DWARF
ANIMATION STUDIO

Rapport de stage

effectué à Dwarf Animation Studio

MPM appliquée à la simulation de neige

par **Julien BRASSEUR**

Du **17 mars 2014**
au **29 août 2014**

Tuteur universitaire : **William PUECH.**
Maître de stage : **Bélisaire EARL.**

Remerciements

Je tiens tout d'abord à remercier Dwarf Animation Studio ainsi que son directeur général Olivier Pinol pour m'avoir accueilli dans sa structure et avoir mis à notre disposition tout les moyens qui lui étaient possibles pour mener à bien notre mission.
Je remercie tout particulièrement Bélisaire Earl pour le choix de ce sujet de stage à la fois passionnant, très enrichissant et ambitieux.
Je tiens également à remercier William Puech et Benjamin Gilles pour leur temps précieux qu'ils m'ont accordé lors de nos réunions, ainsi que leurs conseils.
J'aimerais aussi remercier Tuan-Huy Truong avec qui j'ai été ravi de collaborer pendant 6 mois.
Un remerciement spécial à Maria Giannakourou qui nous a rejoint en cours de route et qui s'est vraiment impliquée en plus de nous être d'une grande aide grâce à son expérience en matière d'effets spéciaux.
Enfin j'aimerais remercier Michael Delaporte, Jérémy Clément et Cédric Paille, pour tout les conseils et le savoir qu'ils m'ont apporté, ainsi qu'à tout le reste de l'équipe Dwarf.

Table des matières

Introduction

1.1 Dwarf Animation Studio

1.1.1 Présentation

Créé en 2010 par Olivier Pinol et Cédric Paille, Dwarf Animation Studio (ou Dwarf Labs) s'est basé dans le sud de la France. Le studio réunit des artistes et des ingénieurs dont certains ont déjà travaillé sur de grosses productions pour Dreamworks ou Weta Digital. Il a l'ambition de devenir une des références européennes voir internationales en matière de films d'animations. Le studio a déjà réalisé des courts métrages pour des entreprises telles que Cartier ou les Réseaux Ferrés de France. Dwarf Animation Studio doit sûrement sa croissance et son succès a la volonté d'allier l'art et une technologie toujours à la pointe. C'est d'ailleurs pour cela que Dwarf Animation tente de développer ses propres outils le plus possible. Le studio représente le pôle divertissement du groupe Septeo qu'il a rejoint en 2012 et qui lui a permis de se développer d'avantage. Dwarfs Labs a entre autres participé au festival de Cannes où il a présenté son court métrage « Lune et le loup ». En 2013 Olivier Pinol lance la première session de la Dwarf Academy qui est un centre de formation créé dans le but d'accompagner et de former de nouveaux artistes et techniciens pour les besoins du studio. Les élèves disposent des mêmes outils que les artistes et les cours sont dispensés par des professionnels travaillant chez Dwarf.

1.1.2 Fonctionnement de l'entreprise

Le studio fonctionne autour de ce que l'on appelle un « pipeline » : chaque département (concept 2D, modeling, texturing, surfacing, lighting, rigging, animation, ...) joue un rôle bien précis et à un moment précis lors d'une production. Ainsi les animateurs ne pourront pas animer les personnages tant que le département rigging n'aura pas créé le squelette de ces personnage. Et cela va de même entre le département modeling et texturing ainsi que pour les autres. Il est donc indispensable de planifier et d'optimiser ce pipeline afin d'anticiper tout les besoins et obstacles que l'équipe pourrait rencontrer pendant la réalisation d'un projet. C'est d'ailleurs une des missions confiée aux ingénieurs de Dwarf Labs que de faciliter le travail des artistes et répondre aux problèmes rencontrés.

1.2 Contexte

En plus de vouloir développer ses propres outils et être à la pointe de la technologie, le studio souhaite également éviter de rencontrer des problèmes survenus dans des productions passées. Lors de la collaboration avec Cartier les spécialistes en effets spéciaux ont été contraints de recourir à des astuces et contournements pour modéliser de la neige. Dans le même temps, une équipe d'ingénieurs de Disney Pixar et de mathématiciens de l'Université de Los Angeles publiait un papier au SIGGRAPH 2013 détaillant une méthode pour simuler de la neige. C'est donc dans ce cadre que la mission de mon stage de fin d'études prend place.

MPM

MPM (pour Material Point Method) est une méthode des éléments finis et une dérivée de la méthode PIC (Particle In Cell) utilisée dans les calculs de dynamique des fluides. Elle a été mise au point en 1993 par Sulsky, Chen et Schreyer, dans le but d'améliorer la simulation des problèmes de pénétration dans un solide.

A la différence de la méthode PIC, MPM est donc plutôt destinée aux calculs de dynamique des solides. Par la suite MPM sera améliorée pour inclure le calcul des frottements, ce qui permettra de simuler des flux granulaires, ainsi que les fissures et les propagations de fissures. C'est grâce à ces améliorations qu'Alexey Stomakhin et al. ont pu appliquer MPM à la modélisation de neige en incorporant certains changements notamment la déformation plastique.

MPM comporte 5 étapes principales : projeter les données des particules du domaine Lagrangien[1] vers une grille Eulérienne[2], évaluer ces données dans les grilles (notamment le champ de force) pour préparer l'évaluation des équations constitutives, résoudre le modèle constitutif, réévaluer les forces internes et enfin déplacer en conséquences les particules dans le domaine Lagrangien.

Dans toute la suite de ce document, les indices i et p feront respectivement référence au domaine Eulérien et Lagrangien ou autrement dit aux coordonnées du nœud dans la grille et à la position d'une particule dans le repère monde.

1. En dynamique des fluides la description lagrangienne est l'une des deux techniques qui permettent de caractériser un écoulement. Elle consiste à suivre dans le temps les particules fluides le long de leurs trajectoires.

2. En dynamique des fluides la description eulérienne décrit le champ de vitesses qui associe à chaque point un vecteur vitesse.

2.1 Discrétisation des masses et vélocités

MPM débute donc par remplir des grilles de nœuds en se basant sur les valeurs transportées au cours du temps par les particules, telles que la masse, la vélocité, la densité et le volume.

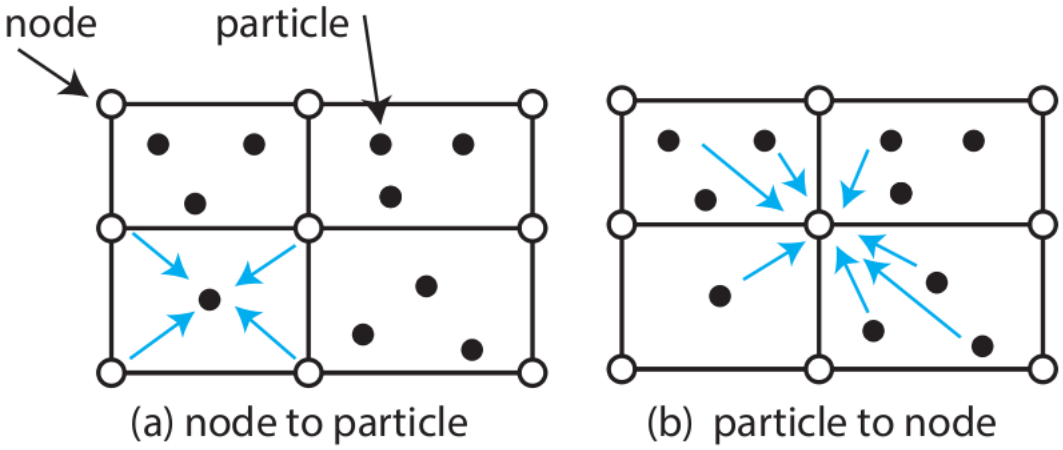


FIGURE 2.1: étapes de transfert des quantités. A gauche : transfert des noeuds vers les particules. A droite : transfert des particules vers les noeuds.

