Mathématiques et Physique pour le jeu vidéo

Construction d'un moteur physique de jeux vidéo

Journal de Développement

Quentin Morel [MORQ22109800]
Gabriel Reboul [REBG22089901]
Clémence Clavel [CLAC30540107]

Phase 1 : Construction d'un moteur élémentaire de gestion de particules

26/09/2022

Difficultés rencontrées

OpenGL

La principal obstacle que nous avons rencontré lors de notre développement est l'utilisation d'OpenGL. Aucun membre de notre groupe n'avait utilisé OpenGL auparavant et nous avons eu des difficultés à le prendre en main, à comprendre sa syntaxe et son utilisation.

Matrices et Projection

Ne venant pas tous de la même filaire, certains membres n'ont pas étudié les domaines de la 3D notamment de la vision 3D avec les matrices de rotation, projection etc…

Architecture

Il nous a été difficile de faire des choix sur l'architecture du projet, sur la façon de l'ordonner, de mettre en relation les différents éléments et sur la façon de les utiliser au sein du projet.

Choix faits et justifications

Visual Studio

Visual Studio est un environnement de Développement en plus d'un compilateur (MSVC) pour le langage C++ . Sa grande popularité ainsi que sa puissance de debuggage (mode pas à pas, etc…), fait qu'il s'agit d'un très bon outil pour développer en C++.

CMake

CMake est un utilitaire pour la compilation de programmes.
Sa plus grande simplicité et sa facile implémentation de code pour du cross-plateforme a fait que notre choix s'est tourné vers ce dernier plutôt que "make".

ImGui

ImGui étant une librairie très optimisée avec une grande flexibilité et facile à implémenter dans un programme Vulkan et OpenGl. Pour notre moteur de physique de particule, il est très important de pouvoir contrôler/modifier des variables rapidement et sans avoir à redémarrer le programme incessamment.

OpenGL

Notre moteur physique devant être un moteur 3D, nous nous sommes orienté vers OpenGL plutôt que Vulkan. OpenGL étant très populaire et très documenté comparé à Vulkan qui lui est sorti en 2016.

GLFW

GLFW est une librairie pour créer des fenêtres, des contextes et des surfaces, et recevoir des entrées et des événements. Afin d'utiliser OpenGL il nous fallait déjà créer une fenêtre qui ensuite sera utilisée par OpenGL. GLFW étant facile d'utilisation et ayant les éléments de bases (fenêtres, entrées clavier, gestion de la souris) pour notre application, nous l'avons donc choisi.

Mathématiques et Physique

Vector3D

Implémentation d'un vecteur de 3 dimensions et des fonctions nécessaires (coordonnées, soustraction ,addition, multiplication par un scalaire, produit scalaire, produit vectoriel, affichage, …) .

Particule

Implémentation d'une particule et des éléments nécessaires : position, vitesse, accélération.

Integrator

Gestion de la mise à jour de la physique du jeu grâce en conservant un taux de rafraîchissement constant. (par exemple changement de la position selon la vitesse de l'élément)

Integrable

Élément qui subit la physique, par exemple Particule hérite d'Integrable.

Gestion du jeu et de la boucle

Physical Engine

La boucle de jeu est localisée dans la class PhysicalEngine qui est la classe mère du projet. C'est elle qui va créer le scène et gérer

Game

Gestion des actions enclenchées dans le jeu selon les entrées du joueur.

InputManager

Gestion des entrées du joueur grâce à des callbacks de GLFW (entrées claviers et mouvements de la souris).

Scene

Classe pour la gestion des GameObjects ainsi que de la caméra du jeu.

Jeu de test

Jeu de déplacement de la sphère visible à l'écran en faisant un appui sur une des touches directionnelles du clavier, la sphère va se déplacer dans la direction de la flèche, à la vitesse choisie dans la fenêtre speed de IMGUI

Phase 2 : Gestion d’objets formés de plusieurs masses par l’ajout de différentes forces et contraintes

23/10/2022

Difficultés rencontrées

Difficulté d’implémentation des forces

Difficulté de comprendre le fonctionnement exact des forces et leur implémentation en code et leur gestion dans la boucle de jeu.
Les forces ayant posé le plus de problème sont celles qui lient des particules directement. Un calcul entre les deux doit être fait pour mettre à jour la résultante de force et ensuite rajouter les forces indépendantes sur chaque particule.

Difficulté d’implémentation des Collisions

Difficulté de comprendre le fonctionnement exact des Collisions, et comment séparer les générateurs de force (système de détections), des systèmes de résolution de force.
L’idée d’utiliser un Registre de collision n’est pas le choix initial, les autres solutions essayées n’ont pas été concluantes (lier les générateur de collision directement aux particules n’était pas très optimisé et ne permettait pas de gérer les collision naïves).

Choix faits et justifications

Générateur de force

C’est la classe mère qui génère les forces sur une particule choisie. Toutes les classes qui héritent de ForceGenerator implémentent une force particulière.
Gravity implémente la gravité.
Spring implémente un ressort entre deux particules.
AnchoredSpring implémente un ressort entre une particule et un point fixe.
Buoyancy implémente la flottabilité d’une particule.
Drag implémente la force de traînée.
L’utilisateur est libre d’ajouter des forces à une particule comme bon lui semble.
À chaque frame, toutes les forces sont appliquées aux particules qui leur sont liées.

