

Projet de Simulateur de Fluide

Stephane LEJEUNE, Jacques PHAM BA NIEN

9 mai 2025

Table des matières

1	Introduction	3
2	Théorie de simulateur de fluide	4
2.1	Equation de Navier-Stokes	4
2.2	Simulateurs	6
2.2.1	Simulateur Newtonien	7
2.2.2	Simulateur Lagrangien	10
2.2.3	Simulateur Mixtes	11
2.2.4	Simulateur SPH (Smoothed Paricle Hydrodynamics) . .	11
3	Le projet	15
3.1	Pre-Unity : Le choix de rust	15
3.2	Post-Rust : Unity	17
3.2.1	Architecture générale du projet	17
3.2.2	Données simulées	17
3.2.3	Initialisation des paricules	18
3.2.4	Étapes du simulateur	18
4	Conclusion	20

1 Introduction

Pour le projet, nous avons choisis de construire un simulateur de fluide, projet non proposé de base, mais c'est un projet que Stephane et Jacques voulaient déjà faire depuis la L2, nous avons donc proposer le projet et il a été accepté.

Étant en double licence, le style de compétences utiles qu'on peut apprendre avec un projet informatique sont différentes, le projet de simulateur de fluide nous a sembler être un bon compromis entre demandant des compétences informatiques et mathématiques.

Ce projet, nous permet d'explorer comment les techniques qu'on as appris durant différent cours analyse numériques peuvent être utiliser pour quelque chose de plus "réel" (et visuel) que résoudre un système linéaire ou des équations différentiels abstraites.

Du point de vu informatique, ce projet nous oblige à interagir avec une interface graphique, choisir nos structures pour représenter les particules (ou ne pas les représenter)

Nous sommes donc reconnaissant que notre proposition de projet ait été accepté.

L'objectif du projet est double :

1. Implémenter une simulation de fluide stable, visuellement cohérente, et interactive, capable de gérer plusieurs centaines, voire milliers de particules.
2. Intégrer cette simulation dans un moteur de jeu 2D, en l'occurrence, *Unity*, pour permettre une visualisation en temps réel avec un contrôle

sur les paramètres physiques.

Durant le développement, de nombreux défis ont été rencontrés : gestion des densités divergentes, explosions numériques liées à une mauvaise évaluation de la pression, instabilités aux frontières, ou encore des performances non optimales à partir d'un certain nombre de particules. Plusieurs améliorations progressives ont été apportées, incluant l'ajout d'une grille spatiale, la prédiction de position, un modèle de pression stable (Tait), et une modélisation simplifiée de la viscosité.

Ce rapport détaille les fondements physiques du modèles SPH, la manière dont il a été implémenté, optimisé et visualisé dans Unity, ainsi que les résultats obtenus, leurs limites et des pistes d'améliorations futures.

2 Théorie de simulateur de fluide

2.1 Equation de Navier-Stokes

Pour comprendre comment un simulateur de fluide marchent, il nous as d'abord fallu comprendre comment un fluide est sensé se comporter.

On as certes un modèle en nous de comment cela fonctionne, on a déjà vu des fluides, mais transmettre cette intuition en instructions est loin d'être facile, cela n'est pas notre travail, c'est celui des physiciens. Nous avons donc regarder à comment les physiciens décrivent les lois que les fluides doivent respecter.

Déjà, un “fluide” en physique ne décrit pas seulement le comportement d'un liquide, cela décrit aussi le comportement des gas, leurs comportement sont étudier dans la “mécanique des fluides”.

Ses lois sont appelées “équations de Navier-Stokes” :

1. Équation de continuité :

$$\nabla \cdot \vec{V} = 0$$

2. Équation de moments :

$$\rho \left(\frac{\partial \vec{V}}{\partial t} + \vec{V} \cdot \nabla \vec{V} \right) = -\nabla p + \mu \nabla^2 \vec{V} + \vec{F}$$

L'équation de continuité nous dit que l'énergie mécanique du fluide ne change pas avec le temps (sa dérivée est nulle).

