Projet : Création d’un jeu vidéo 2D Snake
Version du document : 3

Propriétaire du document : Equipe 22

Date de Création : 24/11/2023

Document de Conception de moteurs de Jeux Vidéos.

I. Introduction :

Concevoir les moteurs d'un jeu vidéo équivaut à sculpter l'intelligence même du jeu, une tâche à la fois cruciale et créative. C'est ici que les méthodes agiles, en particulier Scrum, se révèlent indispensables. Scrum offre une flexibilité précieuse pour travailler sur les aspects clés du jeu, tels que les graphismes, les mouvements, et l'interaction utilisateur. Ces moteurs agissent comme les maîtres d'œuvre du jeu, façonnant chaque détail pour créer une expérience immersive et innovante. Son agilité permet également de s'ajuster aux changements tout au long du projet.

Un document de conception des moteurs et du noyau est incontournable pour planifier et détailler l'architecture, les fonctionnalités, et les composants clés du jeu.

Ce présent document détaille par conséquent la conception des moteurs nécessaires pour le développement d’un jeu vidéo. Les moteurs inclus dans cette conception sont le moteur physique, le moteur graphique, le moteur d’input, le moteur de son et le noyau.

II. Architecture Générale :

L'architecture générale d'un jeu vidéo repose sur un ensemble de moteurs essentiels, chacun ayant un rôle spécifique. Le moteur physique gouverne les lois du mouvement et les interactions physiques, assurant un réalisme saisissant. Le moteur graphique façonne l'aspect visuel du jeu, des graphismes aux animations, créant un univers visuel captivant. Le moteur d'Input permet une interaction fluide entre le joueur et le jeu en gérant les commandes. Intégrer le moteur de son à cette structure complexe enrichit davantage l'expérience de jeu en gérant tous les aspects sonores. Ce moteur, dédié à la reproduction des effets sonores et des musiques d'ambiance joue un rôle crucial pour créer une atmosphère immersive. Enfin, le kernel, cœur opérationnel, coordonne l'ensemble, assurant la cohérence globale de l'expérience de jeu. Chaque moteur, spécialisé dans sa fonction, collabore étroitement avec le kernel, créant une architecture qui offre une immersion complète, où chaque composant contribue de manière cruciale à l'expérience globale du joueur.

III. Moteur Physique :

✓ Objectif :

Le moteur physique constitue le fondement du réalisme dans un jeu vidéo. Son objectif global est de créer un environnement de jeu où les objets interagissent physiquement de manière cohérente avec les lois du monde réel.

Sa principale fonction est de simuler les lois de la physique, apportant une dimension tangible aux mouvements des objets. Son rôle ainsi est de calculer les mouvements, les collisions, et les réponses à ces dernières, contribuant ainsi à créer une expérience de jeu immersive et crédible.

En fournissant une simulation physique réaliste, le moteur physique contribue à l'immersion du joueur en rendant les mouvements, les collisions et les interactions visuellement et mécaniquement plausibles.

✓ Architecture :

Le moteur physique constitue le pilier fondamental d'un jeu vidéo, modélisant les mouvements et les interactions entre objets. Il est construit autour d'une architecture modulaire comprenant plusieurs classes, offrant une flexibilité et une extensibilité remarquables. Chaque classe est conçue pour s'intégrer de manière transparente dans le moteur physique, facilitant la maintenance et les futures extensions du système. Ces composants travaillent en tandem pour offrir une simulation physique réaliste, formant ainsi la base solide de l'environnement du jeu.

1) Classe Vector2 :

La classe `Vector2` représente un vecteur bidimensionnel, utilisé pour définir les coordonnées (x, y) de la position, de la vitesse et de l'accélération des objets physiques.

2) Classe PhysicObject :

La classe `PhysicObject` représente les objets physiques dans le jeu. Elle détient des variables telles que la position, la vitesse, l'accélération, la largeur, la hauteur et la masse.

Cette classe offre des méthodes pour mettre à jour la position en fonction de la vitesse et ajuster la vitesse en fonction de l'accélération. Chaque objet physique est également associé à une instance de la classe `Bounds` pour gérer ses limites spatiales.

La classe principale, `PhysicObject`, encapsule les caractéristiques fondamentales des objets physiques tels que la position, la vitesse, et l'accélération. Cette classe expose des méthodes telles que `update()` qui, lors de son appel, ajuste dynamiquement la vitesse et la position en fonction de l'accélération, assurant ainsi un réalisme dans les mouvements des objets.