Ce transfert vers une grille Eulerienne se fait à l'aide d'une fonction de pondération N ou grid basis function(GBH) en anglais. La fonction de pondération utilisée par Alexey Stomakhin et al. est une B-spline cubique :

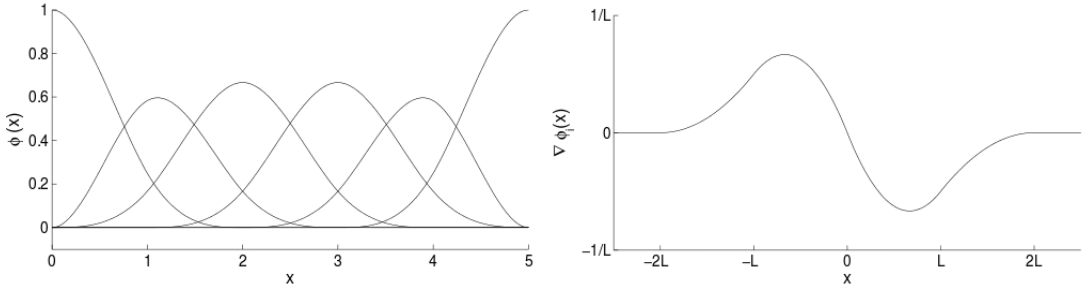


FIGURE 2.2: à gauche : la fonction de pondération B-spline cubique, à droite : le gradient correspondant à cette fonction

On peut écrire cette fonction comme suit :

$$N(x) = \begin{cases} \frac{1}{2}|x|^3 - x^2 + \frac{2}{3} & 0 \leq |x| < 1 \\ -\frac{1}{6}|x|^3 + x^2 - 2|x| + \frac{4}{3} & 1 \leq |x| < 2 \\ 0 & \text{sinon} \end{cases}$$

Cette fonction de pondération permet de donner un poids à un couple nœud/particule qui n'est autre que la représentation de l'influence qu'aura la particule sur ce nœud. Comme on peut le voir dans la fonction ci dessus, au delà d'une distance de 2 nœuds, le poids est nul et donc la particule n'a aucun effet sur le nœud. C'est pour quoi dans la suite de ce rapport, l'indice i en dessous des sommes : $\sum_i$ fait référence à l'ensemble des nœuds voisins à 2 de distance ou moins.

2.2 Calcul des forces internes

Cette étape consiste à remplir le champ de force à l'aide du stress de Cauchy et du gradient de poids selon l'équation suivante :

$$fi(x) = -\sum V_p \sigma_p \nabla w_{ip} \tag{2.1}$$

2.2.1 Gradient de poids

Le gradient de poids ∇w_{ip} permet de décrire les variations spaciales à l'intérieur du fluide, il se calcule en prenant la dérivée partielle de la fonction de pondération sur chaque axe, ce qui nous donne un vecteur 1x3 comme résultat.

2.2.2 Stress de Cauchy

Le stress de cauchy ou tenseur de stress est une matrice 3x3 décrivant l'étirement en un point donné dans un fluide. Ce stress de Cauchy se calcule ainsi :

$$\sigma_p = \frac{1}{J_p} \frac{\partial \psi}{\partial \hat{x}_i} \tag{2.2}$$

où J_p est le déterminant jacobien du gradient de déformation que nous aborderons plus tard et Ψ la fonction de densité d'énergie élasto-plastique :

$$\Psi(F_E, F_P) = \mu(F_P)||F_E - R_E||^2 + \frac{\lambda(F_P)}{2}(J_E - 1)^2 \tag{2.3}$$

où μ et λ sont les coefficients de Lamé.

2.2.3 Coefficients de Lamé

Les deux coefficients de Lamé μ et λ, nommés d'après le mathématicien français Gabriel Lamé, dépendent directement du module de Young E et du coefficient de Poisson v. Ces deux derniers définissent le comportement d'un materiau lors de sa déformation, v permet de caractériser la contraction du materiau et E sa traction.

$$\mu(F_P) = \mu_0 e^{\xi(1-J_P)} \qquad \lambda(F_P) = \lambda_0 e^{\xi(1-J_P)} \tag{2.4}$$

où μ_0 et λ_0 sont les coefficients de Lamé initiaux :

$$\mu_0 = \frac{E}{2(1+v)} \qquad \lambda_0 = \frac{Ev}{(1+v)(1-2v)} \tag{2.5}$$

2.3 Mise à jour du momentum

Grâce aux forces calculées précédemment nous pouvons maintenant mettre à jour la grille de vélocités :

$$v_i^* = v_i + \Delta t m_i^{-1} f_i \tag{2.6}$$

2.4 Résolution du système linéaire

La résolution du système linéaire ci dessous pour chaque vélocité dans la grille permet, selon la méthode employée, de gagner en précision ou en temps de calcul.

$$\sum_j \left(I\delta_{ij} + \beta \Delta t^2 m_i^{-1} \frac{\partial^2 \Phi}{\partial \hat{x}_i \partial \hat{x}_j} \right) v_j^{n+1} = v_i^* \tag{2.7}$$

où I est la matrice identité Δt le timestep et Φ la fonction d'énergie potentielle totale. Pour une résolution implicite, il suffit de prendre $\beta > 0$ et $\beta = 0$ pour une résolution explicite. La méthode explicite donne plus de précision dans la simulation mais demande plus de temps de calcul, la méthode implicite sera plus adaptée pour une simulation qui évolue lentement et qui demande moins de précision.

2.5 Calcul du gradient de déformation

L'étape suivante du MPM consiste à mettre à jour le gradient de déformation F. Le gradient de déformation est une matrice jacobienne[1] composée d'une matrice de translation (S_E) et d'une matrice de rotation (R_E). Ces dernières, ainsi que le déterminant jacobien de F sont utilisés comme on l'a vu au timestep suivant dans le calcul du stress de Cauchy. La mise à jour du gradient de déformation s'écrit donc ainsi :

$$F_p^{n+1} = (I + \Delta t \nabla v_p^{n+1}) F_p^n \tag{2.8}$$

avec ∇v_p^{n+1} le gradient de velocité (matrice 3x3) obtenu en multipliant la velocité v_i^{n+1} et la transposée du gradient de poids $\nabla (w_{ip})^T$:

2.6 Transfert des grilles vers les particules

La derniere étape du MPM consiste a retransferer les velocités dans la grille de velocités vers les particules en utilisant une fois de plus la fonction de pondération :

$$v_p^{n+1} = (1 - \alpha) v_{PICp}^{n+1} + \alpha v_{FLIPp}^{n+1} \tag{2.9}$$

ou $\alpha = 0.95$ et v_{PIC} et v_{FLIP} sont remplacés par :

$$v_{PICp}^{n+1} = \sum_i v_i^{n+1} w_{ip} \qquad v_{FLIPp}^{n+1} = v_p + \sum_i (v_i^{n+1} - v_i^n) w_{ip} \tag{2.10}$$

2.7 Modifications du MPM

Je vais détailler dans cette section tout les changements apportés au MPM pour s'approcher de l'aspect de la neige. Lors du début du développement du solveur de neige, nous avons essayé d'implémenter toutes les étapes décrites par Stomakhin et al. dans leur papier. Or ces étapes ne font pas toutes partie du MPM, certaines ont été ajoutées ou modifiées pour obtenir le comportement de la neige :

1. En analyse vectorielle, la matrice jacobienne est une matrice associée à une fonction vectorielle en un point donné. Le déterminant de cette matrice, appelé jacobien, joue un rôle important dans la résolution de problèmes non linéaires.