L'équation de moments est plus complexe, elle décrit les forces locales qui sont exercées sur notre fluide, décomposons ce que veut dire chaque terme :

- ρ est la densité (locale, mais cela n'importe peu pour la suite) de notre fluide, donc c'est la masse divisée par le volume autour d'un point.
- \vec{F} est la force externe exercée sur notre fluide, ici c'est juste la gravité. La force exercée par la gravité localement est $\vec{g} \frac{m}{V}$ où \vec{g} est la force gravitationnelle universelle, m est la masse et V est le volume. Autrement dit :

$$\vec{F} = \rho \vec{g}$$

- \vec{V} est le champ vectoriel de la vitesse du fluide. Champ vectoriel car en tout point, le vecteur direction peut être différent.
- p est la pression du fluide.

- μ est une constante propre au fluide étudié, cela modèle la viscosité, plus elle est grande, plus le fluide est visqueux, ce terme limite l'accélération.

Cette équation nous dit donc que le changement de vitesse locale est dû aux forces extérieures, et à la différences de vitesse environnantes, en évitant de trop se compresser, et n'accélération pas trop vite proportionnellement à la constante de viscosité.

Tout du long, nous allons simplifier notre fluide étudié, en général, les fluides dont on s'intéresse (surtout l'eau) ne sont pas visqueux (nous rajouterons la viscosité plus tard pour des raisons de précision numériques, mais ignorons ce paramètre pour l'instant), et aussi (quasiment) incompressible, cela simplifie nos équations, ce qui rends la simulation plus simple et plus rapide, nos nouvelles équations :

- Conservation :

$$\nabla \cdot \vec{V} = 0$$

- Incompressible :

$$\nabla p = 0$$

- Moments :

$$\frac{\partial \vec{V}}{\partial t} = \vec{g} - \vec{V} \cdot \nabla \vec{V}$$

2.2 Simulateurs

Certes maintenant on a “comment les fluides doivent agir”, on n’a pas “comment les faire agir comme tel”.

Pour observer la vie réel, cela n'a pas d'importance, pour un mathématicien non plus, mais un ordinateur ne peut pas gérer des équations différentiels de manière exactes sur un volume continu, il y aurait juste beaucoup trop de données à stocker et de calcul à faire (même, une infinité non dénombrable, d'où cette incapacité).

Ainsi, on est obligé de simplifier notre tâche, au lieu d'obtenir un système agissant *exactement* comme un fluide idéal, il faut faire un système qui agit *approximativement* comme un fluide idéal, les physiciens font déjà cela pour simplifier les calculs (nous ne faisons pas en général de la théorie des probabilités et des équations depuis le quantique pour des fluides, car il y aurait trop de termes agissant entre eux, obscurcissant les calculs, et même pourrait rendre des équations insolubles de manière numériques), en informatique, nous allons passer du continu au discret, perdant ainsi de la précision pour en échange, avoir de la possibilité et de la vitesse de calcul.

Il y a plusieurs manières de discrétiser notre problème :

2.2.1 Simulateur Newtonien

On peut discrétiser notre problème en considérant d'avoir une grille de vecteurs au lieu d'un champ, nous appelons chaque élément de la grille une "boîte" pour des raisons de facilité de visualisation.

Ainsi, il faut modéliser la quantité de fluide passant d'une boîte à ses boîtes voisines.

Nous avons seulement besoin de modéliser le passage de fluide d'une boîte à ses voisines directes dans les directions de gauche, droite, haut et bas, pas besoin des diagonales.

Ainsi, une optimisation est de sauvegarder les magnitudes de ces flux au lieu de la magnitude et direction du flux partant du centre de la boîte.

