoir simuler des milliers de corps et de les voir s'attirer gravitationnellement, nous avons trouvé cela très beau et donc, cela nous a motivé encore plus afin de produire un bon travail. Ce projet nous a aussi attiré par sa complexité, mais aussi pour le fait de réaliser un projet original qui n'est pas un jeu et qui puisse servir dans d'autres domaines scientifiques.

Références

[1] THF152651. Gravitational N-Body Simulations with JavaFX 3D. Janvier 2018.

[2] Philippos Papaphilippou. Building an N-Body Simulator from scratch. Mars 2016.

[3] Exploring N-Body Algorithms. Janvier 2004.

[4] A collection of animations, sites, pictures, a few papers ... concerning the classical N-body problem.

[5] Communauté Wikipédia. Problème à N corps.

[6] Communauté Wikipédia. Lois du mouvement de Newton.

[7] Communauté Wikipédia. Simulation de Barnes-Hut.

[8] Communauté Wikipédia. Octree.

[9] Communauté Wikipédia. Quadtree.

[10] Communauté Wikipédia. Centre de gravité.

[11] Communauté Wikipédia. Orbite de transfert.

[12] Communauté Wikipédia. Méthode Leapfrog (résolution numérique d'équations différentielles).

[13] OpenJFX. JavaFx javadoc.

[14] Genuine Coder. JavaFX 3D Tutorial for Beginners. 2018.

[15] RosettaCode. N-body problem. Mars 2020.

[16] ArkUmbra. BarnesHutSimulation (in Python). Avril 2019.

[17] lazzati-astro. BarnesHut-Nbody-2D (in Python). Janvier 2017.

[18] Syncleus. Aparapi Getting Started. 2016.

[19] Communauté Wikipédia. Particule-Particule-Particule-Mesh.

[20] Communauté Wikipédia. Méthode multipolaire rapide.

[21] Bibliothèque d'images gratuites. Pexels.com