— Gestion des collisions sur la grille. Les collisions sont calculées deux fois à chaque timestep, une fois sur la grille et une fois sur les particules. Cette partie sera gérée par Houdini qui fournit des méthodes simples pour ce genre de traitement.
— Insertion d'une partie « plastique » au gradient de déformation. La neige a cette particularité de se comporter parfois comme un fluide et parfois comme un solide. Il a donc fallut séparer la déformation en deux parties. Une partie élastique et une partie plastique. La partie élastique correspond à un gradient de déformation prenant ses valeurs à l'intérieur d'une certaine plage. Tant que le gradient reste à l'intérieur de cette plage, la déformation est réversible. Lorsque le gradient dépasse la borne supérieure ou inférieure, la déformation devient alors irreversible ce qui a pour effet de fissurer le solide (voir figure ci dessous). Dans notre cas, des paquets de neige se détachent lors d'une déformation plastique.

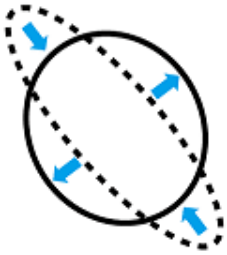


FIGURE 2.3: représentation du gradient de déformation. les flèches bleues représentent la direction vers la position initiale du solide.

— Ajout d'un effet « collant » lors du contact entre la neige et un corps externe. Cet effet est réalisé simplement en mettant les vélocités à 0 lors du contact avec les surfaces auxquelles on souhaite que la neige adhère.

Houdini

FIGURE 3.1: Logo de la suite Houdini de la societe SideFx

Houdini est un outil très complet pour l'animation et les effets spéciaux, développé par la société SideFx. Il est utilisé autant dans les grands studios que par les particuliers et les universités grâce à sa version gratuite. Son domaine d'application s'étend de la modélisation à l'éclairage en passant par la composition, le rendu de personnages, de nuages et de volumes ainsi bien sûr que l'animation. Houdini intègre également d'autres outils très puissants permettant de créer des simulations hautement paramétrables telles que du feu, de la fumée, des fluides, des particules, des vêtements ou tout autre objet obéissant aux lois de la physique. Houdini propose un processus de travail procédural basé sur des nœuds qui permet de faire gagner du temps aux artistes lors des productions.

3.1 Architecture nodale

Dans Houdini, chaque action est stockée dans un nœud (voir figure ci dessous). Que l'on crée une nouvelle géométrie, que l'on applique une translation ou que l'on initialise des paramètres, toutes les actions sont transcrites sous forme de nœuds. Ces nœuds sont interconnectables, modifiables, paramétrables et déplaçables. Lors du lancement d'une animation, Houdini va parcourir ces nœuds de haut en bas et de droite à gauche. Un nœud se présente sous cet aspect : Au centre une miniature

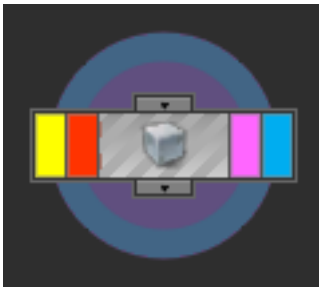


FIGURE 3.2: Représentation graphique d'un noeud dans le gestionnaire nodale d'Houdini

permet de reconnaître rapidement le type de nœud (ici une géométrie cube). Au dessus les connexions entrantes, c'est à dire le flux de données qui arrive dans ce nœud et qui vont être traitées par ce dernier (dans l'exemple il n'y en a qu'une mais il peut y en avoir plusieurs). Au dessous les connexions sortantes (même chose que pour les connexions entrantes). A gauche et à droite du nœud on peut voir ce que l'on appelle des « flags », qui sont des options du nœud à activer ou désactiver :

- Bypass(jaune) : Ce flag, lorsqu'il est activé, permet de préciser à Houdini qu'il ne doit pas le traiter lors du parcours des nœuds.
- Lock(orange) : Ce flag permet d'empêcher d'appliquer les modifications faites dans les paramètres du nœud. C'est à dire que le nœud ne mettra pas à jour ses paramètres (par exemple la taille du cube sur le nœud précédent) tant qu'il ne sera pas débloqué.
- Template(rose) : Ce flag permet d'afficher la géométrie en fils de fer dans le viewport[1] d'Houdini.
- Display(bleu) : Ce flag permet de préciser à Houdini si il doit ou non inclure le nœud dans le rendu de l'animation.

Les figures ci dessous démontrent qu'il est possible de créer très rapidement ce que l'on veut avec Houdini, juste en associant les bons nœuds.
Dans la première figure on peut voir le résultat obtenu juste en utilisant le nœud de l'exemple.

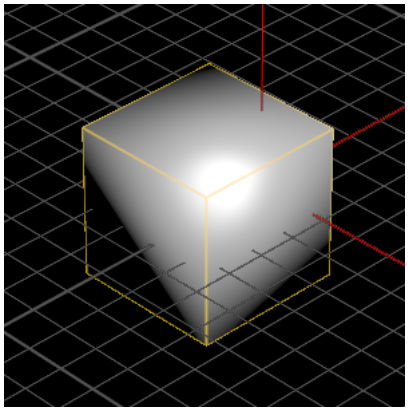
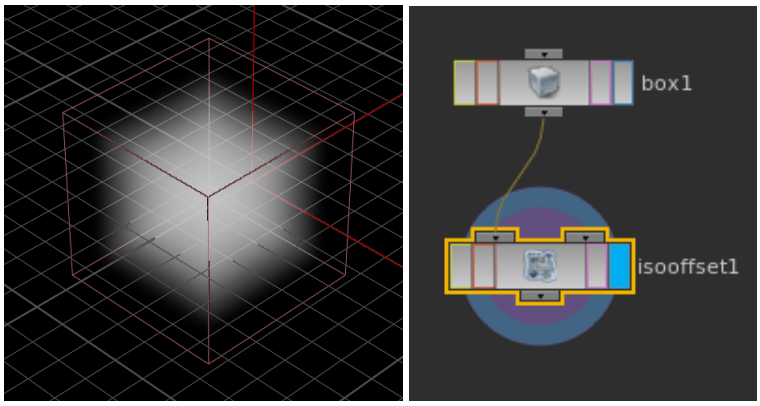


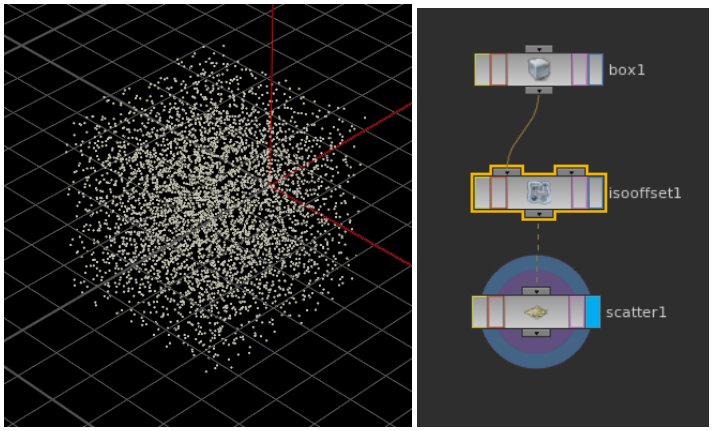
FIGURE 3.3: Une geometrie cubique dans le viewport d'Houdini

1. Le viewport est une interface graphique permettant de visualiser une scène en 3D.

On peut connecter un autre nœud en sortie du premier, qui va générer un nuage de fumée à partir de la géométrie :



Pour finir on ajoute un nœud qui va générer des particules à partir des données en entrée. Voici donc la façon de créer un nuage de points :



Il existe beaucoup d'autres nœuds, chacun ayant un type correspondant au réseau auquel il appartient.