3) Classe Bounds :

La classe `Bounds` est responsable de la gestion des limites spatiales d'un objet physique.

Elle détermine les limites d'un objet en fonction de sa largeur, de sa hauteur et de sa position, fournit des méthodes pour obtenir les coordonnées des points centraux et des bords de l'objet, ainsi qu'une méthode `updateBounds()` pour actualiser dynamiquement ces limites en fonction des propriétés de l'objet physique.

4) Classe PhysicEngine :

Le moteur physique est géré par la classe `PhysicEngine`, qui maintient une liste d'objets physiques. Cette classe permet la création de nouveaux objets physiques via la méthode `createPhysicObject` et assure la mise à jour de la physique de l'ensemble des objets par l'intermédiaire de la méthode `updateEngine`. Elle illustre l'application du principe de modularité avec une séparation claire des responsabilités.

La classe assure également la gestion des collisions avec les méthodes `checkCollisionSelf` (entre un objet et lui-même) et `checkCollision`, facilitant la détection de collections entre les objets physiques. Elle maintient une collection d'objets physiques à travers la liste `physicObjects`, offrant des opérations telles que la suppression d'un objet physique avec la méthode `removePhysicObject`

Le `PhysicEngine` agit ainsi comme le gestionnaire central de tous les objets physiques du jeu.

✓ Diagramme de classe :

Le diagramme suivant, un diagramme de classe propre au moteur physique, représente l'architecture adoptée pour le mettre en place :

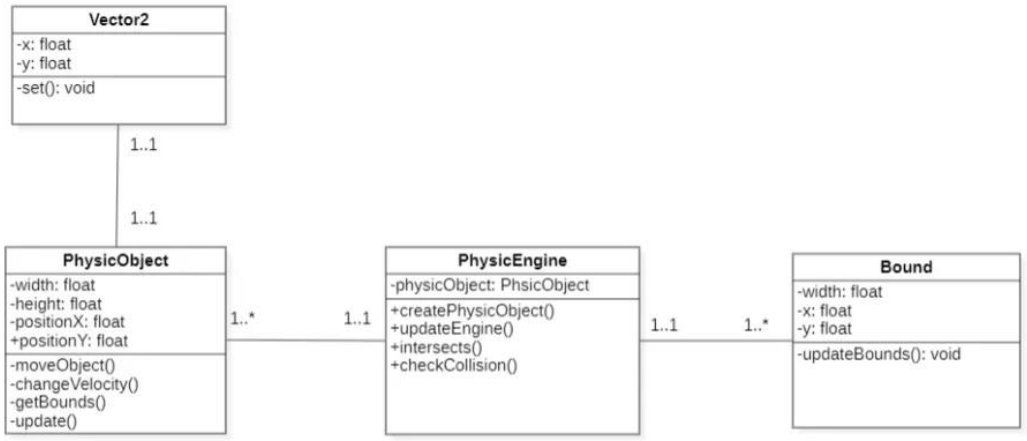

Figure 1: Diagramme de classe - Moteur Physique.

IV. Moteur Graphique :

✓ Objectif :

Le moteur graphique, pilier essentiel d'un jeu vidéo, vise à donner vie à l'esthétique visuelle de l'expérience ludique. Son objectif fondamental consiste à traduire les données du jeu en une représentation visuelle cohérente et attrayante. En d'autres termes, il transforme les informations sur les objets, les décors et les actions du jeu en images affichées à l'écran. À travers l'utilisation de techniques avancées de rendu graphique, ce moteur orchestre l'illumination, les ombres, les textures et les effets spéciaux pour créer un univers visuel captivant. En résumé, le moteur graphique contribue

fondamentalement à l'aspect visuel du jeu, permettant aux joueurs de s'immerger pleinement dans un monde visuellement vivant.

- ✓ **Architecture :**

L'architecture du moteur graphique présenté suit un modèle de conception modulaire basé sur des classes distinctes, chacune ayant une responsabilité spécifique. Il se compose des classes suivantes :

1) Classe Vector2 :

La classe Vector2 joue un rôle fondamental dans la représentation des positions dans l'espace bidimensionnel du moteur graphique. En tant que conteneur pour les coordonnées x et y, elle offre une solution simple et efficace pour manipuler les emplacements des objets graphiques. Cette classe, bien qu'apparemment modeste, fournit une base essentielle pour la localisation précise des éléments visuels au sein du jeu.

