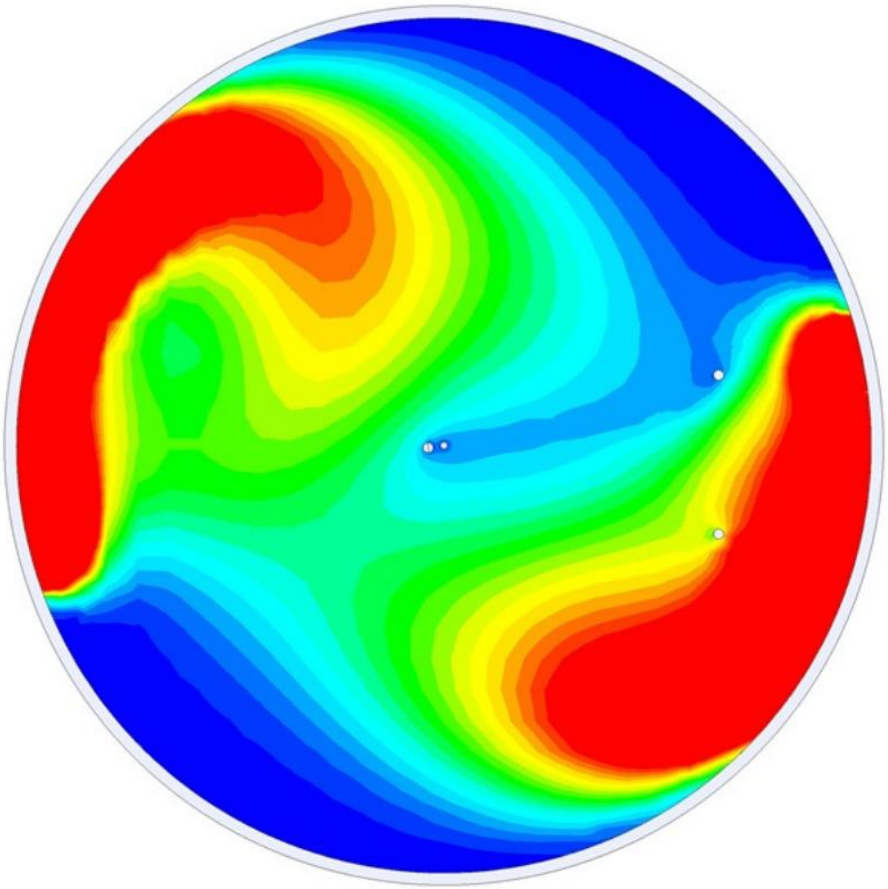
MUNIER Maxime, PLAUT Martin

Simulation d’écoulements à surface libre avec des particules



Introduction

Dans les années 60, l'aéronautique est encore une science en pleine ébauche : à des vitesses proches de celle du son, les avions traditionnels subissent des ondes de choc sur les ailes, ce qui augmente considérablement la traînée et réduit l'efficacité aérodynamique. La NASA se propose donc de concevoir une aile supercritique permettant de retarder au maximum ces ondes de choc.

Les essais en soufflerie, étant traditionnellement utilisés pour tester de nouvelles conceptions d'ailes, sont cependant trop coûteux et chronophages. Avec l'avènement des ordinateurs et des méthodes de calcul numérique, les ingénieurs commencent à utiliser des simulations pour modéliser l'écoulement de l'air autour des ailes : il est désormais possible de tester rapidement de nombreuses variantes de la forme d'une aile et de choisir celles qui offrent les meilleures performances en termes de réduction de la traînée et d'amélioration de la portance.

F-8A Crusader – www.nasa.gov

Depuis, cette technologie a été largement adoptée dans l'industrie aéronautique, et de nos jours est universellement reconnue dans tous les autres domaines scientifiques. Son étude est donc tout indiquée dans le cadre de la formation d'ingénieur à l'ENSMM.

Le projet a pour objectif de modéliser l'écoulement d'un fluide en deux dimensions, dans une interface Java. Comme toute approche numérique, il s'agira de discrétiser notre système. On choisit d'étudier un ensemble de particules, nommé « Nuage », qui va représenter le fluide.