En pseudocode, le calcul du flux ressemble à ça :

Algorithm 1 : Simulation de Newton

Entrée : grille

Sortie : Même grille

- 1 **for** *boîte dans grille* **do**
 - 2 | Ajout flux de boîte à la boîte en dessous ;
 - 3 Mettre_*pression_à_zero*(grille) ;
-

Pour fixer la pression (faire que le flux en entrée et en sortie de toutes boîtes est à 0), une approche (Gauss-Seidel) est de fixer la pression de chaque boîte un par un, cela va défixer nos boîtes fixées auparavant, mais en répétant cette opération, on converge vers la bonne solution (car la matrice associée à une diagonale dominante, mais ici la démonstration importe peu).

Appelons $s(i, j)$ la “facilité” à pousser le flux de i vers j (par exemple, peut être mis à 0 pour représenter que j est un mur, permet aussi de mettre en contact des fluides plus dense, etc.), on peut alors faire :

Pour des raisons de performances, si il n’y a pas beaucoup de flux, il est également possible de mettre plusieurs boîtes ensemble, de manière à ne plus avoir de grille, et se focaliser sur les endroits où il y a le plus d’actions (ce sont les endroits les plus importants niveau comportement), il faudra alors changer le s , une autre raison pour laquelle avoir s est pratique.

Algorithm 2 : Mettre pression à zero Newton

Entrée : grille, nombreÉtapes et s

Sortie : Même grille

```
1 étape = 0;  
2 while étape < nombreÉtapes do  
3   for boîte dans grille do  
4      $d = 0;$   
5      $\text{flux} = 0;$   
6     for voisin dans voisins de boîte do  
7        $d = d + s(\text{boîte}, \text{voisin});$   
8        $\text{flux} = \text{flux} + \text{flux\_entre}(\text{boîte}, \text{voisin}) * s(\text{boîte}, \text{voisin});$   
9     for voisin dans voisins de boîte do  
10       $\text{flux\_entre}(\text{boîte}, \text{voisin}) -= \text{flux} * s(\text{boîte}, \text{voisin}) / d;$ 
```

2.2.2 Simulateur Lagrangien

Une autre approche est de discretiser notre problème en une manière plus intuitive, en simulant nos *particules* d’eaux, néanmoins, dans un fluide usuel, il y a beaucoup trop de particules, donc à la place, on va simuler des très grosses particules, ce qui est absurde physiquement mais marche relativement bien.

L’approche Lagrangienne permet de ne pas avoir à considérer que tout notre environnement est dans un fluide, par exemple ne pas avoir à considérer à la fois l’eau *et* l’air qui l’entour.

Cette approche est plus proche de comment on intuitionne les fluides, mais plus loin du calcul mathématique que nous avons dérivé, il faut donc travailler avec des calculs plus complexes pour obtenir des valeurs tel que la vitesse, la température ou même la pression, ce problème de pression impacte de manière directe les performances, car on a généralement besoin de cette valeur pour simuler un fluide (à moins qu’on ait un grain très fin, alors les collisions entre particules génèrent d’elles même avec haute probabilités, des interactions faisant penser à celle de la densité, en effet, un “fluide” n’est qu’un amas de beaucoup de particules interagissant localement entre elles, notre comportement idéal n’est que le résultat probabiliste moyen que ces interactions créent à grande échelle, sujet étudié en Physique Statistique).

Les soucis d’implémentation d’une méthode Lagrangienne sera expliquer plus tard (le simulateur SPH (Smoothed Particle Hydrodynamics)) en détail, nous laissons donc cette sous section sans.

Néanmoins, ce qu’une approche Lagrangienne permet de faire, est de prendre un simulateur physique d’interaction de corps usuel, et directement avoir nos fluides “gratuitement”.

Cela permet aussi de l'étendre plus facilement, en effet, nous avons déjà l'interaction entre petites particules, avec moins d'effort qu'avec une méthode Eulerienne, nous pouvons rajouter l'interaction de ces particules sur des objets de grande tailles.