2) Classe GraphicObject :

Le cœur visuel du moteur graphique réside dans la classe GraphicObject. Responsable de la représentation graphique d'objets dans le jeu, elle encapsule des attributs tels que la largeur, la hauteur, la position, la couleur, et éventuellement, la texture. L'objet peut être représenté soit par un rectangle coloré, soit par une image chargée à partir d'une texture. La classe permet de mettre à jour la position de l'objet, de créer une représentation visuelle de celui-ci, et de redimensionner cette représentation. En utilisant des composants graphiques de JavaFX, cette classe offre une flexibilité permettant la création d'objets visuels diversifiés, allant des simples formes géométriques aux images texturées plus complexes.

3) Classe GraphicEngine :

La classe GraphicEngine agit comme le gestionnaire central de tous les objets graphiques du jeu. Elle est responsable de leur création, de leur stockage et de leur mise à jour régulière. Cette classe favorise une approche modulaire en permettant d'ajouter, de retirer et de mettre à jour des GraphicObjects de manière dynamique. Grâce à cette organisation, le moteur graphique peut évoluer efficacement avec une variété d'objets tout en maintenant une structure claire et organisée.

4) Classe GameFrame :

La fenêtre de jeu est orchestrée par la classe GameFrame. Celle-ci offre une interface utilisateur graphique via JavaFX, intégrant la scène où les objets graphiques sont affichés. En tant que point d'entrée de l'application, elle crée une fenêtre JavaFX, initialise la scène, et lance une boucle de jeu basée sur le temps pour maintenir un taux d'images par seconde constant. La classe permet également d'ajouter des objets graphiques à la scène. La classe GameFrame agit comme une interface entre le moteur graphique et l'aspect visuel du jeu.

5) Boucle de Jeu :

Au cœur du moteur graphique, la boucle de jeu orchestre le rythme du rafraîchissement visuel. En gérant le temps et en déclenchant la mise à jour régulière du moteur graphique, elle maintient la cohérence temporelle du jeu. La boucle de jeu dans GameFrame s'assure que le moteur graphique est mis à jour régulièrement, fournissant une animation fluide et constante.

Cette architecture favorise une séparation claire des responsabilités, rendant chaque classe indépendante dans son domaine. La classe GameFrame agit comme un point d'entrée de l'application, coordonnant l'initialisation de la fenêtre et le lancement de la boucle de jeu. La boucle de jeu assure

un rafraîchissement régulier du moteur graphique, qui, à son tour, gère la création et la mise à jour des objets graphiques.

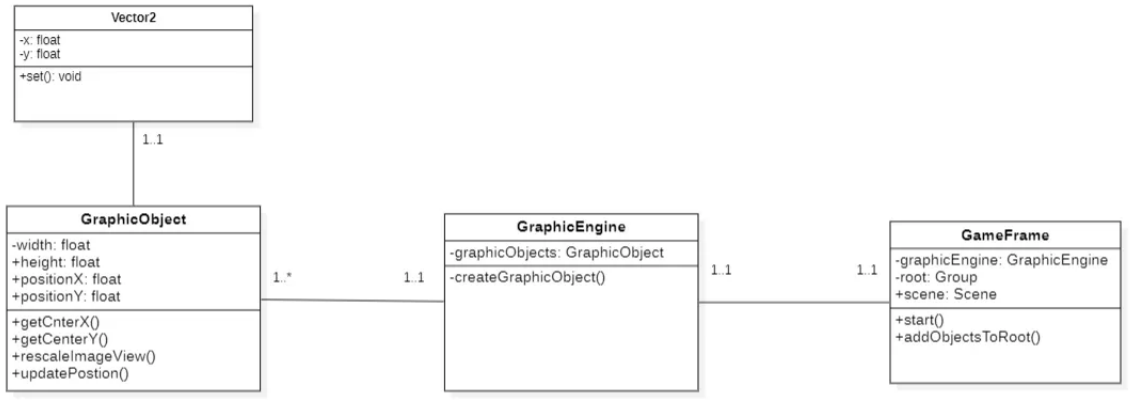

Figure 2: Diagramme de classe - Moteur Graphique.

V. Moteur d'Entrées (Input) :

- ✓ **Objectif :**

Le module d'entrée a pour but de rendre le jeu facile à contrôler. Il capture les touches du clavier, puis les transforme en commandes compréhensibles par le jeu. Il permet au joueur de diriger et d'interagir avec le jeu de manière naturelle, offrant une expérience de jeu agréable et personnalisable.