Registre de force

L'implémentation du registre de force a été faite à l'intérieur du composant particules. Chaque GameObject ayant un composant Particule peut posséder une liste de force qui va permettre de calculer une force résultant nous permettant d'obtenir la vitesse et accélération du GameObject.
Les forces comme Spring (Ressort) entre deux particules sont quant à elles de la même manière stockées dans un seul Composant Particule, mais la résultante des forces est calculée pour les deux objets. Modifiant la résultante des forces des deux éléments.

Générateur de Collision

C'est la classe mère qui génère les collisions sur 2 ou un groupe de particule chosis. Elle possède la fonction addCollision qui ajoute une collision à traiter s' il y a lieu. Toutes les classes qui héritent de ParticleContactGenerator implémentent un type de collision particulier.
ParticleContactLink abstrait pour tout type de contact qui implique exactement 2 particules
ParticleCable implémente un cable entre deux particules
ParticleRode implémente une tige entre deux particules
ParticleCollide implémente la détection de collision naïve entre tous les **ParticuleCollider** (class qui permet de définir un rayon propre à chaque particule).
L'utilisateur est libre d'ajouter des collider aux particule comme bon lui semble.
À chaque frame, toutes les forces sont appliquées aux particules qui leur sont liées.

Registre de Collision

L'implémentation du registre de force a été faite à l'intérieur de la scène car c'est à celle-ci que l'on veut ajouter tout ce qui permet de faire des collisions. Le registre de collision contient l'intégralité des générateurs de collision. L'objectif du registre de collision est de générer à chaque frame l'intégralité des collisions qui vont se déclencher.

Résolution de Collision

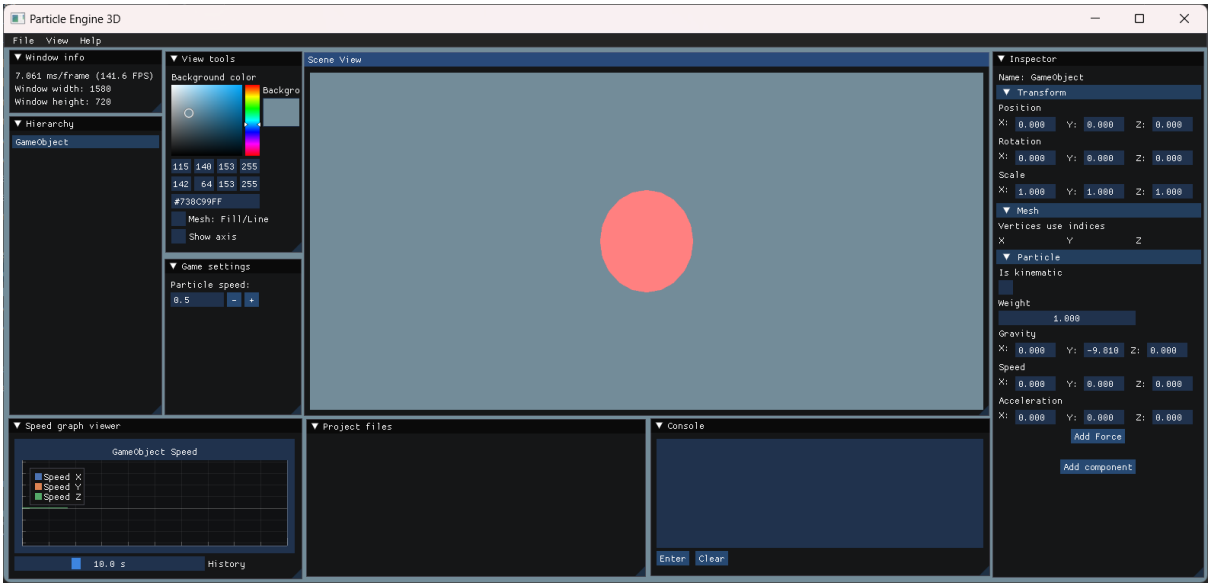
Le système de résolution de collision permet de mettre à jour toutes les particules qui sont censées subir une collision, il le fait dans un ordre précis pour limiter la propagation cyclique des collisions.

A chaque frame on génère l'intégralité des collision qui vont devoir être traité grâce au registre de collision, puis on les résout grâce à **ParticleContactResolve.**

Fonctionnalités supplémentaires

Interface Utilisateur

Afin de faciliter le débogage et l'amélioration du projet beaucoup d'interface ont été ajoutées au projet, notamment avec ImGui. La scène est maintenant visible dans une fenêtre et différents contrôles ont été ajoutés pour simplifier sa modification.



L'état de la vitesse de l'objet pouvant être visualisé sur le graphe.