2.2.3 Simulateur Mixtes

Tout comme vous avez pus le remarquer, les approches Newtoniennes et Lagrangiennes sont très différentes, et ont des bénéfices et des inconvénients.

Ce qui laisse la possibilité de mixer ces deux approches pour essayer de combler les inconvénients de chaques méthodes par l'autre.

Néanmoins, cette approche demande de gros efforts d'implémentation, beaucoup de test car il y a beaucoup plus de facteurs mouvants dans le système, pour des effets sur la performances qu'on peut obtenir en choisissant un système et l'optimisation bien avec le même effort (mais un système mixte et optimisé pourrait être plus rapide).

2.2.4 Simulateur SPH (Smoothed Particle Hydrodynamics)

C'est le simulateur que nous avons décider d'implémenté, nous avons d'abord penché sur une approche Newtonienne pour des raisons de simplicité, mais cette approche nous permet de plus de choses, sans non plus être incroyablement complexe niveau théorique ou pratique.

Représentation des particules Chaque particules sont caractérisées par :

- un vecteur position \vec{r} (le p est déjà pris pour la pression)
- un vecteur vitesse \vec{v}

- une densité ρ
- une pression p

Chaque particule possède également une masse m partagé avec les autres particules, choisi de manière à respecter la masse total du fluide.

Fonction de lissage (Kernel) L'originalité de SPH repose sur l'utilisation d'un noyau de lissage $W(r, h)$, une fonction radiale (ne dépendant seulement de la distance) pondérant l'influence d'une particule en fonction de la distance $r = \|\vec{r}_i - \vec{r}_j\|$ entre les particules i et j (on impose aussi $\vec{r}_i \neq \vec{r}_j$), définit par :

$$W(r, h) = \begin{cases} \frac{4}{\pi h^3} (h - r)^2 & \text{si } r < h \\ 0 & \text{sinon} \end{cases}$$

On a que pour n'importe quel grandeur physique A sur un point (n'étant pas forcément une particule), on peut l'approximer par :

$$A(\vec{r}) = \sum_i m_i \frac{A_i}{\rho_i} W(\|\vec{r} - \vec{r}_i\|, h)$$

où A_i est cette grandeur à la particule i .

La dérivé spatiale est également utilisée pour calculer les forces de pression, avec $r < h$:

$$\nabla W(r, h) = -\frac{8}{\pi h^3}(h-r)\frac{\vec{r}}{r}$$

Calcul de densité La densité locale ρ_i est estimée à partir des particules voisines :

$$\rho_i = \sum_j m_j W(\|\vec{r}_i - \vec{r}_j\|, h)$$

Ce qui fait sens physiquement, une masse pondérée par le volume du voisinage.

Modèle de pression - Équation de Tait Afin de stabiliser et d'éviter les oscillations numériques, une équation d'état non-linéaire est utilisée :

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

avec :

- ρ_0 : la densité au repos (exemple : 1000kg/m³ pour l'eau)
- k : constante de raideur (pressionCoefficient)
- γ : exposant (souvent entre 5 et 7, ici fixé à 7)

Ce modèle évite les pressions négatives et assure une réponse plus fluide à la compression.

On pourrait se dire puisque notre fluide est “incompressible” que il ne devrait pas avoir de différence entre ρ_0 et ρ_i , mais nous faisons une simulations, au long terme le fluide agi de manière incompressible en agissant contre la pression, mais ce n’est pas instantané.

Force de pression La force de pression exercée sur une particule i est calculée par

$$\vec{f}_i^{\text{pression}} = - \sum_j m_j \left(\frac{P_i + P_j}{2\rho_j} \right) \nabla W(\|\vec{r}_i - \vec{r}_j\|, h)$$

Ce terme assure la répulsion entre les paricules trop proches et la cohésion du fluide.

