Conception

M2 Image et Multimédia – Chef d'œuvre

Encadré par Charly Mourglia et Valentin Roussellet

Rédigé par : Fabien BOCO, Anselme FRANÇOIS, Dimitri RAGUET

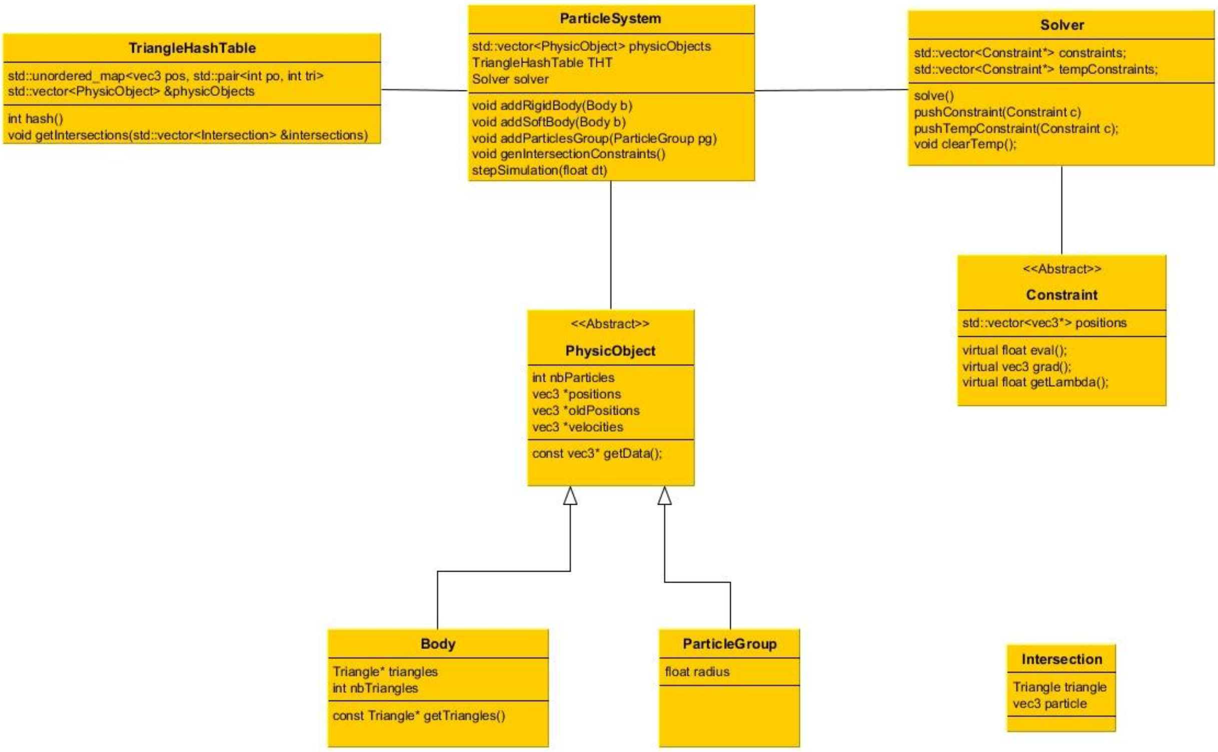
Introduction

Ce rapport présente l’architecture de notre moteur physique implémentant le principe des « Position Based Dynamics » ainsi qu’une révision du planning de développement.

Comme cela a été décrit dans notre précédent rapport, notre programme est censé être intégré en tant que plugin dans le Radium Engine qui nous a été proposé. Mais le moteur physique peut parfaitement être implémenté indépendamment de tout moteur de rendu. C’est donc bien l’architecture du moteur physique qui sera détaillée ici, et non celle du Radium Engine ou encore du plugin permettant d’utiliser notre moteur physique car nous n’avons pas, à ce jour, le recul et la compréhension nécessaires du Radium Engine pour donner l’architecture détaillée du plugin.

Conception

Diagramme



Description des modules

Constraint <<virtual>>

Cette classe permet d’évaluer une contrainte (voir 1er rapport) appliquée à une liste de points (représentée par le champ positions). Elle est composée de trois méthodes :

- virtual float eval() = 0 : Elle renvoie la valeur de la contrainte à minimiser
- vec3 grad(vec3 &pos) : Renvoie la valeur du gradient pour une particule donnée
- float getLambda(vec3 pos) : Renvoie lambda
- std::vector<vec3*> position : Liste de pointeurs vers les particules sur lesquelles la contrainte s’applique

Intersection

Structure contenant les informations d'intersection entre un point et un triangle. Les intersections sont générées par la classe TriangleHashTable.

- Triangle * triangle : triangle avec lequel la collision a été générée
- Particle * particle : la particule

Solver

La classe Solver permet de modifier la position des points contenus dans la contrainte de telle sorte que la valeur renvoyée par la méthode eval() des contraintes soit minimisée au mieux.