3.2 Réseaux

Les réseaux permettent d'organiser les nœuds dans Houdini, sachant qu'un nœud peut lui même contenir un réseau dans lequel se trouve d'autres nœuds. On peut donc traverser des réseaux dans Houdini en entrant dans des nœuds et en en sortant comme on naviguerait dans une architecture de dossiers et de fichiers UNIX. Il existe 9 types différents de réseaux :

Réseau	Description
Objects (OBJ)	Contient les objets dits de « haut niveau » de la scène (géométries, squelettes, lumières, caméras, ...) C'est à ce niveau que l'artiste va placer ses objets dans l'espace et créer des liens hiérarchiques entre eux.
Geometry (SOP)	Ce type de réseau est contenu a l'intérieur d'un nœud au niveau supérieur (OBJ). Il contient les opérateurs de surface (SOPs) qui donnent sa forme à un objet.
Particles (POP)	Permet de créer une simulation de particules a l'intérieur de ce réseau, ce dernier possède des nœuds qui généreront les particules et d'autres qui agiront sur les particules (force nodes)
Dynamics (DOP)	Ce réseau contient des opérateurs dynamiques appelés solvers qui agissent sur les objets chargés à l'intérieur de ce réseau.
Shaders (SHOP)	Contient des nœuds capables de contrôler l'apparence des surfaces au rendu. Il y a généralement peu de connexions à l'intérieur de ce réseau car chaque nœud représente un shader qui travaille sur les données en entrée.
VEX Builder (VOP)	Ce réseau sert généralement lui aussi à créer des shaders à la différence que les shaders ici sont programmables par l'artiste en utilisant le langage VEX. VEX est un langage d'expressions ressemblant au C et C++ et utilisable à de nombreux endroits dans Houdini.
Motion and audio channel operators (CHOP)	Le réseau CHOP contient des nœuds permettant de manipuler des données sous forme d'ondes, généralement pour créer et éditer des courbes d'animations mais parfois aussi pour manipuler du son.
Compositing (COP)	Contient des opérateur permettant de manipuler des pixel maps. Ces dernières sont utilisées pour la composition d'images tels que les passes de rendu.
Render outputs (ROP)	Contient des opérateurs contrôlant la sortie des images et des animations.

3.3 Réseau DOP

C'est dans ce réseau que nous allons mettre en place tout les nœuds nécessaires à l'implémentation de notre méthode MPM. Le réseau DOP contient des objets de simulation construits et contrôlés par les « nœuds DOP » aussi appelés micro solvers. Le réseau DOP est lui même contenu dans un nœud « DOP network » dans le réseau OBJ. Il est ainsi possible d'avoir plusieurs simulations dans une même animation. Une simulation très simple dans Houdini ressemblerait à ceci :

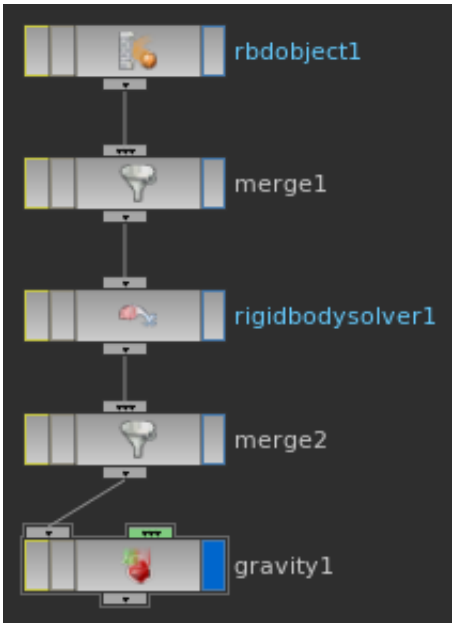


FIGURE 3.4: representation graphique des noeuds utilises pour realiser une simulation simpliste dans Houdini

Sur cette figure, on commence par créer un objet dynamique, ce dernier peut être prédéfini ou être chargé depuis le réseau SOP (il serait possible de récupérer le nuage de point du dernier exemple). On le définit ensuite en tant que « rigid body », cela a pour effet de rendre l'objet sensible aux forces et aux collisions. Enfin on applique une gravité. Les nœuds « merge » servent à assembler plusieurs flux de données entrants.

Une simulation conséquente telle qu'une simulation de fluide est l'association entre un grand nombre d'objets dynamiques et de microsolvers (les microsolvers sont des nœuds dynamiques ayant la plupart du temps « Gas » pour préfixe, faisant référence aux fluides en général). Houdini propose d'ailleurs un solver de fluide appelé FLIP (mis au point dans les années 90 et implémentant la méthode PIC, il est en continuelle amélioration depuis). Le FLIP solver est une grosse usine à gaz qui convertit n'importe quelle géométrie en fluide pour ensuite le simuler. La figure ci dessous donne une idée de l'organisation interne du FLIP solver :

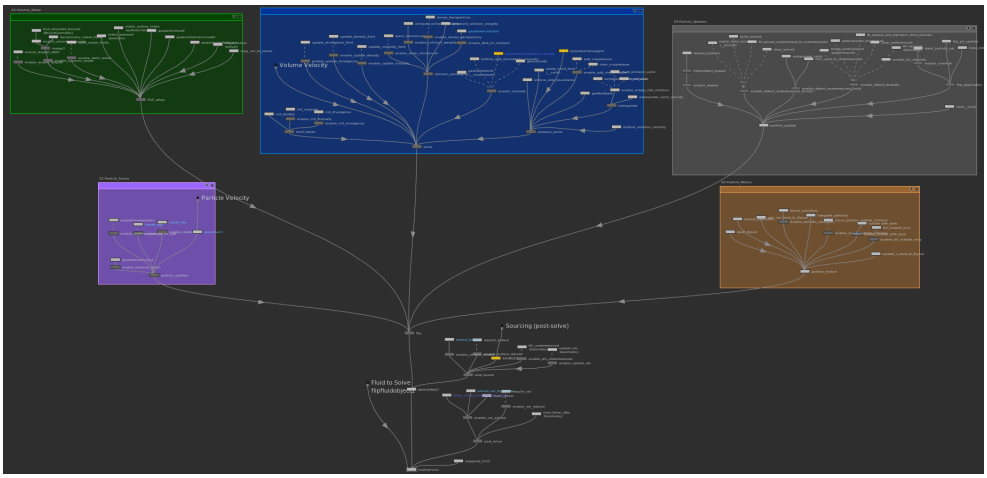


FIGURE 3.5: Représentation graphique de la methode FLIP qui sert à modeliser et simuler un fluide

En rose les forces externes appliquées au fluide, en vert toutes les initialisations et la mise en place des objets dynamiques utilisés, en bleu le cœur de la simulation, en gris la mise à jour des quantités sur les particules et en orange le déplacement effectif des particules. Nous nous sommes inspirés de cette architecture pour créer notre propre simulation en la simplifiant et en ne gardant que le strict minimum pour coller au plus près de la méthode MPM. Nous avons repris le même système d'initialisation des composants de la simulation, le déplacement des particules, la gestion des collisions et des forces externes.