Viscosité Un terme de viscosité artificielle permet de modéliser l’amortissement des vitesses relatives et d’éviter les artéfacts numériques, ce terme est nécessaire dû à notre méthode de résolution d’équation différentiel (notre “intégrateur”), en calculant l’état suivant à partir de l’état actuel, l’énergie à tendance à se créer à cause des erreurs numériques (et en calculant l’état suivant selon comment on calculerait l’état précédent pour être celui actuel, l’énergie à tendance à se perdre, créant de soi une viscosité qu’on aurait à réduire, des approches complexes existent, tel que les intégrateurs *simpléctiques* qui intègrent en restant sur la surface d’un *simplexe*, obtenu avec les équations Hamiltoniennes de la mécanique, mais ce sujet bien qu’intéressant et source d’amélioration n’a pas sa place à être trop développer ici). Ce terme est donné par :

$$\vec{f}_i^{\text{viscosité}} = \mu \sum_j m_j \left(\frac{\vec{v}_j + \vec{v}_i}{\rho_j} \right) \nabla W(\|\vec{r}_i - \vec{r}_j\|, h)$$

où μ est le coefficient de viscosité propre au fluide.

Force externe Une force de gravité est aussi appliquer à chaque particules :

$$\vec{f}_i^{\text{gravité}} = m_i \vec{g}$$

3 Le projet

3.1 Pre-Unity : Le choix de rust

Nous (surtout Jacques) avons commencé avec de (trop) grandes ambitions sur le setup du projet.

- le faire en *rust* pour avoir des très bonnes performances.
- implémenter le build avec *nix* pour une gestion automatique de *toutes* dépendences (incluant obtenir rust, pour avoir la même version, le même système pour nous tous) avec *crane*.
- une gestions de faille de sécurités par un repot d’audit de packets, analyseur statique d’appelle insécure en temps qu’option de base d’utiliser crane!
- pouvoir faire du cross-platform, et compiler notre code rust dans le CI pour plusieurs architectures et OS.

- utiliser *nom*, une librairie rust de grandeurs physiques, qui nous permettrait d'enlever tout bug correspondant à mélanger des éléments de grandeurs différents (qui sont forcément des bugs, car dépendant du choix exacte de grandeur)

Des questions se sont posées sur comment faire une fenêtre en rust, il y a très loin qu'une seule option, les deux idées prédominantes ont été d'utiliser la librairie *bevy* ou *SDL2* (proposition de Jacques, ayant un peu d'expérience en *SDL2* dans d'autres langages (C)) ou de manière plus bas niveau, *wgpu* (proposition de Stéphane).

De manière merveilleuse, on n'aurait même pas à gérer l'installation de *SDL2* grâce à nix ! Et lier la librairie statiquement aurait fait que il n'y aurait pas eu besoin pour les utilisateurs d'installer quoi que ce soit, juste l'exécutable.

Mais il y a eu des mais, essayer de setup le cross-platform était vraiment frustrant, trop frustrant, je ne l'ai au final pas fait, il aurait fallu à mes camarades d'installer nix, et sans utiliser du caching, ça demandait trop souvent de réinstaller les dépendances, ce qui est long ! Et même sans les demandes d'installation de dépendances, il n'y avait pas le caching usuel fait par *cargo* le build system de rust de base, tout était recompilé depuis la source au lieu de compiler une fois, puis utiliser des objets compilés. Ce qui a rendu l'utilisation encore plus lente que normalement. Chaque modification prenait des minutes, ce qui tue assez rapidement toutes les joies ou volontés de programmer, donc l'utilisation de nix a été abandonnée, avec lui, les perspectives d'utiliser *SDL2*.

Nous avons tous de notre côté ensuite essayé de faire marche *bevy* ou *wgpu*, mais *bevy* utilise un modèle assez spéciale, *ECS* (Entity Component System) qui paraissait pas très naturel et n'était pas le but de ce projet, et *wgpu* était trop bas niveau.

