Implémentation en C++ de la méthode Smoothed Particle Hydrodynamics

Estimations à postériori

Sacha Cardonna

16 octobre 2022

Dans ce projet on s'intéresse à la simulation de fluide sur ordinateur ; l'objectif y est de générer des animations réalistes de fluides liquides et gazeux. La plupart de ces techniques de simulation sont simplement des approximations numériques des équations de *Navier-Stokes*, qui décrivent le mouvement des fluides. On rappelle que celles-ci s'écrivent, avec contraintes d'incompressibilité, comme suit

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right) = -\nabla p + \eta\boldsymbol{\nabla}^2\boldsymbol{u} + \rho\boldsymbol{g},$$
$$\boldsymbol{\nabla}\cdot\boldsymbol{u} = 0,$$

où ρ est la masse volumique du fluide, $\boldsymbol{u}$ le champ vectoriel de vitesse, p la pression, et $\boldsymbol{g}$ un vecteur de forces externes (par exemple la gravité). Il existe deux approches principales pour approximer numériquement ces équations (plus généralement, tout problème impliquant des champs d'écoulement) : les méthodes lagrangiennes "particle-based" et les méthodes eulériennes "grid-based". Dans une perspective lagrangienne, nous observons une parcelle de fluide discrétisée individuelle pendant qu'elle se déplace dans l'espace et le temps. À l'opposé, les solveurs eulériens se concentrent sur le mouvement d'un fluide à travers des emplacements spécifiques dans l'espace au fil du temps. On peut penser à modéliser ces parcelles lagrangiennes comme des particules et ces emplacements fixes eulériens dans l'espace comme des mailles.

Bien que les solveurs lagrangiens et eulériens soient largement utilisés en dynamique des fluides numérique, une propriété notable des solveurs à base de particules est qu'ils ne sont généralement pas aussi limités dans l'espace car ils ne reposent pas entièrement sur une grille de simulation eulérienne sous-jacente. Ceci, ainsi que leur relative facilité de mise en œuvre, rend les solveurs lagrangiens particulièrement bien adaptés aux applications telles que les jeux vidéo, qui impliquent souvent de grands environnements dynamiques.

L'une des méthodes les plus populaires pour la simulation des fluides à base de particules est l'hydrodynamique des particules lissées (*Smoothed Particle Hydrodynamics*), développée pour la première fois par Gingold et Monaghan en 1977 pour les simulations astrophysiques. Müller a d'abord appliqué cette même technique à la simulation de fluides en temps réel pour les jeux, et cette méthode est depuis devenu la méthode de choix pour les simulateurs avec des applications dans l'ingénierie, la prévention des catastrophes, l'animation informatique haute fidélité, les jeux vidéos, le CGI dans les blockbusters...

1. Résulats de physique newtonienne.

Dans un cadre de physique newtonienne, on peut utiliser les principes de conservation de la masse, de la quantité de mouvement et de l'énergie. Rappelons tout d'abord la définition de la dérivée matérielle ou lagrangienne comme l'opérateur non linéaire suivant :

$$\frac{D}{Dt} := \frac{\partial}{\partial t} + \boldsymbol{u}\cdot\boldsymbol{\nabla},$$

où $\boldsymbol{u}$ est la vitesse de l'écoulement. Par opposition à la dérivée eulérienne par rapport à une position fixe dans l'espace, la dérivée matérielle décrit l'évolution d'un champ suite à un déplacement. En se souvenant de la relation classique $\boldsymbol{F} = m\boldsymbol{a}$, on applique la dérivée matérielle et on obtient

$$m\frac{D\boldsymbol{u}}{Dt} = \boldsymbol{F}^{\text{total}},$$

qu'on peut étendre aux forces agissant sur la particule :

$$\rho\frac{D\boldsymbol{u}}{Dt} = \sum \boldsymbol{F} = \boldsymbol{F}^{\text{pression}} + \boldsymbol{F}^{\text{viscosité}} + \rho\boldsymbol{g}.$$

La contribution du fluide à la force totale sur chaque particule est divisée en deux parties : la pression et la viscosité, avec $\boldsymbol{g}$ représentant à nouveau toute force externe telle que la gravité. En modélisant la contribution de force de la pression comme le gradient des pressions, et la contribution de force de la viscosité proportionnelle à la divergence du gradient du champ de vitesse, nous avons

$$\rho\frac{D\boldsymbol{u}}{Dt} = \rho\left(\frac{\partial\boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\boldsymbol{\nabla}\boldsymbol{u}\right) = -\nabla p + \eta\boldsymbol{\nabla}^2\boldsymbol{u} + \rho\boldsymbol{g}.$$