- *void solve(int nbIteration)* : Résout le système à l'aide de la méthode de gauss Seidel en effectuant nbIteration
- *void pushConstraint(Constraint c)* : Ajoute une contrainte permanente au système
- *void pushTempConstraint(Constraint c)* : Ajoute une contrainte temporaire au système, typiquement une contrainte d'intersection
- *void clearTemp()* : Supprime les contraintes temporaires du système

La classe stockera deux types de contraintes :

std::vector<Constraint>* constraints : Les contraintes persistantes dans le champ *constraints*.

std::vector<Constraint>tempConstraints* : Les contraintes temporaires dans le champ tempConstraint, typiquement des contraintes d'intersection qui sont censés ne plus exister une fois le système résolu.

ParticleSystem

C'est la classe principale du moteur qui permet de mettre en place la scène et la stocker.

- *void addRigidBody(Body b*) : Permet d'ajouter un objet Body dans la scène. Cette méthode ajoutera par la même occasion les contraintes nécessaires au solveur permettant de simuler un objet rigide (contraintes de distance et d'angle)

- *void addSoftBody(Body b)* : Permet d'ajouter un objet Body dans la scène. Cette méthode ajoutera par la même occasion les contraintes nécessaires au solveur permettant de simuler un objet déformable (contraintes de distance, voire de pression)
- *std::vector<PhysicObject *> physicObjects* : Contient l'ensemble des particules provenant d'un maillage.
- *TriangleHashTable triangles* : Contient les triangles avec lesquels on teste les collisions à chaque tour de boucle
- *Solver solver* : Contient les contraintes de la scène et permet de les optimiser

PhysicObject <<abstract>>

Cette classe a pour principal but de stocker les données sur lesquelles on applique la PBD

- *const vec3 * getData*() : Accesseur permettant de récupérer la position des particules du système après la résolution du PBD
- *int nbParticles* : Nombre de particules
- *vec3 * positions* : Liste contenant la position des particules
- *vec3 * oldPositions* : Liste contenant la position des particules avant la résolution du système, en vue du calcul de la vitesse de ces dernières
- *vec3 * velocities* : Vitesse des particules obtenues pour la prochaine boucle de calcul du PBD

Body

Ensemble des particules et de triangle qui représentent un maillage (rigide ou soft).

- std::vector<Triangle> triangles : Liste de triangles

ParticleGroup

Ensemble de particules représentées par des sphères définie par leur rayon.

- float radius : rayon de la particule

TriangleHashTable

Cette classe permet d'accéder de manière efficace aux triangles présents dans un voxel de la scène. Les principaux avantages de cette structure sont que d’une part les parties vides de la scène ne sont pas stockées en mémoire, et d’autre par l'accès à un voxel est très rapide.

- void getIntersection(vector<Intersection>& intersection) : Effectue les calculs permettant de détecter les collisions et les ajoute dans intersection.
- int hash(vec3) : Permet de générer un identifiant pour stocker les triangles dans la hash table.
- Update() : Met à jour la position des triangles.

Révision du planning

Changements

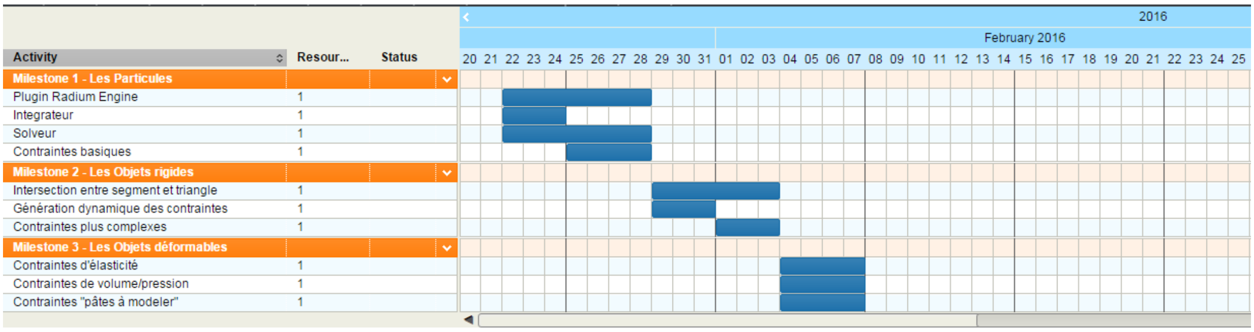
Tout d’abord nous avons dû revoir notre planning du fait que nous sommes désormais 3 au lieu de 4 (donc redistribution des tâches).

Nous avons revu à la baisse nos prévisions d’avancement (Suppression de l’implémentation sur GPU)

On peut désormais plus facilement se répartir individuellement les tâches.

Une nouvelle tâche dédiée à l’implémentation du plugin Radium Engine a vu le jour. On estime cette tâche assez conséquente et critique.

Gantt



Activity	Resour...	Status
Milestone 1 - Les Particules		
Plugin Radium Engine	1	
Integrateur	1	
Solveur	1	
Contraintes basiques	1	
Milestone 2 - Les Objets rigides		
Intersection entre segment et triangle	1	
Génération dynamique des contraintes	1	
Contraintes plus complexes	1	
Milestone 3 - Les Objets déformables		
Contraintes d'élasticité	1	
Contraintes de volume/pression	1	
Contraintes "pâtes à modeler"	1	