Partie I : Construction de la classe Particule

Tout d'abord, régissons le comportement d'une unique particule. Définie par sa **masse** $\boldsymbol{m}$ et son **rayon** $\boldsymbol{h}$, elle va évoluer librement dans un cadre défini par des bords. On crée une classe, nommée Particule, dont le diagramme UML est donné ci-dessous :

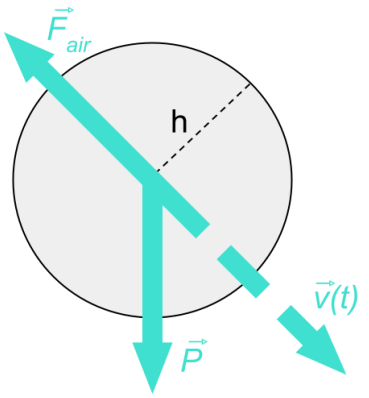
Particule	
Attributs	
- Masse m - Rayon h	- Position $r = (r_x\ ,\ r_y)$ - Vitesse $v = (v_x\ ,\ v_y)$ - Accélération $a = (a_x\ ,\ a_y)$
Méthodes	
+ *Getters* et *Setters* + calculAcceleration() + integrationEuler(τ) + gestionCollisionsBord()	+ distance(Particule p2) + energie() + rendu(Graphics 2D contexte)

Diagramme UML de la classe Particule

Pour suivre son mouvement, il nous faut sa position en chaque instant. Pour cela, on effectue un bilan des actions mécaniques à la particule considérée, en vue d'utiliser le principe fondamental de la dynamique.

Bilan des forces appliqué à la particule :

- Poids : $\vec{P} = mg.\vec{y}$
- Frottement visqueux avec l'air : $\overrightarrow{f_{air}} = -\beta.\vec{v}$



Le principe fondamental de la dynamique donne :

$$\vec{a} = \frac{1}{m}\sum \vec{F}$$

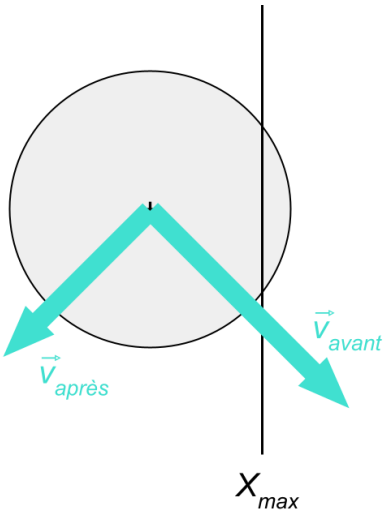
Comme l'accélération dépend de la vitesse, nous avons affaire à un classique système du deuxième ordre. Ne pouvant pas le résoudre à l'aide des équations horaires dès que le nombre de particules va augmenter (problème à 3 corps), on utilise la **méthode d'Euler** pour la résolution. En notant τ le pas d'intégration, on peut écrire :

$$\overrightarrow{v(t+\tau)} = \overrightarrow{v(t)} + \tau\overrightarrow{a(t)} \qquad \text{ainsi que} \qquad \overrightarrow{r(t+\tau)} = \overrightarrow{r(t)} + \tau\overrightarrow{v(t)}$$

Le choix de τ sera explicité plus tard.

Il s'agit aussi de gérer les conditions aux limites physiques. Notre fenêtre de travail est délimitée par des bords dont il convient de gérer les rebonds. La condition à regarder à chaque pas d'intégration est simple : si la particule dépasse d'un bord, on la fait « rebondir ».

Comme une image vaut mille mots, regardons l'exemple ci-contre :



Si la position de la particule, ajoutée à son rayon, est plus grande que x_{max}, alors on change la vitesse de la particule selon l'axe x :

$$\overrightarrow{v_{après}} = \alpha(-v_{avant,x}\ , v_{avant,y})$$

Avec α un **coefficient de réflexion** modélisant la perte d'énergie

Ce modèle, bien que physiquement abusivement erroné, donne des résultats très réalistes.