- ✓ **Architecture :**

L'architecture du module d'entrée repose sur deux classes principales : `InputHandler` et `KeyBinds`.

- La classe `InputHandler` agit comme le gestionnaire central des entrées. Elle utilise une structure de données `HashMap` pour mapper les codes d'événements (les touches du clavier) à des actions spécifiques définies par des objets `Runnable`. Ainsi, chaque événement d'entrée est associé à une action particulière, permettant une gestion flexible et personnalisée des commandes.

- La classe `KeyBinds` étend la classe `KeyEvent` de JavaFX et représente une liaison clé personnalisée. Elle encapsule les informations relatives à un événement clavier, telles que le code de touche et le caractère associé. Cette classe est utilisée pour créer des objets `KeyEvent` personnalisés dans le contexte du jeu.

Cette architecture permet une gestion flexible et réactive des commandes de jeu.

VI. Moteur de Son :

- ✓ **Objectif :**

Le moteur de son vise à gérer la lecture et le contrôle d'effets sonores dans un environnement de jeu. Il est conçu pour être intégré dans un système de jeu afin d'améliorer l'expérience immersive en fournissant une gestion efficace des effets sonores en réponse aux événements du jeu.

✓ **Architecture :**

Le moteur de son `SoundEngine` utilise la bibliothèque Java Sound pour créer et manipuler des clips audio, représentés par des instances de la classe `Clip`. L'architecture modulaire du moteur se compose de différentes fonctions de contrôle telles que `play`, `pause`, `resumeAudio`, `restart`, et `stop`, offrant un contrôle précis sur la lecture des clips sonores. La méthode `load` permet de charger de nouveaux fichiers audio, assurant une polyvalence dans le choix des sources sonores. La réinitialisation du flux audio à l'aide de la méthode `resetAudioStream` facilite le chargement de nouveaux clips sans avoir à recréer l'instance du moteur. Cette conception modulaire favorise l'intégration facile du moteur de son dans divers contextes de jeu, fournissant ainsi une expérience sonore immersive et adaptable.

VII. Noyau (Kernel) :

✓ **Objectif :**

L'objectif du noyau (`Kernel`) dans un jeu vidéo est d'assurer la cohérence et la synchronisation entre les différents moteurs du jeu, notamment le moteur physique, le moteur graphique et le moteur d'entrée. Le noyau agit comme une entité centrale qui orchestre l'ensemble du système, permettant une interaction fluide entre les éléments du jeu.

Le noyau sert de point de rencontre pour les moteurs individuels. Il gère l'initialisation, la mise à jour et la communication entre le moteur physique, le moteur graphique et le moteur d'entrée.

En réceptionnant les événements du moteur d'entrée, le noyau déclenche des actions appropriées dans les autres moteurs. Par exemple, la pression d'une touche peut déclencher un mouvement dans le moteur physique et une réaction visuelle dans le moteur graphique.

✓ **Architecture :**

L'architecture du noyau se compose de plusieurs classes qui interagissent pour assurer le fonctionnement global du jeu.

1) Classe CoreKernel :

Le `CoreKernel` sert de noyau central du jeu, coordonnant les moteurs graphiques et physiques, la gestion des entrées et la boucle principale du jeu.

Le `CoreKernel` lui-même agit comme une entité centrale qui orchestre le fonctionnement global du jeu. Il intègre le `physicEngine` pour gérer la simulation des lois physiques, le `graphicEngine` chargé de représenter visuellement le jeu, et l'`inputHandler` pour capturer les événements du clavier. Le `timeScale` ajustable offre un moyen de réguler la vitesse du jeu, permettant de le mettre en pause ou de le reprendre.

Cette classe constitue le cœur du système de jeu, coordonnant les différentes composantes telles que le moteur physique (`physicEngine`), le moteur graphique (`graphicEngine`), et le gestionnaire

d'entrées (`inputHandler`). Avec la capacité de mettre en pause le jeu grâce à la propriété `timeScale`, le `CoreKernel` offre un contrôle sur la temporalité du jeu (de réguler la vitesse du jeu, permettant de le mettre en pause ou de le reprendre).