Nous souhaitons modéliser des fluides incompressibles, c'est-à-dire des fluides dans lesquels la densité de matière est constante dans chaque volume infinitésimal qui se déplace avec la vitesse d'écoulement. Ainsi, on considérera uniquement Ω comme une partie arbitraire de fluide, où s'applique la conservation de la masse, donc telle que $\frac{d\,\text{Vol}(\Omega)}{dt} = 0$. Cette variation de volume peut être mesurée en intégrant la composante normale du champ de vitesse du fluide le long du bord $\partial\Omega$, et en utilisant le théorème de Gauss on obtient

$$\iint_{\partial\Omega} \boldsymbol{u}\cdot\boldsymbol{n} = 0 \Rightarrow \iiint_{\Omega} \boldsymbol{\nabla}\cdot\boldsymbol{u} = 0.$$

2. Smoothed Particle Hydrodynamics.

2.1. Introduction.

L'idée de base de cette méthode est dérivée de l'interpolation intégrale. Essentiellement, SPH est une discrétisation du fluide en éléments discrets, des particules, sur lesquelles les propriétés sont lissées par une fonction noyau. Cela signifie que les particules voisines dans le rayon de lissage affectent les propriétés d'une particule donnée, telles que les contributions de pression et de densité.

La fonction importante suivante donne une approximation d'une quantité A en un point $\boldsymbol{r}$:

$$A_s(\boldsymbol{r}) = \int A(\boldsymbol{r}')W(\boldsymbol{r}-\boldsymbol{r}',h)\,\mathrm{d}\boldsymbol{r}' \simeq \sum_j A_j\frac{m_j}{\rho_j}W(\boldsymbol{r}-\boldsymbol{r}_j,h)$$

où m_j est la masse de la $j-$ième particule, ρ_j sa densité, et où $W(\boldsymbol{r},h)$ est un noyau de lissage à symétrie radiale de longueur h tel que $W(-\boldsymbol{r},h) = W(\boldsymbol{r},h)$ et $\int W(\boldsymbol{r},h)\,\mathrm{d}r = 0$. En appliquant cela aux équations de Navier-Stokes ci-dessus, nous devons d'abord déterminer un moyen de définir la densité du fluide en fonction des particules voisines, bien que cela découle trivialement de l'approximation précédente, en remplaçant ρ par la quantité arbitraire A tel que

$$\rho_i := \rho(\boldsymbol{r}_i) = \sum_j m_j\frac{\rho_j}{\rho_j}W(\boldsymbol{r}-\boldsymbol{r}_j,h) = \sum_j m_j W(\boldsymbol{r}-\boldsymbol{r}_j,h).$$

Dû à la linéarité du gradient, on a facilement

$$\nabla A(\boldsymbol{r}) = \nabla\left(\sum_j A_j\frac{m_j}{\rho_j}W(\boldsymbol{r}-\boldsymbol{r}_j,h)\right) = \sum_j A_j\frac{m_j}{\rho_j}\nabla W(\boldsymbol{r}-\boldsymbol{r}_j,h),$$
$$\nabla^2 A(\boldsymbol{r}) = \sum_j A_j\frac{m_j}{\rho_j}\nabla W(\boldsymbol{r}-\boldsymbol{r}_j,h).$$

2.2. SPH pour Navier-Stokes.

Suite à ce qu'on a vu avec les équations de Navier-Stokes, il est évident que la somme des champs de densité de force sur le côté droit de l'équation donne la variation de quantité de mouvement $\rho\frac{D\boldsymbol{u}}{Dt}$ des particules du côté gauche de l'équation. De cela, on peut décrire l'accélération de la $i-$ème particule :

$$\boldsymbol{a}_i = \frac{\mathrm{d}\boldsymbol{u}}{\mathrm{d}t} = \frac{\boldsymbol{F}_i}{\rho_i}.$$

On cherche ainsi à approximer les termes $\boldsymbol{F}^{\text{pression}}$ et $\boldsymbol{F}^{\text{viscosité}}$ avec la méthode SPH comme suit :

$$\boldsymbol{F}_i^{\text{pression}} = -\nabla p(\boldsymbol{r}_i) = -\sum_j m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W(\boldsymbol{r}-\boldsymbol{r}_j, h),$$

$$\boldsymbol{F}_i^{\text{viscosité}} = \eta\nabla^2\boldsymbol{u}(\boldsymbol{r}_i) = \eta\sum_j m_j \left(\frac{\boldsymbol{u}_j - \boldsymbol{u}_i}{\rho_j}\right)\nabla^2 W(\boldsymbol{r}-\boldsymbol{r}_j, h).$$

