8INF935 - Mathématiques et physique pour le jeu vidéo

Phase 2 - Journal de bord

Jules RAMOS (RAMJ27020100) - Mikhael ChOURA (CHOM23060100) - Gaëlle THIBAUDAT (THIG24539900) - Béatrice GARCIA CEGARRA (GARB19510105)

Table des matières

Introduction

L'objectif de cette deuxième phase de développement était de poursuivre le développement du moteur physique en ajoutant le système de gestion de forces, de collisions et d'impulsions ainsi que la manipulation d'un amas de particules implémentant l'ensemble de ces systèmes.

I. La logique du GameWorld et du registre

Afin de calculer la résultante des forces appliquées à chaque particule pour chaque frame et de pouvoir utiliser l'intégration d'Euler pour calculer la nouvelle vélocité et position de la particule comme en phase 1, une logique présentée en cours a été implémentée. Dans le gameworld, l'ensemble des particules à considérer, les forces fixes telles que la gravité et la friction de l'air, et l'ensemble des ressorts, câbles et tiges de la scène sont stockés dans des vectors de la STL. L'update du fichier ofApp appelle l'update du gameworld, qui, à chaque génération de force découlant des forces précédemment citées, va associer les particules concernées et stocker la paire dans la structure ParticuleForceRegistration du ParticuleForceRegistry. Une fois toutes les associations réalisées, l'update de ce registre est appelée. Chaque force implémentant une méthode updateForce du fait de l'héritage de l'interface ParticuleForceGenerator, cette méthode est appelée pour calculer la force générée sur les particules en jeu. Ces forces résultantes sont stockées dans un attribut de la classe particule, AccumForce. Une fois les updates finies, on appelle l'intégrateur de chaque particule qui utilise AccumForce comme accélération de la particule pour la frame, puis on vide Accumforce. Le registre est également vidé, permettant de refaire un calcul des forces à la frame suivante.

A noter : Pour cette phase, l'intégration de Verlet à été abandonnée car elle ne considère que la position et d=ne permet donc pas l'implémentation d'une système de gestion par le biais de la force résultante donc du calcul de la vitesse d'une particule pour en déduire la position.

II. Les collisions

Pour la gestion des collisions, une double boucle for, avec une complexité de $O(n^2)$, vérifie si la distance entre les centres de deux sphères est inférieure à la somme de leur rayon. Si c'est le cas, une impulsion de collision est générée. De même, pour les collisions avec le sol, la coordonnée en y de la sphère à laquelle s'ajoute son rayon est vérifiée pour générer une impulsion. Les impulsions

diffèrent des forces telles que la gravité car, n'étant générée que ponctuellement sur une frame, on ajoute à la vélocité de la particule la vélocité créée par l'impulsion.

Afin d'éviter les oscillations dans le sol et d'avoir des particules au repos, on nullifie la vélocité de la particule en direction du sol en ajoutant à sa vélocité l'opposé de sa projection sur le vecteur normal au sol. Pour déterminer l'état de repos de la sphère, la vérification initiale était si $g \times \delta t > v$, avec g due à l'application de g sur une frame. Nous nous sommes cependant rendus compte que cette condition n'était jamais respectée et donc que l'état au repos n'était jamais atteint. En augmentant $g \times \delta t$ par une quantité arbitraire, que nous avons laissée à 12.0f, et donc en augmentant le seuil de vitesse auquel l'objet est considéré au repos, les résultats étaient plus cohérents et réalistes. Nos impulsions "minimales" pour les particules manipulées étaient donc trop grandes et le seuil a dû être changé pour assurer un résultat cohérent.

L'application physique de la collision, telle qu'elle a été introduite en cours, est une impulsion soit l'ajout d'une vélocité proportionnelle à la vélocité et la masse ainsi que d'une position pour séparer les deux éléments entrant en collision. Cet ajout manuelle d'une position à un objet cinétique cause deux problématiques : La téléportation d' objets physiques proportionnellement à leur masse et le contournement des forces et vitesses causant le déplacement Le dernier cas pose une problématique certaine vis à vis des tiges et câbles puisque les forces / vitesses ne sont pas passées d'un objet à l'autre par calcul et voie d'actions réciproques. Autrement dit, à faible vitesse, au repos, la collision cause le déplacement de l'objet concerné sans que l'autre soit affecté.