Ajout d'un système orienté composants

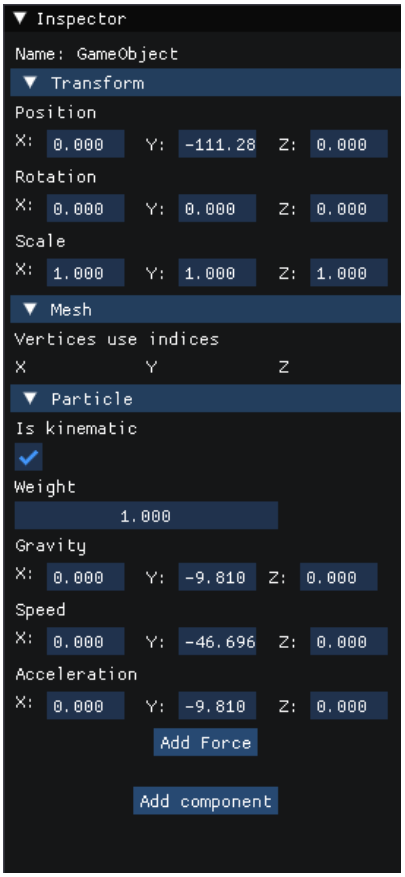
Notre projet doit être assez flexible afin de permettre des modifications des plus aisées, de plus notre projet ne sera pas forcément composé que de particules, nous aurons aussi des

Rigidbody et autres. C’est pourquoi son utilisation est des plus importante afin de pouvoir créer le moteur le plus polyvalent possible, une utilisation orientée composant nous permettant même en débuggage d’ajouter des composants à un GameObject sans avoir à changer le code une fois implémenté.

Ajout des forces dans le composant Particle

Afin de gérer nos particules, un composant a été créé à cet effet. Un composant “particule” auquel on peut ajouter des forces et de mettre à jour son état/position.
A chaque mise à jour du moteur, tous les composants de tous les gameobjects se mettent à jour, mettant à jour ainsi le composant “particule”.

Afin de pouvoir déboguer facilement, l'inspecteur est mis en place pour le moteur. Avec la possibilité de changer l’état du composant, avec l’accélération, la vitesse, la gravité, etc…



Jeu de test

On peut voir à l'écran un amas de 50 particules. Toutes sont soumises à un ressort lié à un point fixe en (0,0,0) et elles subissent des collisions entre elles. En utilisant les flèches directionnelles l'utilisateur peut déplacer le point fixe et observer la conséquence sur l'amas de particules. A l'aide de la souris, l'utilisateur peut également manipuler la caméra notamment la translater ou zoomer.
L'utilisateur peut sélectionner chacune des particules et changer sa position, sa vitesse mais également lui ajouter des forces qui s'ajoutent à sa force list. A partir de cette force list, l'utilisateur peut manipuler ses forces et en modifier les composants (exemple : modifier la constante de gravitation g pour la gravité). Il peut également supprimer une force qui s'applique à une particule en sélectionnant la particule.

Phase 3: Gestion des corps rigides par l’ajout de la physique rotationnelle

21/11/2022

Fonctionnalités supplémentaires

Quaternion

Ajout d’une classe quaternion qui permet de faire des calcules spécifiques aux quaternions. Notamment sa construction à partir d’un vecteur, la rotation d’un quaternion, la mise à jour avec une vitesse angulaire…

Matrix33

Ajout d’une classe Matrix33 qui permet de faire des calcules matriciel pour des matrices de taille 3*3. Elle permet également de générer une matrice de rotation à partir des quaternion et d’effectuer la rotation de vecteurs.

Matrix34

Ajout d’une matrice 34 qui permet en plus de la Matrice33 de gérer la translation.

Matrix44

Ajout d’une classe Matrice 44 qui sert uniquement à effectuer des multiplication de Matrice34 en Utilisant la matrice de transformation affine.

Test unitaires des outils mathématiques

Ajout de tests unitaires pour les outils mathématiques. Ces tests sont essentiels car cette partie est toujours la plus compliquée à debug. Ces tests ont d’ailleurs permis de déceler bon nombre de bugs.

PhysicalComponent

Rigidbody

Les objets de type Rigidbody subissent la physique rotationnelle. A chaque frame, la somme des moments des forces qui lui sont appliquées ainsi que sa matrice d'inertie sont recalculées afin de soumettre l'objet à la deuxième loi de Newton pour la rotation, mettant ainsi à jour son accélération et sa vitesse angulaire.

Difficultés rencontrées

Test unitaires des outils mathématique

La problématique la plus grande sur la création des test est de pouvoir faire des assertion exact malgré le fait que l'ordinateur effectue des approximations. Il y a donc eu un travail de recherche dans les conditions initiales pour s'assurer que l'ordinateur envoie toujours des valeurs exactes dans les assertions des tests.

Problème lors de du calcul des forces angulaire

La problématique était de pouvoir calculer les forces angulaires avec les forces generator créer pour la particule sans modifier leur comportement avec les particules. C'est pourquoi nous avons rajouté une fonction dans ForceGenerator qui renvoie le vecteur force après calculs de celui-ci. Le rigidbody peut ainsi récupérer cette intensité et l'appliquer au point désiré.