La pression p au niveau d'une particule est calculé à l'aide d'une équation d'état reliant la densité à une densité de repos définie, généralement l'équation de *Tait* ou l'équation des *gaz parfaits*, telle que $p = k(\rho - \rho_0)$, où ρ_0 est la densité au repos et k une constante du gaz dépendante de la température du système. Notons que ces formulations peuvent changer selon les méthodes utilisées. Par exemple, l'application directe de SPH au terme de pression $-\nabla p$ est donnée par

$$\boldsymbol{F}_i^{\text{pression}} = -\nabla p(\boldsymbol{r}_i) = -\sum_j m_j \frac{p_j}{\rho_j}\nabla W(\boldsymbol{r}-\boldsymbol{r}_j, h),$$

qui n'est pas symétrique. Considérons le cas de deux particules, où la particule i n'utiliserait que la pression de la particule j pour calculer sa force de pression et inversement. Les pressions à deux emplacements de particules ne sont pas égales en général, donc la force n'est pas symétrique. L'équation de force de pression SPH ci-dessus utilise ce qui est devenu une symétrisation canonique (somme pondérée). Dans l'article original de Müller, la symétrisation est effectuée à l'aide d'une moyenne arithmétique :

$$\boldsymbol{F}_i^{\text{pression}} = -\nabla p(\boldsymbol{r}_i) = -\sum_j m_j \left(\frac{p_j + p_j}{2\rho_j}\right)\nabla W(\boldsymbol{r}-\boldsymbol{r}_j, h).$$

Ceci doit à nouveau être abordé dans le terme de viscosité, ce qui donne la relation asymétrique

$$\boldsymbol{F}_i^{\text{viscosité}} = \eta\sum_j m_j \frac{\boldsymbol{u}_i}{\rho_j}\nabla^2 W(\boldsymbol{r}-\boldsymbol{r}_j, h).$$

2.3. Tension superficielle.

Jusqu'à présent, nous avons donné une brève justification du Navier-Stokes à partir des principes physiques de base et avons discuté de l'utilisation de la méthode SPH comme schéma de discrétisation lagrangien, nous permettant ainsi de modéliser les forces de pression et de viscosité. Un élément manquant et pourtant crucial pour une simulation crédible est la tension superficielle.

La tension superficielle est le résultat des forces d'attraction entre les molécules voisines dans un fluide. A l'intérieur d'un fluide, ces forces attractives sont équilibrées et s'annulent, alors qu'à la surface d'un fluide, elles sont déséquilibrées, créant une force nette agissant dans la direction de la surface normale au fluide, minimisant généralement la courbure de la surface du fluide. L'amplitude de cette force dépend de l'amplitude de la courbure actuelle de la surface ainsi que d'une constante σ dépendante des deux fluides qui forment la frontière.

Les forces de tension superficielle, bien qu'elles ne soient pas présentes dans les équations incompressibles de Navier-Stokes, peuvent être explicitement modélisées de différentes manières. Encore une fois, l'approche la plus courante et peut-être la plus facile à saisir est présentée par Müller et ses collègues, qui révise les équations de Navier-Stokes en ajoutant un autre terme basé sur les principes physiques ci-dessus pour la tension superficielle :

$$\rho(\boldsymbol{r}_i)\frac{\mathrm{d}\boldsymbol{u}_i}{\mathrm{d}t} = -\nabla p(\boldsymbol{r}_i) + \eta\nabla^2\boldsymbol{u}(\boldsymbol{r}_i) + \rho(\boldsymbol{r}_i)\boldsymbol{g}(\boldsymbol{r}_i) - \sigma\nabla^2 c_s(\boldsymbol{r}_i)\frac{\nabla c_s(\boldsymbol{r}_i)}{|\nabla c_s(\boldsymbol{r}_i)|},$$

où $c_s(\boldsymbol{r})$ est un champ de couleur 1 où les particules sont présentes, et 0 partout ailleurs, tel que

$$c_s(\boldsymbol{r}) = \sum_j \frac{m_j}{\rho_j} W(\boldsymbol{r} - \boldsymbol{r}_j, h).$$

Notons que le gradient lissé du champ $\boldsymbol{n} = \nabla c_s$ donne le champ normal de surface pointant dans le fluide et la divergence de n mesure la courbure de la surface $\kappa = \frac{-\nabla^2 c_s}{\boldsymbol{n}}$. Ainsi on a la force

$$\boldsymbol{F}^{\text{surface}} = \sigma\kappa\boldsymbol{n} = -\sigma\nabla^2 c_s \frac{\boldsymbol{n}}{|\boldsymbol{n}|}$$

formée en répartissant la traction de surface entre les particules proches de la surface en multipliant un champ scalaire normalisé qui n'est non nul que près de la surface.

2.4. Fonctions noyaux.

Müller déclare que "la stabilité, la précision et la vitesse de la méthode SPH dépendent fortement du choix des noyaux de lissage". La conception des noyaux à utiliser dans SPH est un domaine de recherche actif, mais il existe un certain nombre de principes fondamentaux qui sont généralement respectés, bien illustrés par les noyaux proposés à l'origine par Müller : `poly6`, `spiky` et `viscosity`. Les noyaux avec des erreurs d'interpolation de second ordre peuvent être construits comme pairs et normalisés. Les noyaux qui sont 0 des dérivées nulles à la frontière contribuent à la stabilité. On ne donne pas explicitement les expressions de ces fonctions (données dans l'implémentation plus bas).