Voyons en détail le rôle des différentes méthodes de la classe Particule :

- *integrationEuler()* permet de calculer à chaque pas d'intégration (τ) la vitesse et la position à partir de la méthode expliquée plus haut
- *calculAcceleration()* initialise l'accélération en divisant les différentes forces (poids et frottement de l'air) par la masse
- *gestionCollisionBord()* effectue l'algorithme de détection des chocs avec les limites physiques de la fenêtre
- *distance()* permet le calcul de la distance cartésienne entre deux particules
- *energie()* calcule l'énergie cinétique d'une particule
- *rendu()* affiche l'interface graphique

Partie II : Construction de la classe Nuage

Dans cette partie, on souhaite simuler plusieurs particules. L'ensemble de ces particules, nommé Nuage, possède sa classe propre, dont la modélisation UML est :

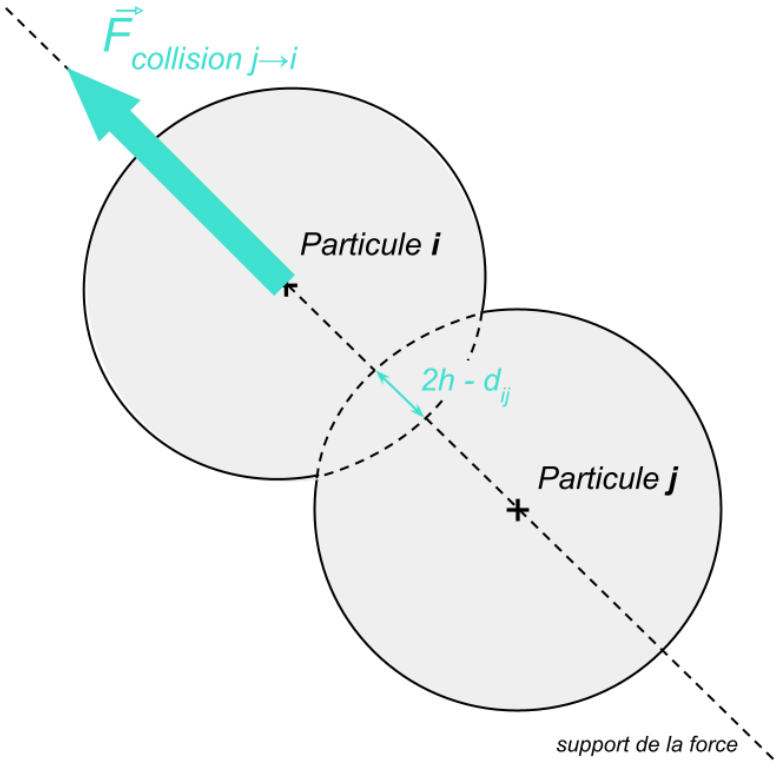
Nuage	
Attributs	
- listeParticule : ArrayList	
Méthodes	
+ gestionCollision + ajouterParticule(Particule)	+ gestionCollisionEntreParticule() + majNuage() + renduNuage()

Diagramme UML de la classe Nuage

Pour gérer les collisions entre celles-ci, on utilise le **modèle ressort** : chaque particule possède une force de répulsion de nature électrostatique qui est proportionnelle à la distance avec les autres particules. Dans le bilan des forces, il suffit donc d'ajouter une composante : supposons que la particule ***i*** entre en collision avec la particule ***j***. Elle va subir de cette dernière la force suivante :

