Document de conception moteur physique : Forces et contacts

Groupe C

Réalisé par :

Antoine De Saint Martin DESA09050103

Victor Guiraud GUIV09029901

Axel Guza GUZA16069906

Table des matières

Contexte : Dans le cadre du cours « 8INF935 – Mathématique et Physique pour le jeu vidéo » nous devons réaliser un moteur physique.

La conception

Durant la conception nous avons pris de l'inspiration d'informations du cours, de notre professeur et de nos connaissances afin concevoir nos algorithmes. Nous avons essayé de nous éloigner davantage de l'algorithmie présenté dans le livre « I. Millington. Game physics engine development», et avons fait de notre mieux pour retranscrire les mathématiques sous forme de code C++.

Forces

Afin d'initialiser l'implémentation de nos classes de forces, nous avons suivi les suggestions de notre cours sur l'ajout de forces. Nous avons implémenté de la gravité, un effet de traînée selon la forme de la particule, de la flottabilité (buoyancy), un ressort suivant la loi de Hooke, ainsi que des générateurs et des registre de forces.

Ces forces vont nous permettre d'augmenter le degré de réalisme de la simulation de notre moteur.

La gravité nous sert simplement à appliquer un effet de gravité (terrestre ou non) à notre particule.
Le phénomène de traînée simplifié va faire opposition au mouvement en fonction de coefficients de traînée.
Des ressorts qui vont simuler des corps déformables et qui seront utiles pour mettre en place le système flottabilité.
Cette flottabilité va nous permettre de simuler de l'interaction entre un fluide et un corps.

Contacts/collisions

De même qu'avec nos classes de forces, nous avons suivi les suggestions de notre cours sur la gestion des contacts. Nous avons implémenté un système de contact entre des particules, un système de résolution de collision entre ces particules, de

génération de contact, des liens de “pseudo contact”, ainsi qu' un câble et une tige de contact.

World

Le World est une classe qui va gérer toute la physique du monde. Il va permettre la mise à jour du temps d'image, l'instanciation des particules, des forces et des contacts etc.

Il nous permet de créer les collisions entre les particules en fonction du type de contact qu’y a lieu et aussi de demander le rendu des particules et des murs.

Wall

Le Wall va nous permettre de simuler un mur ou les particules vont avoir à entrer en contact et cela va alors les arrêter.Ce mur va également pouvoir servir de sol mais aussi un plafond.

Dans l'implémentation du wall nous avons une fonctionnalité qui permet de le définir via 2 points le plan du mur et ainsi une droite constituée d’un ensemble de Vector3D qui vont permettre de détecter les particules en collision avec l’un d’entre eux. Nous testons la distance séparant une particule du sommet A et du sommet B, si la somme des deux est légèrement supérieure à la distance AB, nous allons avoir possiblement un contact.

Nous avons implémenté un système de collision qui permet de savoir quand la particule est trop rapide et qu'elle pourrait rentrer dans le mur (interpénétration) afin de la figée dans le mur. Et si le mur n'est pas droit, la vélocité est modifiée afin qu' il se déplace sur l’axe des x comme s’il glissait.

Main

Le Main.cpp est la classe principale qui permet de faire tourner tout le programme. Dans cette classe nous allons trouver différents algorithmes et blocs de tests.
Initialement nous nous étions beaucoup inspirés de la programmation I. Millington afin de gérer le temps et nous avions donc implémenter une classe de “Timing” pour gérer cela. Sous les conseils de notre professeur nous avons donc employé la librairie standard C++ *chrono* afin de gérer ce timing.

Nous avons utilisé notre main de l’étape 1 contenant déjà le code OpenGL et Imgui nécessaire afin d’initialiser une fenêtre de visualisation de notre engine.

Dans un premier temps nous avons implémenter des variables propres aux différents éléments que nous avons rajoutés durant cette étapes (position de mur, volume de flottabilité , coefficients de traînée, gravité…)
Puis nous avons fait en sorte de pouvoir manipuler ces variables en direct à l’aide de l’interface Imgui.
Nous avons également mis des variables permettant de générer des particules, mur, générateur de force et contact dans l’engine et même des fonctions d’instanciation en direct une fois la solution lancée. Pour ce qui est des particules et des murs, ils peuvent être instanciés, les contacts et générateurs de forces possèdent encore des erreurs et des intéruptions surviennent.

Nous avons créé la fonction *TestEachFunctionnalityMethod()* afin d'effectuer des tests des différentes fonctionnalités que nous avons ajoutées.

Pour notre jeu d’illustration des systèmes implémentés, nous avons positionné un amas de particules dans notre repère orthonormé où nous avons instancié une particule, des forces, un mur et des contacts. Une fois la simulation démarrée, les particules vont être "propulsées" selon la physique qui leur sont appliquées et se rencontrent entre elles.