La méthode `gameLoop()` représente le moteur du jeu, orchestrant la séquence d'actions à chaque frame. La création d'objets de jeu est simplifiée par les méthodes `createGameObject(...)`, établissant une liaison cohérente entre les composants physique et graphique.

2) Classe GameObject :

Les objets du jeu sont représentés par la classe `GameObject`, qui combine un `PhysicObject` pour la simulation physique et un `GraphicObject` pour la représentation graphique. Cette conception favorise une séparation claire des responsabilités entre les moteurs physique et graphique, facilitant la maintenance et l'extension du système.

La méthode `updateGameObject()` assure la synchronisation entre les aspects physique et graphique de l'objet, contribuant à maintenir une cohérence entre ces deux dimensions essentielles du jeu.

VIII. Couche GamePlay:

✓ **Objectif**

L'objectif général de la couche gameplay est de définir et mettre en œuvre les mécanismes fondamentaux qui créent l'expérience interactive pour les joueurs. Cette couche est au cœur du jeu, déterminant comment les différentes composantes du jeu interagissent entre elles pour offrir une expérience ludique et captivante.

La couche gameplay forme l'épine dorsale du jeu, transformant des éléments statiques tels que les graphismes et les moteurs en une expérience interactive et dynamique.

✓ **Architecture :**

La couche gameplay du jeu Snake, implémentée dans la classe `Main`, démontre une conception organisée et orientée objet. La classe coordonne l'initialisation du noyau via l'objet `coreKernel`, la création du serpent avec des textures spécifiques et une vélocité initiale, la gestion des entrées utilisateur avec des mappages clavier, et le lancement du jeu. Une fonction `update` encapsule la logique de déplacement du serpent, implémentée par le noyau, et la mise à jour de la position de chaque segment de son corps. L'ajout de nouveaux segments par la fonction `addBodySegment` et le positionnement sur une grille avec `moveToGridPosition` contribuent à la mécanique de jeu du serpent. L'ensemble reflète une structuration claire des composants de jeu, assurant une interaction cohérente avec le noyau du jeu `CoreKernel' et une expérience de jeu plaisante. L'utilisation du noyau permet une séparation des préoccupations, favorisant la modularité et la maintenabilité du code. Chaque fonction est définie de manière à encapsuler une partie spécifique du comportement du serpent, illustrant ainsi une approche modulaire et orientée objet pour le développement du jeu Snake.

IX. Diagrammes de conception :

- **Diagramme de packages :**

Le diagramme de packages suivant résume l'architecture suivie :

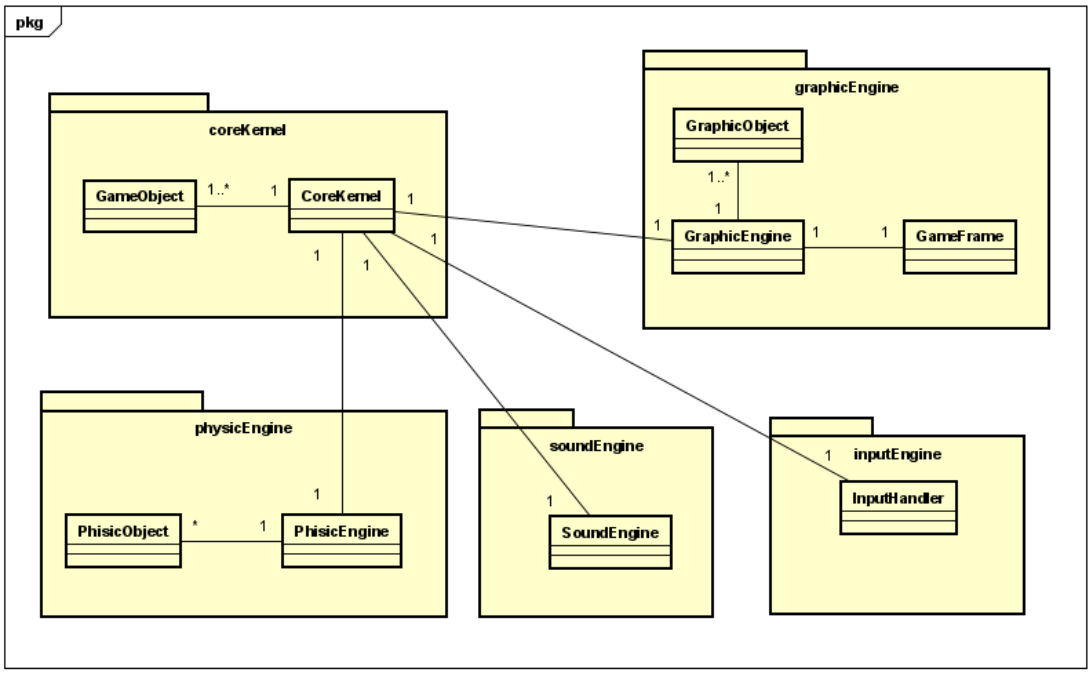

Figure 3: Diagramme de package.