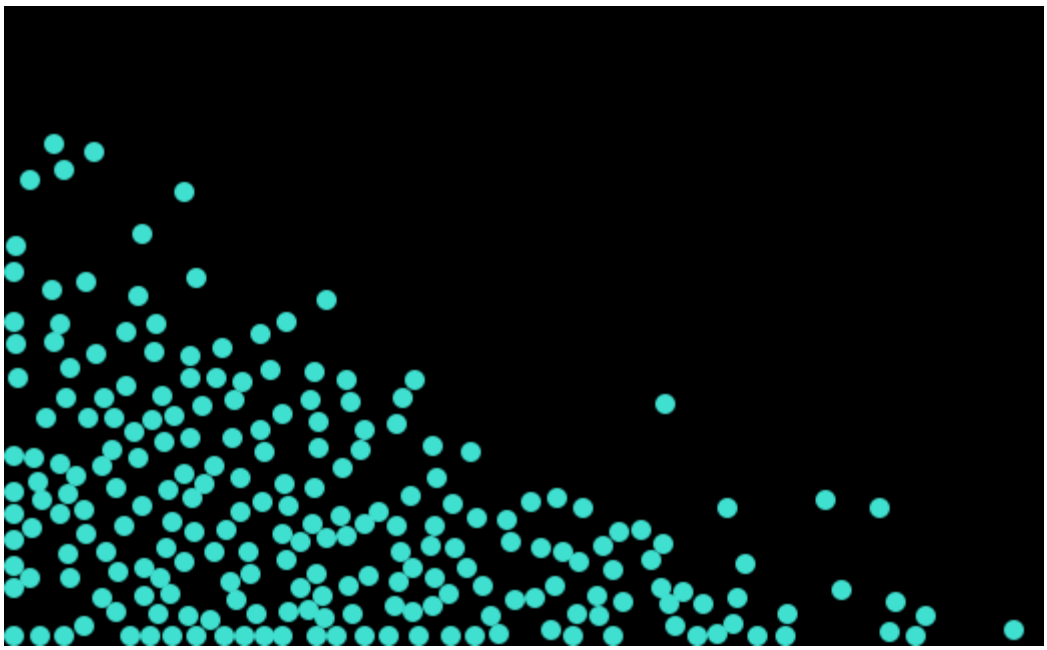
$$\overrightarrow{f_{collision}} = -k(2h - d_{ij}).\overrightarrow{u_{ij}} \text{ à condition que } d_{ij} < 2h$$

*(Par le principe des actions réciproques, la particule **j** subira la même force, de sens opposé)*



On se rend alors compte que le choix de τ est crucial au bon fonctionnement du programme ! S'il est trop grand : entre deux pas d'intégration, les particules peuvent se rapprocher « trop » et ainsi être soumises à une force de collision beaucoup trop importante. On aurait alors de la création d'énergie ainsi qu'une divergence du système, ce qui n'est pas souhaitable. A l'inverse, si τ est trop petit, le programme sera lent.

Pour choisir τ, on se base sur la célérité du son dans le milieu considéré, le son permettant de modéliser l'onde de choc. On limite c à $10v_{max}$, avec $v_{max} = \sqrt{2gH}$ (vitesse d'une particule en chute libre de haut en bas de la fenêtre). Pour que l'onde sonore n'évolue pas « trop » vite, on limite donc τ à h/c.



Rendu graphique d'un nuage de particules

Partie III : Amélioration de la classe Nuage

Si l'on analyse les données plus en profondeur, on se rend compte que la complexité est élevée : pour n particules, il y a $O(n^2)$ calculs à chaque pas d'intégration, ce qui limite fortement notre simulation. Cela provient des doubles boucles, notamment dans la gestion de collision qui étudie, pour une particule donnée, chaque particule du nuage. Or on se rend bien vite compte que la particule en question est en contact avec une portion infime du nuage : un grand nombre de calculs ne sert à rien ! Pour y remédier, on utilise une table de hachage : celle-ci combine les avantages d'un tableau trié (**recherche** efficace) et d'un tableau non trié (**insertion** efficace). Dorénavant, on aura seulement à étudier un cadre très restreint de l'interface.

Il s'agit de séparer notre cadre en blocs, mais sur quel(s) critère(s) ? La taille du quadrillage est essentielle : si nos cases sont trop grandes, le quadrillage ne sert à rien ! On constate qu'une case de côté $2h$ est un bon compromis : (rigoureusement, la complexité devient $O(n.\ln(n))$, ce qui n'est pas négligeable :