Finalement nous avons ajouté des librairies OpenGl permettant de faire fonctionner la solution sur n’importe quel ordinateur.

Les difficultés rencontrées

Nous avons tous rencontré des erreurs ou des problèmes à différents niveaux.

Pour plusieurs algorithmes des interrogations ont subsistées :
-L'algorithme de résolution des contacts entre particules, nous avons une liste de contacts que notre physicWorld contient, cependant, comment définir que tel ou tel contact est plus important à résoudre ? J'avais effectué le code pour, les vélocités relatives étaient récupérées et triées dans l'ordre croissant (dans le fichier Precision.h plusieurs méthodes sont implémenté pour le tri), or je ne voyais finalement pas en quoi la détection des contacts se basait sur la différence de vélocité. Ainsi, je prends chacun des contacts et vérifie leur interpénétration directement pour savoir si nous avons contact ou non.
-Pour ce qui est des câbles et des tiges, si notre code doit être générique avec ContactResolver, je ne voyais pas comment différencier un contact classique et un contact spécial. Ainsi pour les câbles et tiges, la pénétration du contact est différente de 0 (et la restitution == 1 pour les tiges), je sais donc si je suis face à un simple contact ou si je dois réagir en conséquence d'un cable/tige avec des conditions.
-J'ai remarqué une baisse de FPS sur le programme par rapport au rendu 1, je n'ai pas pris le temps d'y mettre le nez, j'aurai pu essayer de voir avec un profiler quelle partie du code prenait trop de mémoire ?
-Via la méthode choisie pour la résolution des contacts avec les murs, difficultés à paramétrer les valeurs fixes du code. (à quel distance mur/particule je dois déclencher la collision, à quelle distance je dois déplacer la particule dans le sens du vecteur résultant pour éviter une seconde collision directement générée…)
-La compréhension du fonctionnement attendu du câble/du ressort également, je pense que nous avons des résultats cohérents visuellement, cependant il manque la certitude du fonctionnement attendu.

Axel se posait initialement beaucoup de questions sur comment nous devons implémenter nos programmes de résolution de collision et de "world". Nous nous posions la question de comment implémenter un sol afin d'éviter le problème de checking des éléments sur se dernier.

Victor à du mal à traduire les mathématiques de contact sous formes de C++. Puis d'appliquer notre physique à OpenGL afin d'appliquer tous les effets aux particules.
Il a fait des tentatives de test de compilation de notre solution sur des ordinateurs ne disposant pas de nos librairies afin de la fonctionner sur n'importe quelle machine .
Il à ensuite essayé de réfléchir à la conception d'un très petit jeu qui pourrait illustrer la physique de notre solution.

Source :

- https://github.com/ocornut/imgui
- https://www.opengl.org/
- https://moodle.uqac.ca/mod/url/view.php?id=653865
-

Les librairies :

-https://www.sfml-dev.org/ : SFML librairie actuellement utilisée.

-https://www.opengl.org/resources/libraries/glut/glut_downloads.php : librairie GLUT appairée à OpenGL.

-https://www.just.edu.jo/~yaser/courses/cs480/opengl/Using%20OpenGL%20and%20GLUT%20in%20Visual%20Studio.htm : utilisation de GLUT et OpenGL sur Visual Studio.

-https://www.dll-files.com/glut32.dll.html : récupération .DLL GLUT.

Les ressources de répertoires GIT et livre :

-https://github.com/idmillington/cyclone-physics/blob/master/src/demos/main.cpp : répertoire GIT du livre associé au cours, Millington. Game physics engine development.

-https://ebookcentral.proquest.com/lib/uqac-ebooks/home.action : livre de référence du cours.

Aides sur les librairies :

-https://learnopengl.com/Getting-started/Hello-Triangle : commencement sur OpenGL.

-https://dokipen.com/modern-opengl-part-3-hello-triangle/ : commencement sur OpenGL.

https://opengl.developpez.com/tutoriels/apprendre-opengl/ : implémentation supplémentaire opengl

-https://en.cppreference.com/w/cpp/header : librairies de bases c++.

-https://www.developpez.net/forums/d1108488/applications/developpement-2d-3d-jeux/api-graphiques/opengl/dessiner-repere-xyz/ : renseignements élaboration repère avec OpenGL.

-https://www.opengl.org/resources/libraries/glut/spec3/node14.html : renseignement sur GLUT.

-https://cpp.hotexamples.com/fr/examples/-/-/GET_XBUTTON_WPARAM/cpp-get_xb-utton_wparam-function-examples.html : récupération interraction souris.

-https://learn.microsoft.com/en-us/windows/win32/learnwin32/keyboard-input : récupération interraction clavier.

-https://en.cppreference.com/w/cpp/chrono/system_clock : gestion du temps en c++ avec l'entête chrono.

-https://community.khronos.org/t/glcolor3f-not-working/69751/2 : coloration des objets avec OpenGL.