III. Forces de friction (statiques et dynamiques)

Il a été choisi de repousser l'implémentation de la force de friction statique à une phase ultérieure du projet. La phase actuelle ne concernant que des objets sphériques, la force de friction statique équivaut à l'application d'une rotation sur les objets et non d'une rotation. Ces notions seront vues plus en amont dans les prochaines phases. Aussi, la classe de la force est à leur actuelle codée mais non implémentée dans la simulation physique du monde.

La force de friction dynamique est utilisée pour simuler la friction de l'air sur les objets physiques. La force est une constante de la simulation, égale pour toutes les particules; elle s'oppose directement et en toutes circonstances à la direction de la particule. C'est à partir de ce moment que sont introduites les opérations mathématiques de projection et de normalisation des vecteurs. Contrairement au cas de la friction statique, le vecteur normal de direction de la force peut être calculé à partir du vecteur orthonormé du mouvement. La vitesse étant la tangente à la trajectoire de la particule à l'instant du calcul, elle est utilisée comme référence pour le sens et la direction du mouvement mais normalisée puisqu'elle n'a pas d'impact sur la force de friction. Les coefficient K1 et K2 sont fixés dans le GameWorld.cpp, au niveau des instanciations. K1 est plus grand que K2 à un facteur 100 pour limiter l'amortissement des mouvements.

IV. Gestion des câbles et des tiges

Initialement les tiges devaient être gérées, comme évoqué en cours, sous forme de collisions. Néanmoins, les particules au repos, c'est-à-dire seulement soumises aux forces de gravité, ne possèdent pas de forces résultantes pas de forces résultantes donc pas de force à laquelle opposer la force de la tige permettant donc de calculer sa norme ou sa direction. Les vitesses qui ne sont pas créées par l'application d'une force (impulsion câble / collision, vitesse initiale des particules) ne s'appliquent pas à la particule qui n'est pas concernée. Aussi, pour pallier ces problématiques, il a été choisi d'implémenter la force des tiges sous la forme d'impulsions. Il est à noter que la téléportation causée par une collision crée une quantité de déplacement trop importante pour être compensée par un gain de vitesse. Afin de garder un résultat cohérent même dans des cas de grand nombre de collisions, une certaine tolérance a été donnée aux tiges. Ce coefficient arbitraire a été testé et laissé

à un dixième de la longueur de la tige, car ce nombre donnait des résultats satisfaisants de manière analogue au choix de la tolérance des collisions au repos.

Concernant les câbles, une condition à l'application de l'impulsion a été ajoutée. La force n'est appliquée à la particule que si la direction de la particule est dans le sens contraire à la position de la particule reliée. Cette vérification contre en même temps un enchaînement trop court de la position, puisque les particules sont des objets en perpétuel mise à jour de logique et quasi en permanence en mouvement, et une interférence entre le rapprochement causé par l'impulsion et ses conditions de calcul. Une dernière observation importante est l'impact de la simulation des câbles, ressorts et tiges en tant que forces et non en tant qu'objets. Ces derniers peuvent être traversés et "rompus" par des collisions. Ils n'ont en outre pas de collisions propres.

V. Mouvements harmoniques (amortis et non-amortis)

Le calcul des oscillations harmoniques vu en cours concerne un cas particulier des oscillations dans un espace à deux dimensions, où l'origine du repère absolu coïncide avec la position initiale de la particule, dans un contexte temporel continu. Pour créer une oscillation dans un contexte temporel discret (calcul de des durées entre les frames et mise à jour discrète de la position) sans pour autant arrêter la simulation physique du monde où de la particule en oscillations, plusieurs choix d'implémentation ont été faits :

- Initialisation des conditions initiales de position et de vitesse avec les données de la particule à l'instant où l'oscillation débute
- Incrémentation d'un temps local propre à la particule avec les durées des frames, entre le début et la fin de l'oscillation.
- Calcul de la direction de l'oscillation dans le sens opposé et la direction du mouvement
- Ajout de la position calculée à la position courante de la particule, elle-même mise à jour lors de l'intégration d'Euler.