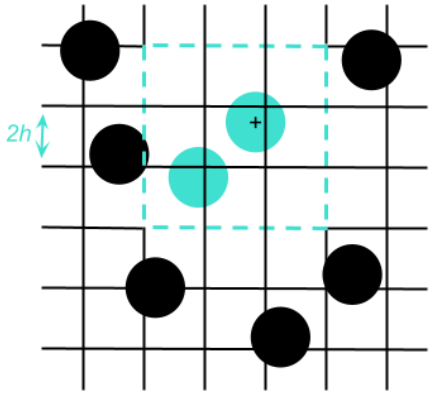
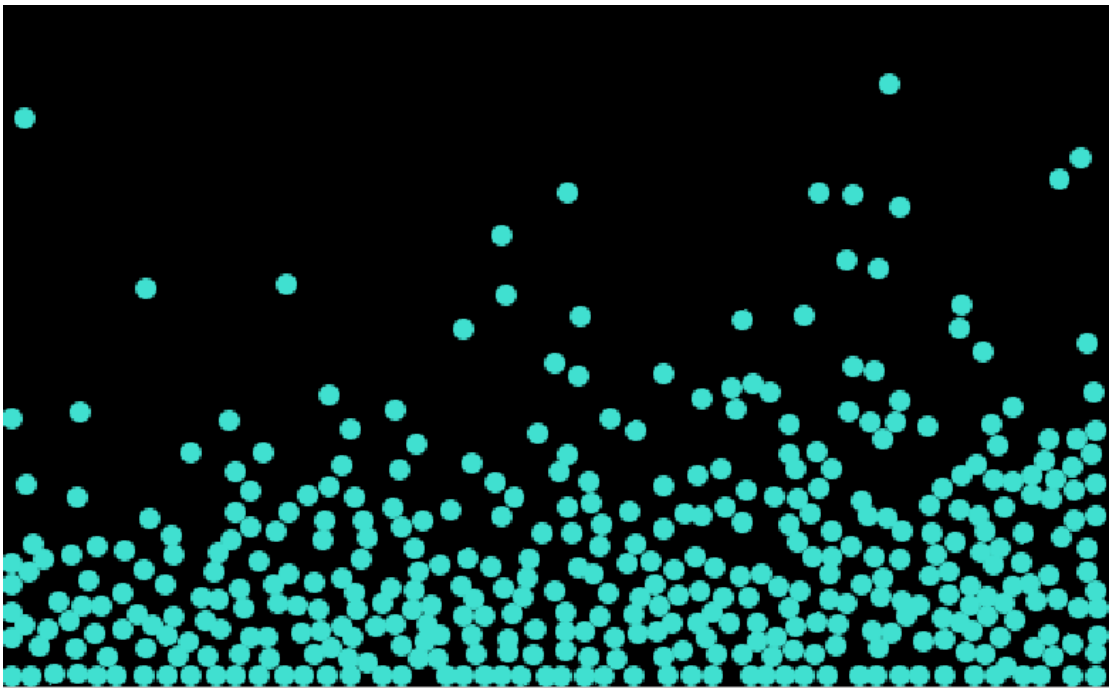


Schéma explicatif de la table de hachage

Notre nouveau nuage possède les mêmes méthodes que l'ancien, mais elles sont légèrement modifiées, notamment pour la gestion de collision. Détaillons le déroulé de cette méthode pour mettre en lumière l'intérêt de la table de hachage :

1 – On commence par parcourir toutes les cases de l'interface
2 – Pour chaque particule appartenant à la i-ème case :
- On récupère le hashcode de la case
- On récupère les **particules des cases voisines** avec une double boucle et on les stocke dans une ArrayList
- On applique l'algorithme classique de gestion de collisions entre la particule et celles de la liste.

Cela présente un avantage de complexité énorme car notre ArrayList ne contient que quelques particules (en pratique, une dizaine au maximum) ! On peut augmenter sans problème le nombre de particules. Ci-dessous, il y en a 400, et le programme fonctionne parfaitement.



Rendu graphique du Nuage Hashé

Partie IV : Amélioration des collisions

Afin d'affiner le modèle de collision, on implémente la méthode SPH : *Smoothed-Particle Hydrodynamics*. La densité du fluide dans sur un contour fermé Γ s'exprime physiquement sous la forme suivante :

$$\rho(\Gamma) = \sum_{i \in \Gamma} \frac{m_i}{\pi h^2}$$

Cependant, le nombre de particules étant fini à l'intérieur du contour, notre fonction densité n'est pas continue, ce qui va poser problème lors des calculs qui suivront. On va donc lisser notre fonction en lui appliquant un filtre passe-bas : cela revient à convoluer ρ avec la fonction de lissage W définie ci-dessous :

$$W(d,h) = \frac{1}{S}\begin{cases} \frac{1}{6}\left(2 - \frac{d}{h}\right)^3 - \frac{4}{6}\left(1 - \frac{d}{h}\right)^3 & si\ 0 \le d \le h \\ \frac{1}{6}\left(2 - \frac{d}{h}\right)^3 & si\ h \le d \le 2h \\ 0 & si\ 2h \le d \end{cases}$$