Chaque package représente un moteur, tels que tous les moteurs sont reliés au noyau, et sont indépendants les uns des autres.

- **Diagramme de séquence :**

Le diagramme de séquence suivant illustre le lien entre le noyau et les moteurs : une instanciation du core Kernel mène à une instanciation du moteur graphique, physique et d'entrées. Cette même instanciation du noyau sera appelée dans le programme principale, la couche gameplay (notre main) pour l'exploiter dans le but d'implémenter le jeu.

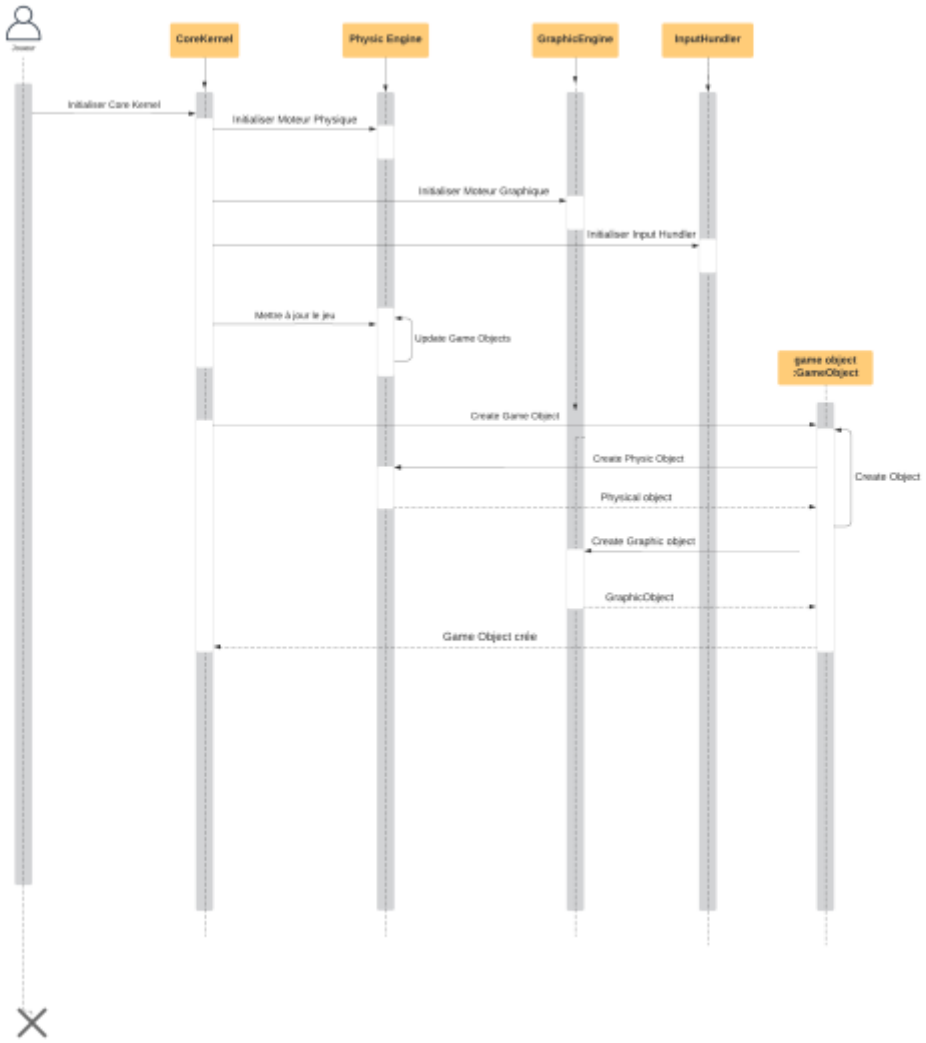

Figure 4: Diagramme de séquence - Lancement du jeu.