Ci dessous l'architecture créée dans laquelle nous avons implémenté le MPM :

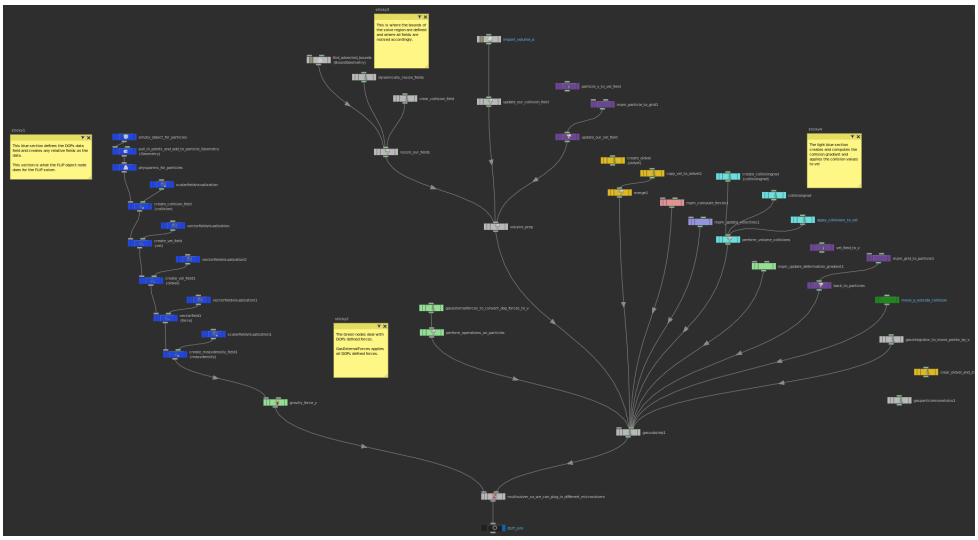


FIGURE 3.6: Représentation graphique du réseau de noeuds utilisé pour implementer le MPM dans Houdini

On peut voir en bleu la création des grilles Eulériennes qui ont été évoquées dans le chapitre sur le MPM, ainsi que des nœuds permettant de visualiser leur contenu pour faciliter le processus de développement.

En vert clair(au centre et à gauche) les nœuds gérants les forces externes. En gris dans la partie supérieure sont les nœuds qui redimensionnent les grilles et chargent l'objet à simuler (ici des particules) à l'intérieur de notre réseau (qu'on appelle également « solver »).

Sur la partie droite, les nœuds en bleu clair calculent les collisions et les appliquent à la grille des vélocités. Les nœuds restants sont les nœuds contenant les étapes du MPM : en violet le transfert des particules vers les grilles en premier et des grilles vers les particules à la fin.

En rouge le nœud qui calcule les forces internes, en violet clair la mise à jour des vélocités grâce à la grille des forces et en vert clair la mise à jour du gradient de déformation.

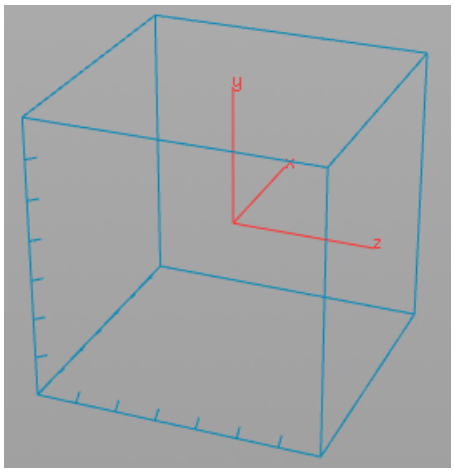


FIGURE 3.7: representation graphique d'un field dans le viewport d'Houdini

Les grilles dont nous parlons dans le MPM correspondent à ce que l'on appelle des « fields » dans Houdini. Ces fields sont composés de voxels jouant le rôle des nœuds du MPM.
Les fields d'Houdini sont redimensionnables, tant au niveau du field lui même qu'au niveau des voxels à l'intérieur et de leur nombre. De plus il est possible, grâce à une certaine combinaison de nœuds, de redimensionner les fields pour qu'ils englobent parfaitement notre objet simulé, ce qui nous facilite grandement la tâche.

Ce qu'on ne voit pas sur la figure 3.6 est qu'Houdini gère tous les points des géométries chargées dans la simulation. Chaque point, ou particule dans notre cas, peut porter autant d'attributs que l'on désire, de tout type (scalaires, vecteurs, matrices, ...).
Une autre particularité, qui fait qu'Houdini se place comme l'outil parfait pour notre simulation, est qu'il y a une persistance des quantités sur les particules au fil du temps (des frames de l'animation). Cela permet d'appliquer parfaitement le principe du MPM qui est de vider les fields et les remplir à chaque itération de l'animation.

Les nœuds contenant les étapes du MPM n'existant bien évidemment pas nativement dans Houdini, il a fallu les développer, et ce grâce au HDK.

Développement

4.1 HDK

Le HDK (pour Houdini Development Kit) est un ensemble de librairies écrites en C++. Ces librairies sont les mêmes que celles utilisées par la société SideFx pour développer leur suite de logiciels. Le HDK permet ainsi plusieurs choses comme :

- Créer de nouveau nœuds de n'importe quel type (SOPs, DOPs, POPs, ...).
- Ajouter des fonctionnalités au langage de scripting VEX inclut dans Houdini
- Dessiner à l'intérieur du viewport en interceptant les géométries avant qu'elles ne soient affichées
- Agrandir le nombre de types de fichiers différents supportés par Houdini
- ...

4.1.1 Bases du HDK

Ci dessous un exemple d'architecture basique du HDK.