Avec d la distance entre 2 particules et $S = \frac{14}{30}\pi h^2$ une constante dépendant de la dimension du problème. On peut noter que la fonction est nulle au-delà de $2h$. Cela permet d'avoir un support borné (ce qui n'est normalement pas le cas pour une gaussienne) et ainsi de limiter le temps de calcul. Notre fonction densité appliquée à une particule i devient :

$$\rho_i = \sum_j m_j W_{ij}$$

Dans notre programme, pour chaque particule i, cela revient à :

- regarder à quelle case appartient la particule
- parcourir les 8 cases autour ainsi que celle-ci
- ajouter à une liste vide toutes les particules j appartenant à ces cases
- calculer la densité de la particule i

On peut ensuite calculer la pression qui s'applique sur une particule :

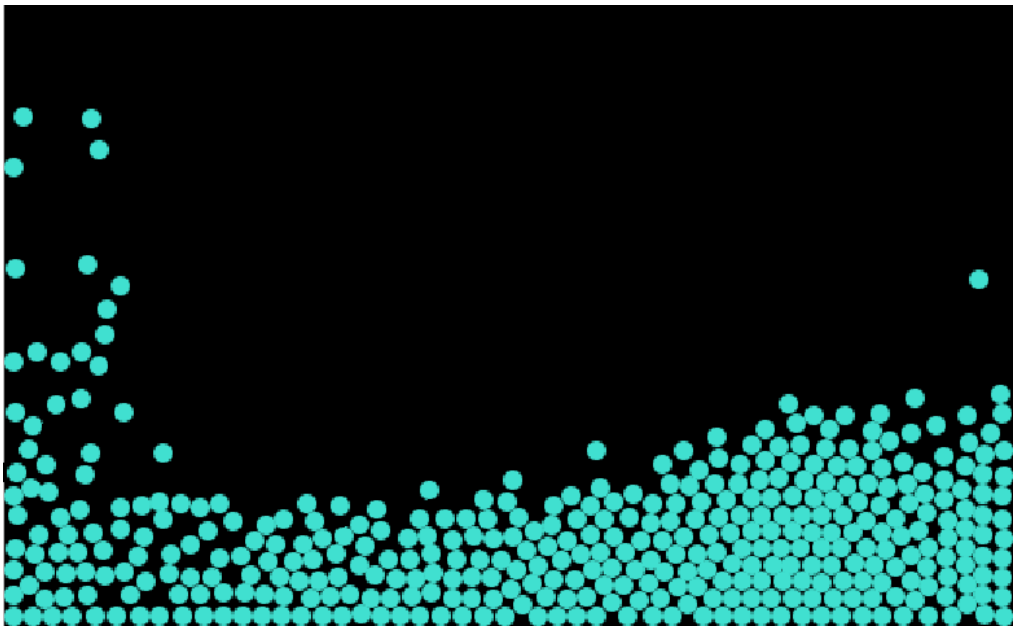
$$P_i = \frac{\rho_0 c^2}{\gamma}\left(\left(\frac{\rho_i}{\rho_0}\right)^\gamma - 1\right)$$

> Que signifie physiquement ρ_0 ? C'est la densité de référence du fluide, c'est-à-dire sa densité maximale : lorsque les particules sont agencées de la manière la plus compacte possible. Elle vaut $\frac{m}{\pi h^2}$.

L'accélération résultante s'écrit :

$$\vec{a} = \vec{g} - \sum_j m_j \left(\frac{P_i}{{\rho_i}^2} + \frac{P_j}{{\rho_j}^2}\right) W'_{ij}.\overrightarrow{u_{ij}}$$