- **Diagramme de classe :**

Le diagramme de classe ci-dessous offre une vue synthétique de l'architecture du système de jeu vidéo, détaillant la relation entre les moteurs spécialisés et le noyau central. Chaque moteur, tel que le moteur physique, le moteur graphique et le moteur d'Input, est représenté par une collection de classes dédiées, illustrant ainsi la modularité du système. Ces moteurs interagissent de manière directe avec le noyau, coordonnant leurs activités sans établir de liaisons directes entre eux. Cette approche favorise la séparation des préoccupations et permet une évolutivité du système sans compromettre sa cohérence globale. Chaque classe au sein des moteurs expose des fonctionnalités spécifiques, contribuant ainsi à la richesse fonctionnelle de l'ensemble. La clarté des relations entre les moteurs et le noyau, ainsi que l'absence de connexions directes entre les moteurs, démontrent une conception réfléchie orientée vers la modularité et la maintenabilité du système.

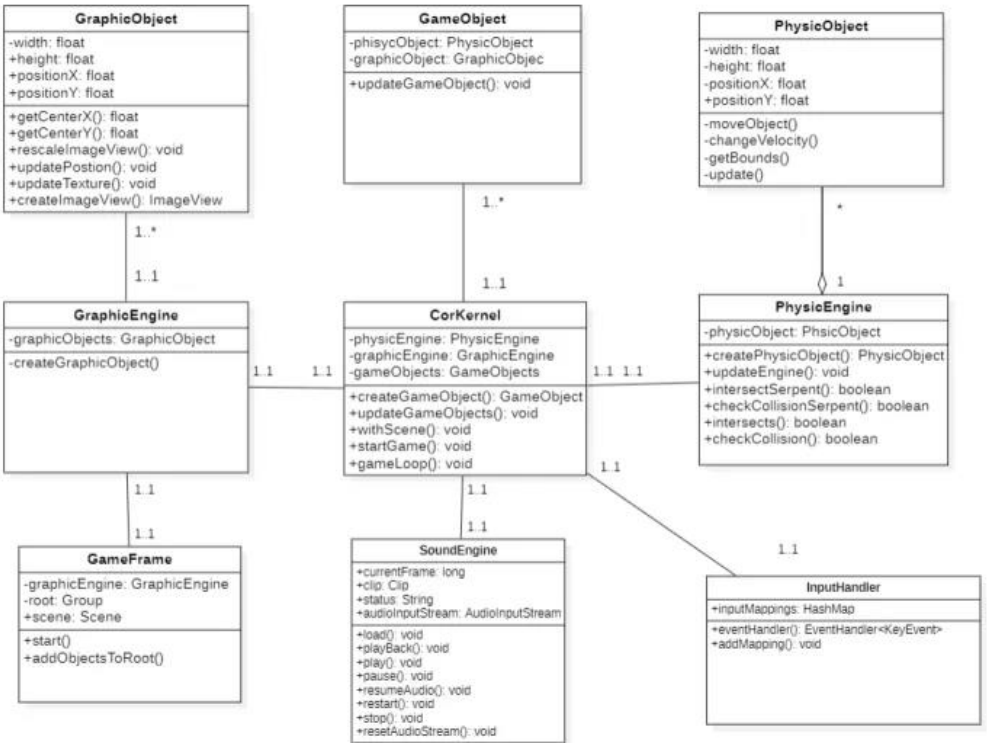

Figure 5: Diagramme de classe globale.

X. Conclusion :

Le jeu Snake présente une architecture soigneusement conçue, divisée en plusieurs couches fonctionnelles, chacune remplissant un rôle spécifique dans l'expérience de jeu. Le noyau du jeu (CoreKernel) agit comme une pièce maîtresse, gérant la physique, les graphiques et les entrées utilisateur. La couche gameplay, illustrée dans la classe `Main`, montre une implémentation élégante du comportement du serpent, avec des mécaniques claires de déplacement, de croissance et de mise à jour. L'utilisation de textures spécifiques et la coordination entre les objets graphiques et physiques contribuent à une représentation visuelle cohérente. L'approche modulaire permet une extensibilité facile du jeu, offrant la possibilité d'ajouter de nouvelles fonctionnalités ou de personnaliser le gameplay. Cette conception réfléchie facilite également la maintenance du code, en assurant une séparation claire des responsabilités entre les différentes couches du jeu. En somme, l'architecture du jeu Snake allie efficacement la simplicité et la flexibilité pour offrir une expérience de jeu captivante.