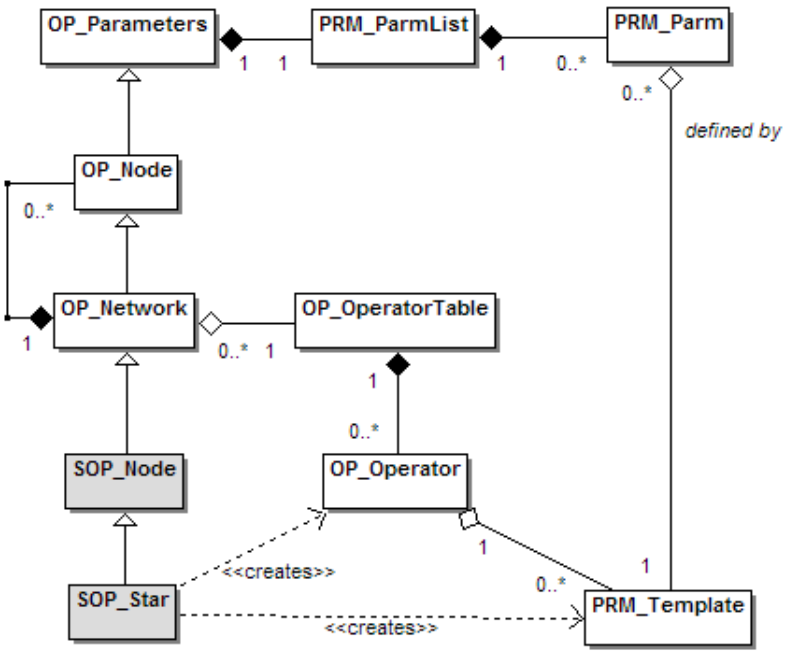


FIGURE 4.1: architecture de base du HDK presentant certaines classes

On peut voir sur cette figure que tous les nœuds dérivent de l'objet OP_Node qui lui-même dérive de la classe OP_Parameters pour des soucis d'abstraction. La classe SOP_Node représente les nœuds qui modifient les surfaces On peut également voir la classe SOP_Star (qui ne fait pas partie du HDK mais a été codée en l'utilisant)représentant un nœud permettant de créer une étoile en 2 dimensions dans le viewport d'Houdini. Cette classe possède une liste de paramètres qui seront éditables une fois la classe compilée et chargée a l'intérieur d'Houdini.

Pour compiler un ou des fichiers utilisant le HDK, SideFx a fourni un outil très pratique appelé « hcustom » qui s'utilise en ligne de commande et qui permet de générer un plugin. Ce plugin se présente sous la forme d'une librairie dso (sous Linux) et est placée automatiquement par hcustom dans le dossier ou Houdini viendra charger ses librairies lors du démarrage.

Après avoir compilé la classe SOP_Star, on obitendra donc une librairie « SOP_Star.dso ». En démarrant Houdini et en créant un réseau SOP, il sera alors possible d'ajouter le nœud « SOP_Star » à l'intérieur d'un réseau SOP et lancer l'animation qui générera une étoile. Comme dit dans le chapitre précèdent, nous sommes dans le cadre d'une simulation et donc a l'intérieur d'un réseau DOP.

4.2 Fonctions de récupération des données

Le premier besoin fut de pouvoir récupérer les données créés à l'intérieur de notre réseau comme on a pu le voir dans le chapitre précédent. Ces données sont les fields et les quantités stockées sur les particules comme la masse, la vélocité ou encore le gradient de déformation. Le HDK propose des fonctions et gestionnaires pour effectuer cette tâche.

Pour récupérer un field il faut procéder comme suit :

```
SIM_DataArray dstMass;

getMatchingData(dstMass, m_obj, SOURCE_MASSFIELD_NAME);
if (dstMass.entries() <= 0)
{
  addError(m_obj, SIM_MESSAGE, "No mass field found.", UT_ERROR_WARNING);
}
else
{
  m_massField = SIM_DATA_CAST(dstMass(0), SIM_ScalarField);
}
```

Ici pour récupérer le mass field, nous créons d'abord un conteneur SIM_DataArray qui accueillera toutes les données renvoyées par la fonction getMatchingData. Cette fonction filtre les données contenues dans m_obj qui ont comme nom « SOURCE_MASSFIELD_NAME » et les stock dans l'objet SIM_DataArray.

Si ce conteneur n'est pas vide, nous récupérons le premier (et unique dans notre cas) field qu'il contient pour le stocker dans un attribut de type SIM_ScalarField de notre classe. Sinon un message d'erreur est affiché dans Houdini indiquant à l'artiste qu'il manque un field.

Cela fonctionne de la même manière pour les géométries :

```
SIM_DataArray geos;
getMatchingData(geos, m_obj, SOURCE_GEO_NAME);
if (geos.entries() == 0)
{
    addError(m_obj, SIM_MESSAGE, "No geometry found.", UT_ERROR_WARNING);
}
else
{
  m_geo = SIM_DATA_CAST(geos(0), SIM_Geometry);
}
```

Ces fonctions étaient présentes dans nos classes à chaque initialisation d'un des nœud.

4.3 Accès et modifications des données

Une fois nos données recuperée, le principe pour y accéder est quasiment le même entre les fields et les particules. Dans les deux cas il est nécessaire de récupérer les données brutes. Car en effet nos objets SIM_Geometry et SIM_ScalarField (et aussi SIM_VectorField pour les fields de vecteurs) sont des encapsulations à haut niveau dans la librairie. Pour pouvoir effectuer des opérations simples sur les valeurs de ces objets, il faut utiliser des fonctions nous donnant des types de données plus brut tels que des tableaux de voxels (pour les fields) ou une liste de points (pour la géométrie). Ainsi pour les fields :

```
UT_VoxelArrayF *massFieldNC = m_massField->
                                  getField()->fieldNC();
```

Il est maintenant possible d'utiliser ces tableaux de voxels pour récupérer et modifier les valeurs. Soit en accédant aléatoirement à ce tableau avec les indices du voxel voulu, soit en utilisant un itérateur qui est beaucoup plus optimisé.

Cependant les fields de vecteurs ont une particularité : Il faut récupérer 3 tableaux de voxels, chacun contenant les valeurs des vecteurs sur un axe, ainsi le premier tableau contient toutes les valeurs en x, etc.

```
UT_VoxelArrayF **velocityFieldNC = new UT_VoxelArrayF*[3];
velocityFieldNC[0] = m_velocityField->getField(0)->fieldNC();
velocityFieldNC[1] = m_velocityField->getField(1)->fieldNC();
velocityFieldNC[2] = m_velocityField->getField(2)->fieldNC();
```

Pour la géométrie cela est légèrement différent car il faut en plus appliquer un verrou avant de lire et d'écrire sur les particules puis libérer la géométrie à la fin :

```
m_geoDetailHandle = m_geo->getOwnGeometry();
m_détail = m_geoDetailHandle.readLock();
getAttrHandlers();
m_geoDetailHandle.unlock(m_detail);
```

Entre les verrous on peut voir la méthode qui initialise les « attributes handlers » qui sont des gestionnaires permettant chacun d'accéder à un attribut des particules de la géométrie :

```
m_massAttrHandler = GA_RWHandleF(gdp->
                findPointAttribute(GEO_STD_ATTRIB_MASS));
if (!m_massAttrHandler.isValid())
{
  m_massAttrHandler = GA_RWHandleF(gdp->
        addFloatTuple(GA_ATTRIB_POINT, GEO_STD_ATTRIB_MASS, 1,
                      GA_Defaults(0.0)));
}
```

Toujours sur le même principe, on cherche un attribut sur la particule qui porte le nom « GEO_STD_ATTRIB_MASS » et on l'affecte à notre gestionnaire (ici un gestionnaire d'attributs scalaire). Si l'attribut n'existe pas, il est créé et restera sur la géométrie jusqu'à la fin de l'animation. Cela fut très utile pour créer des attributs personnalises qu'il n'était pas possible de créer à l'intérieur d'Houdini, tels que le gradient de déformation. On peut maintenant accéder aux attributs d'une particule en utilisant le bon gestionnaire et l'identifiant de la particule :

```
fpreal64 massP = m_massAttrHandler.get(ptOff);
```