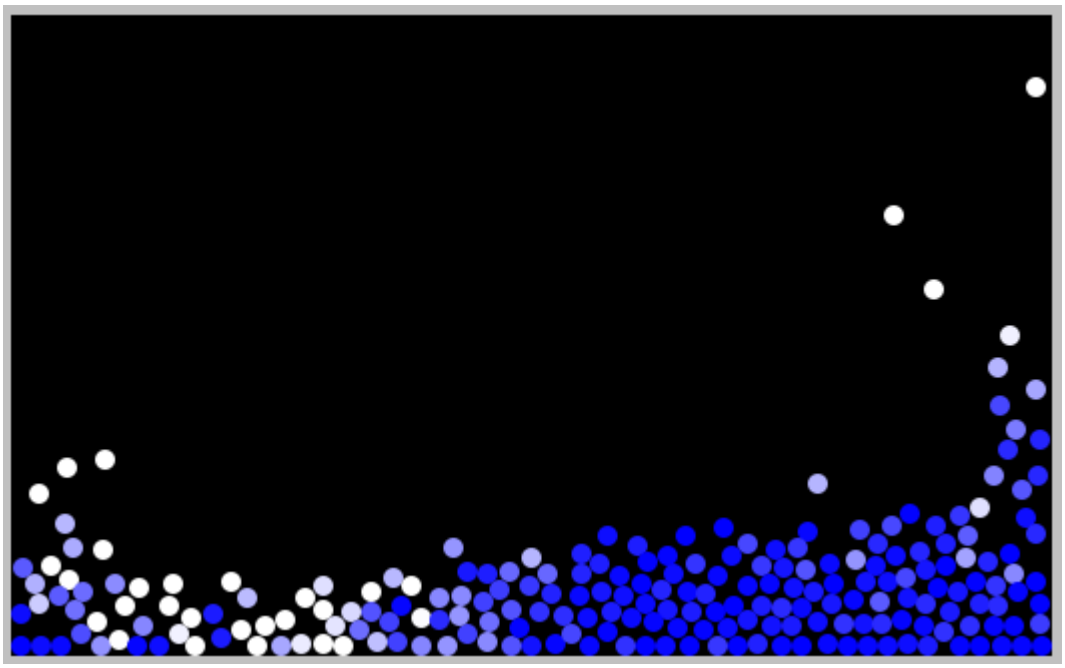
Nous avons également ajouté la force de viscosité. Le rendu est impressionnant : beaucoup plus fluide et réaliste que précédemment. On peut désormais vraiment identifier et reconnaître des motifs comme des vagues et des courants.



Rendu graphique du nuage avec la méthode *SPH*

Améliorations complémentaires

En vue de rendre le projet plus interactif et ludique, on intègre deux fonctionalités. Tout d'abord, de la couleur : chaque particule voit sa couleur varier en fonction de l'énergie cinétique (et donc de la vitesse). L'utilisation de Colormap étant très intuitive, on peut produire facilement de tels résultats, très visuels :



Rendu graphique du nuage avec la méthode *SPH*, en couleur

L’implantation d’interfaces interactives permet également à l’utilisateur de jouer avec les paramètres de la simulation. Pour garantir le suspens, les différentes possibilités ne seront pas détaillées : à vous de jouer !

Interface graphique interactive

Conclusion

Au cours de ce projet, nous avons pu nous familiariser avec Java en modélisant l'écoulement d'un fluide. Nous avons discrétisé notre système sous forme de particules (qu'il a fallu contraindre) puis nous les avons confronté entre elles, dans un nuage. Les collisions entre les particules ont été dans un premier temps rudimentaires, modélisées par un ressort. L'amélioration de ce nuage à l'aide d'une table de hachage était une expérience enrichissante vis-à-vis de la complexité algorithmique. Enfin, nous avons pu affiner notre modèle de collisions avec une méthode plus réaliste.

Cependant, le programme mériterait quelques améliorations :

- On peut encore chercher à perfectionner le modèle de collisions pour coller au mieux à la réalité. Il existe par exemple la méthode de transport de scalaires passifs. Une des limites du modèle *SPH* est par exemple la divergence de la fonction densité lors de la génération de particules : si une particule est créée trop proche d'une autre, la distance étant très faible, le programme s'autodétruit.
- Explorer les possibilités concrètes : tester l'impact de la forme d'une aile d'avion (pour en revenir à l'introduction), vérifier numériquement des lois théoriques (Hagen-Poiseuille, Torricelli, ...), simuler le ballotement d'un liquide dans un camion-citerne en vue d'étudier les effets d'inertie, . . .

« L'harmonieux éther, dans ses vagues d'azur, enveloppe les monts d'un fluide plus pur »