VI. La Caméra et le Player Controller

Nous avons créé une classe Caméra, permettant à l'utilisateur de déplacer sa vision autour d'un point fixe. Cette caméra tourne donc autour d'un cercle grâce aux inputs de souris, nous pouvons également modifier le rayon du cercle pour zoomer/dézoomer la caméra. La difficulté d'implémentation était dans le fait que la caméra doive tout le temps pointer vers le centre du cercle. En effet, il fallait toucher à des notions de Quaternions qui nous étaient encore assez abstraites. Une fonction déjà intégrée dans openFramework nous a permis d'aligner notre Caméra sur le centre du cercle, mais ce serait très instructif de l'implémenter nous-même une fois que nous aurons vu la théorie pour les rotations.

Le Player Controller permet de compléter la caméra en ajoutant des déplacements pour une particule. La caméra est nécessaire pour que le vecteur de direction pointe toujours dans la direction de vision de la caméra. Pour avoir un vecteur de déplacement pour la droite, on utilise un produit vectoriel entre le vecteur forward et un vecteur vertical.

VII. Le Blob

Le blob était une des principales difficultés de la phase 2. L'implémentation de cette fonctionnalité a été réalisée par le biais d'un graphe , composé de sommets (particule) et d'arêtes (ressort). Une particule principale possède un Player Controller, permettant de la déplacer et ainsi de déplacer le blob. Nous avons réussi à généraliser la définition du graphe (donc la création du blob), ce qui n'était pas trivial et a demandé un certain travail. Le blob possède également ses propres

structures de données pour les forces, pour lui permettre d'être le plus modulaire possible, ce qui permet également un meilleur travail d'équipe. Un problème survenu dans cette phase fut un problème purement informatique: les dépendances circulaires. En effet, nous avions besoin d'inclure le Gameworld.h dans le Blob.h et vice-versa, mais nous avions une erreur lorsque nous procédions. Nous avons donc découvert comment faire ce type d'inclusion, en déclarant la classe gameworld dans le header de la classe blob, puis en incluant le header du gameworld, directement dans le fichier d'implémentation des fonctions (blob.cpp).

Nous avons également réussi à implémenter le fait que le blob se sépare en deux blobs de taille à peu près égale, et qu'ils puissent se regrouper également. Cette partie était assez difficile et demandait beaucoupl de débogage pour fonctionner. Nous avons utilisé plusieurs fois le parcours en largeur du graphe, pour par exemple, découper ce graphe en deux parties connexes et disjointes l'une de l'autre. Notre algorithme se rapproche fortement d'un algorithme de coloration de graphe, pour définir deux familles de sommets différents, et de couper les arêtes qui vont d'une famille à l'autre, pour ainsi les séparer complètement.

La jointure de deux blobs se réalise alors en comparant la couleur des deux familles, et leur distance. Si elle n'est pas très grande, le graphe se connecte. L'affichage du nombre de particules de la partie contrôlée par l'utilisateur à été réalisé également en utilisant un parcours de graphe.

Afin que les ressorts reliant les particules du blob puissent se transformer en câble et vice-versa, une structure dans le game world regroupant un ressort et un câble a été implémentée. Nommée, BlobSpring, un check particulier lui est fait dans la mise à jour de la logique: si le ressort est trop étendu et donc les particules éloignées au-dessus d'un certain seuil, un comportement et donc les méthodes des câbles lui est appliqué, sinon c'est le comportement du ressort qui prend le dessus. Cela permet un changement du comportement dynamique et qui ne demande pas de réallocation mémoire, permettant d'optimiser cet aspect du blob au prix d'un vector de la STL supplémentaire.

Enfin, une astuce de programmation pour la création du blob est de créer les particules sur des cercles de manière symétrique. Ainsi, si des cercles non complets sont formés, par exemple en créant un blob de peu de particules, la répartition et donc les liens avec la particule contrôlée seront plus naturels et logiques.