4.4 Calcul du poids

Nous avons longuement cherché une méthode pour pouvoir parcourir nos tableaux de voxels (avec un itérateur)et récupérer les particules voisines, comme le veut la méthode de transfert des particules vers les fields. Sans succès, nous avons choisi une approche différente. Nous avons trouvé le moyen de parcourir les particules et récupérer l'indice du voxel auquel la particule courante appartient. A partir de la il est possible de calculer la distance entre la particule courante et tout les voxels voisins puis d'utiliser la fonction de pondération pour calculer le poids :

```
GA_FOR_ALL_PTOFF(m_detail, ptOff) // macro permettant de
//parcourir toutes les particules de la géométrie
{
  // récuperation de la position de la particule dans le
  //repère monde
  partPos = m_detail->getPos3(ptOff);
  m_massField->indexToPos( 0, 0, 0,m_fieldOrigin);

  // mise a l'echelle de l'indice du voxel vers sa position
  // dans le repère monde
  partPos-=m_fieldOrigin;
  partPos/=m_gridSpacing;
  partPos-=0.5;

  i=std::floor(partPos.x())-1;
  j=std::floor(partPos.y())-1;
  k=std::floor(partPos.z())-1;

  double xwarray[4];
  double ywarray[4];
  double zwarray[4];
```

```
  //parcours des voxels voisins sur lesquels
  //la particule courante a une influence
  // et calcul des poids sur chaque axe
  for(int ii=0;ii<=3;++ii)
  {

  if((i+ii) >= 0 && (i+ii) < m_fieldDivisionsX )
  {
    double dx = partPos.x()-(i+ii);
    //calcul du poids en x
    xwarray[ii] = utils::computeNx(dx, m_gridSpacing);
  }
  else
    xwarray[ii] = 0.0;

  if((j+ii) >= 0 && (j+ii) < m_fieldDivisionsY )
  {
    double dy = partPos.y()-(j+ii);
    ywarray[ii] = utils::computeNx(dy, m_gridSpacing);
  }
  else
    ywarray[ii] = 0.0;
  if((k+ii) >= 0 && (k+ii) < m_fieldDivisionsZ)
  {
    double dz = partPos.z()-(k+ii);
    zwarray[ii] = utils::computeNx(dz, m_gridSpacing);

  }
  else
    zwarray[ii] = 0.0;
  }
}
```

Il est ensuite possible de calculer le poids final avec un produit dyadique :

```
w = xwarray[ii]*ywarray[jj]*zwarray[kk];
```

et calculer la valeur desirée pour le field, ici la masse pour le mass field :

```
massFieldNC->setValue(i+ii, j+jj, k+kk, massFieldNC->
                          getValue(i+ii, j+jj, k+kk) + (massP*w));
```

4.5 Types de données d'Houdini

Les types fournis par Houdini ont été utilisés dès que possible pour des raisons pratiques et de gain de temps. Houdini propose des classes représentant des matrices 3x3, des matrices non carrées, des vecteurs et autres. Les deux types que nous avons le plus utilisé sont « UT_Matrix3T » et « UT_Vector3T ». Ces deux types fournissent des méthodes très utilisées en physique et en 3D qui les rendent indispensables : transposée, inverse, matrice identité, mise a zéro, déterminant, multiplication de matrices, soustractions, multiplications entre un vecteur et une matrice et bien d'autres.

Pour ne pas surcharger de code ce chapitre, d'autres fonctions que j'ai estimé intéressantes se trouvent dans les annexes.

Conclusion

5.1 Résultats

Ce stage a commencé par un long travail de recherches, mon binôme Tuan Truong s'est plongé dans les papiers scientifiques référencés par l'équipe de Disney Pixar ayant rédigé la méthode MPM appliquée à la neige.

Quant à moi je me suis penché sur le fonctionnement d'Houdini : j'ai d'abord essayé de créer des scènes très simples, puis de les agrémenter en tentant d'anticiper le résultat lors de mes actions sur les nœuds et réseaux.
C'est durant cette période que j'ai beaucoup posé de questions autour de moi et que j'ai été demander conseil aux spécialistes en effets spéciaux de Dwarf Animation Studio. Régulièrement je confrontais mes recherches à celles de mon binôme pour trouver des points communs et commencer à élaborer un planning.

Je me suis ensuite intéressé au HDK et à ses nombreuses possibilités. J'ai codé des nœuds de test pour etudier le fonctionnement du HDK et commencer à réfléchir à la manière d'organiser nos noeuds.
Il y a d'abord eu une première ébauche où nous avons créé 3 noeuds, deux pour les transferts et un autre regroupant le calcul des forces, la mise à jour des vélocités et du gradient de déformation.

Ce n'est que durant la deuxième partie du stage que Maria Giannakourou nous a rejoint sur le projet et aiguillé vers une architecture mieux pensée. Elle nous a tout d'abord conseillé de mettre de côté les étapes permettant d'obtenir le comportement de la neige, pour d'abord implémenter une méthode MPM pure. Nous avons également divisé nos noeuds pour en obtenir cinq (cf. Figure 3.6).

Au terme de ce stage nous avons les noeuds de transferts complètement fonctionnels, ci dessous une représentation du champ de masses. On voit bien que les particules sont bien englobées par la fumée ce qui veut dire que les voxels possèdent de bonnes valeurs et que le champ ne possède pas de valeurs aberrantes.

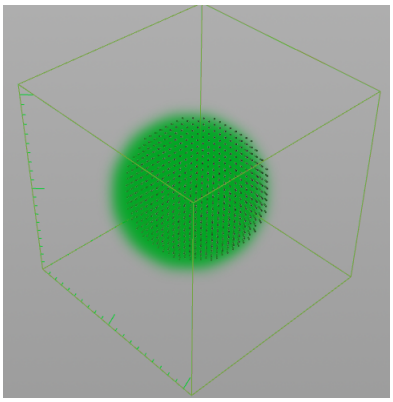


FIGURE 5.1: Affichage des valeurs du mass field sous forme de fumée

Il en est de même pour le champ de vélocités. Comme on peut le voir sur la figure ci dessous, les vélocités sont cohérentes car elles pointent bien vers le bas en raison de la gravité.

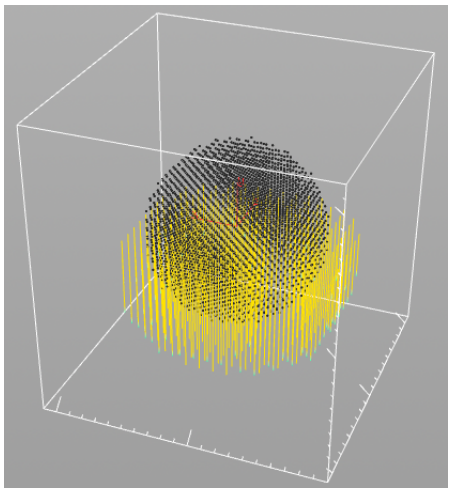


FIGURE 5.2: Affichage des valeurs d'une tranche du velocity field sous forme de vecteurs

Si on fait abstraction des autres noeuds lors de la simulation, on obtient un comportement très satisfaisant. On peut observer les particules tomber et rebondir grâce aux collisions et à la réponse avec une surface.

En revanche dès que nous intégrons les autres noeuds (calcul des forces, mise à jour des vélocités, mise à jour du gradient de déformation) nous avons un comportement anormal avec les particules qui éclatent sur une très longue distance. Nous pensons que ce problème viendrait du calcul du stress de Cauchy et indirectement du déterminant du gradient de déformation.