Ce fut trop, on n'avancait pas, malgré les qualités techniques, si on ne peut pas interagir rapidement avec notre projet, il est plus difficile d'expérimenter, voir ce qui marche, etc., le choix était plus nuancé que prévu, nous avons donc abandonné rust pour avoir quelque chose qui réglait notre plus grand problème, le graphique, ainsi nous avons utilisé Unity ensuite.

3.2 Post-Rust : Unity

Ce fut plus simple.

3.2.1 Architecture générale du projet

Le projet repose sur une séparation claire des responsabilités :

- FluidSimulator.cs : logique de simulation et itération de l'algorithme SPH.
- ParticleComponent.cs : représentation individuelle d'une particule (GameObject)
- BoundsRenderer.cs : affichage des limites de la simulation
- MainController.cs (optionnel) : interface utilisateur et réglage dynamiques.

Chaque particule est associée à un GameObject contenant un *Transform* et un script *ParticleComponent*, qui est mis à jour visuellement à chaque frame à partir des données simulées.

3.2.2 Données simulées

Trois tableaux principaux contiennent les données physiques :

- positions[] : positions des particules

- velocities[] : vitesses
- densities[] : densités locales calculées à chaque pas de simulation

Un tableau de particles[] (struct contenant position/velocity/density) centralise l'état du système. Les GameObjects sont ensuite synchronisés avec ces données.

3.2.3 Initialisation des paricules

L'initialisation peut se faire de manière :

- aléatoire : dispersion uniforme dans la boîte
- en grille : positionnement ordonné (recommandé pour stabilité initial)

Un prefab de particule est instancié pour chaque entité simulée. Le particleCount et particleSpacing sont ajustables à l'exécution.

3.2.4 Étapes du simulateur

Chaque frame effectue les étapes suivantes :

1. Application de la gravité (accélération constante sur les vitesses)
2. Prédiction des positions à partir des vitesses
3. Mise à jour de la grille spatiale (accélération du voisinage)
4. Calcul des pressions (Tait) et forces de pression
5. Ajout d'une force de viscosité pour stabiliser les interactions
6. Mise à jour des vitesses en fonction des accélérations
7. Correction de la position et résolution des collisions (rebond et amorçements sur les bords)

8. Mise à jour ds GameObjects Unity pour la visualisation

Grille spatiale Afin de limiter la complexité du voisinage à $O(n)O(n)O(n)$ plutôt que $O(n^2)O(n^2)O(n^2)$, une grille spatiale 2D est utilisée. Les particules sont associées à des cellules de taille égale au rayon de lissage h . Chaque cellule contient une liste d'indices de particules.

Lors du calcul des interactions, seules les cellules voisines 333×333 sont parcourues, ce qui améliore significativement les performances.

Résolution des collisions Le système est boné par un rectangle de dimensions fixes (ex : 20x15 unités). Chaque particule est testée par rapport à ces bornes :

- Si elle sort, sa position est corrigée à la limite
- Sa vitesse est inversée et amortie par un facteur dampening Factor.

Cela permet de contenir le fluide tout en simulant un rebond partiel.

Visualisation Chaque particule possède un SpriteRenderer, dont la couleur peut être modifiée dynamiquement pour refléter la densité ou la pression.

Les bords de la simulation sont affichés à l'aide d'un LineRenderer, utilisant un matériau non lumineux (Sprites/Default) pour être bien visible en mode 2D.

Paramètres ajustables Les constantes suivantes sont exposées dans l'éditeur pour permettre des expérimentations en direct :

- Nombre de particules
- Taille de la boîte
- Rayon de lissage h
- Coefficient de pression
- Viscosité
- Gravité
- Facteur d'amortissement

Performances Le simulateur est capable de gérer jusqu'à *1000 particules* à environ *100-120 FPS* sans multithreading ni calcul GPU. L'utilisation d'une grille spatiale et la simplification des formules mathématiques permettent une exécution acceptable sur le CPU.

4 Conclusion

TODO