5.2 Problèmes rencontrés

5.2.1 Problèmes de connaissances

Ce problème évoqué ci dessus vient principalement du manque de connaissances dans le domaine de la dynamique des fluides et de la physique. Nous avons rencontré un réel obstacle pour déterminer si les valeurs que nous obtenions étaient dans un ordre de grandeur correct ou non.
Nous avons également découvert grâce à Maria Giannakourou que toutes les matrices et les vecteurs du papier sur lequel nous nous basions, étaient écrits en colonnes alors que nous les traitions comme si ils étaient écrits en lignes depuis le début. Toutes les opérations et l'ordre dans lequel nous faisions ces opérations ont du être repensés.

5.2.2 Problèmes Techniques

Le plus gros soucis au niveau du HDK fut la faible documentation de certaines fonctions et l'absence du corps pour d'autres ce qui nous a causé plusieurs incertitudes quant à leur fonctionnement. Nous avions heureusement accès à l'intégralité des classes de matrices et de vecteurs etant donné que nous les utilisions souvent.

5.3 Améliorations

La première amélioration à apporter à ce projet serait tout d'abord de résoudre les problèmes au niveau du calcul des forces. Il serait ensuite intéressant d'optimiser les accès aux données ainsi que les parcours de voxels car nous atteignons déjà un temps de calcul de plus d'une minute par image lors de la simulation pour seulement 4000 particules. L'équipe de Stomakhin et al. annonce 5.2 minutes par image pour 300000 particules.

Grâce à Dwarf Animation Studio nous avons eu l'opportunité d'avoir une discussion autour de notre projet avec les ingénieurs de SideFx ayant développé le HDK. Ils nous ont donné des conseils et des idées d'optimisation notamment concernant le parcours des tableaux de voxels. Un des développeurs nous a détaillé le fonctionnement des itérateurs de voxels dans le HDK et nous a suggéré d'utiliser une certaine technique pour ne pas parcourir des sections du field qui ne contiennent pas de particules. Il nous a également fait découvrir le principe des MAC grids qu'il serait intéressant d'implémenter. Enfin pour optimiser au mieux notre solver il faudrait utiliser toutes les fonctions « threadées » que le HDK propose. D'autant plus qu'avec le système de partage de la puissance de calcul déployé au studio, un code threadé prendrait tout son intérêt.

J'ai définitivement espoir qu'un jour ce simulateur de neige soit fonctionnel et ait sa place au sein du pipeline du studio. J'ai en tout cas beaucoup appris de ce stage et du milieu de la 3D qui me tient à cœur. Malgré ce projet inachevé je retiens une expérience enrichissante qui ne fait que confirmer mon attirance pour ce domaine.

Références

— A material point method for snow simulation (Alexey Stomakhin et al., 2013)

— Material point method : basics and applications (Vinh Phu Nguyen, Cardiff University)

— An Evaluation of the Material Point Method (Zhen Chen, University of Missouri-Columbia, 2002)

— FEM Simulation of 3D Deformable Solids : A practitioner's guide to theory, discretization and model reduction.(Eftychios D. Sifakis, University of Wisconsin-Madison, 2012)

— Implicit Dynamics in the Material-Point Method(D. Sulsky, University of New Mexico, 2003)

— Position Based Fluids(Miles Macklin & Matthias Müller, NVIDIA)

— Augmented MPM for phase-change and varied materials (Stomakhin et a.l, 2014)

— http://www.sidefx.com/docs/hdk13.0/
— http://www.scratchapixel.com/
— http://forums.odforce.net/forum/23-hdk-houdini-development-kit/
— http://fr.wikipedia.org/
— http://www.continuummechanics.org
— http://www.flow3d.com/

Table des figures

Annexes

7.1 Fonction de calcul du poids sur un axe

```
double
computeNx(const double val, double h)
{
  double x = std::abs(val);
  double xsquare = std::pow(x, 2.0);
  double xcube = std::pow(x, 3.0);

  if(x >= 0.0 && x < 1.0)
  {
    return xcube*0.5 - xsquare + 2.0/3.0;
  }
  else
  {
    if(x >= 1.0 && x < 2.0)
    {
      return -xcube / 6.0 + xsquare - 2.0 * x + 4.0/3.0;
    }
    else
      {return 0.0;}
  }

  cout<<"mauvaise bornes poids "<<endl;
  return 0.0;
}
```

7.2 Fonction de calcul du gradient sur un axe

```
double
  computegNx(const double& val, double h)
  {
    cout << val << endl;
    double x = val;
    if(x >= 0.0 && x < 1.0)
      return (1.5 * std::pow(x, 2.0) - 2.0 * x);
    if(x >= 1.0 && x < 2.0)
      return (-0.5 * std::pow(x, 2.0) + 2.0 * x -2.0);
    if(x < 0.0 && x > -1.0)
      return (-1.5 * std::pow(x, 2.0) - 2.0 * x);
    if(x <= -1.0 && x > -2.0)
      return (0.5 * std::pow(x, 2.0) + 2.0 * x + 2.0);
    return 0.0;
  }
```

7.3 Header du noeud qui calcule les forces

```
class computeForces : public GAS_SubSolver
{
public:
protected:
  explicit    computeForces(const SIM_DataFactory *factory);
  virtual     ~computeForces();

  virtual bool         solveGasSubclass(SIM_Engine &engine,
                              SIM_Object *obj,
                              SIM_Time time,
                              SIM_Time timestep);

private:
  //FUNCTIONS
  static const SIM_DopDescription *getDopDescription();

  DECLARE_STANDARD_GETCASTTOTYPE();
  DECLARE_DATAFACTORY(computeForces,
          GAS_SubSolver,
          "MPM Compute Forces",
          getDopDescription());

  bool init(SIM_Time time);
  void initFields();
  void initGradientDeformationAttributes();
  void updateForces(SIM_Time timestep);
  UT_Matrix3D manuCauchy(GA_Offset ptOff, SIM_Time timestep);
  UT_Matrix3D NeoHookeanCauchy(GA_Offset ptOff, SIM_Time timestep);
  fpreal64 lambdaLameCoeff(fpreal64);
  fpreal64 muLameCoeff(fpreal64);
  UT_Matrix3D estimateNewPlasticPart
                          (const GA_Offset ptOff, SIM_Time timestep);
```

```
  void getFields();
  void getGeometry();
  void getAttrHandlers();

  //ATTRIBUTES

  //general
  SIM_Object *m_obj;
  const GU_Detail* m_detail;
  GU_DetailHandle m_geoDetailHandle;
  SIM_Geometry *m_geo;

  //fields
  SIM_VectorField *m_velocityField;
  SIM_VectorField *m_forceField;

  //field dimensions
  int m_fieldDivisionsX;
  int m_fieldDivisionsY;
  int m_fieldDivisionsZ;
  fpreal64 m_gridSpacing;
  UT_Vector3 m_fieldOrigin;

  //Attribute Handlers
  GA_RWHandleM3 m_FpAttrHandler;
  GA_RWHandleM3 m_elasticAttrHandler;
  GA_RWHandleM3 m_plasticAttrHandler;
  GA_RWHandleV3 m_velocityAttrHandler;
  GA_RWHandleF m_densityAttrHandler;
  GA_RWHandleF m_massAttrHandler;
  GA_RWHandleF m_VolumeAttrHandler;
};
```