3. Implémentation de SPH en `C++`.

Les lecteurs avertis trouveront un aspect important de la simulation des fluides absent de cette implémentation : la tension superficielle. Alors qu'un modèle de tension superficielle de base est discuté par Müller dans son article original, la simplicité du code est grandement améliorée en se concentrant sur la modélisation directement à l'aide de formulations SPH.

Au niveau le plus élémentaire, notre code SPH doit produire une simulation en temps réel raisonnablement réaliste d'un fluide. À cette fin, notre code aura trois composants principaux : un système de rendu, un système de mise à jour (solveur SPH) et un gestionnaire d'entrées utilisateur.

3.1. Système de rendu.

`OpenGL` est la référence en matière de programmation graphique sur ordinateur. Nous utilisons un moteur de rendu très basique basé sur `GLUT`, une boîte à outils fournissant une API de système de fenêtrage multiplateforme. Voici le système de rendu qu'on a utilisé :

```
1  void InitGL(void)
2  {
3    glClearColor(1.f, 1.f, 1.f, 0); // Couleur du fond de la fenetre
4    glEnable(GL_POINT_SMOOTH);
5    glPointSize(H / 2.f);
6    glMatrixMode(GL_PROJECTION);
7  }
8
9  void Render(void)
10 {
11   glClear(GL_COLOR_BUFFER_BIT);
12
13   glLoadIdentity();
14   glOrtho(0, VIEW_WIDTH, 0, VIEW_HEIGHT, 0, 1);
15
16   glColor4f(1.0f, 0.0f, 0.0f, 0.0f); // Couleur des particules
17   glBegin(GL_POINTS);
```

```
  for (auto &p : particles)
  {
    glVertex2f(p.x(0), p.x(1));
  }
  glEnd();
  glutSwapBuffers();
}
```

3.2. Solveur SPH.

Voici le système de mise à jour écrit en utilisant la méthode SPH.

```
void InitSPH(void)
{
  for (float y = EPS; y < VIEW_HEIGHT - EPS * 2.f; y += H)
  {
    for (float x = VIEW_WIDTH / 4; x <= VIEW_WIDTH / 2; x += H)
    {
      if (particles.size() < DAM_PARTICLES)
      {
        float jitter = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        particles.push_back(Particle(x + jitter, y));
      }
      else
      {
        return;
      }
    }
  }
}

void Integrate(void)
{
  for (auto &p : particles)
  {
    // Integration par la methode d'Euler
    p.v += DT * p.f / p.rho;
    p.x += DT * p.v;

    // Conditions au bord
    if (p.x(0) - EPS < 0.f)
    {
      p.v(0) *= BOUND_DAMPING;
      p.x(0) = EPS;
    }
    if (p.x(0) + EPS > VIEW_WIDTH)
    {
      p.v(0) *= BOUND_DAMPING;
      p.x(0) = VIEW_WIDTH - EPS;
    }
    if (p.x(1) - EPS < 0.f)
    {
      p.v(1) *= BOUND_DAMPING;
      p.x(1) = EPS;
    }
    if (p.x(1) + EPS > VIEW_HEIGHT)
    {
      p.v(1) *= BOUND_DAMPING;
```

```
      p.x(1) = VIEW_HEIGHT - EPS;
    }
  }
}

void ComputeDensityPressure(void)
{
  for (auto &pi : particles)
  {
    pi.rho = 0.f;
    for (auto &pj : particles)
    {
      Vector2d rij = pj.x - pi.x;
      float r2 = rij.squaredNorm();

      if (r2 < HSQ)
      {
        pi.rho += MASS * POLY6 * pow(HSQ - r2, 3.f);
      }
    }
    pi.p = GAS_CONST * (pi.rho - REST_DENS);
  }
}

void ComputeForces(void)
{
  for (auto &pi : particles)
  {
    Vector2d fpress(0.f, 0.f);
    Vector2d fvisc(0.f, 0.f);
    for (auto &pj : particles)
    {
      if (&pi == &pj)
      {
        continue;
      }

      Vector2d rij = pj.x - pi.x;
      float r = rij.norm();

      if (r < H)
      {
        // Contribution des forces de pression
        fpress += -rij.normalized() * MASS * (pi.p + pj.p) / (2.f * pj.rho) * SPIKY_GRAD * pow(H - r, 3.f);
        // Contribution des forces de viscosite
        fvisc += VISC * MASS * (pj.v - pi.v) / pj.rho * VISC_LAP * (H - r);
      }
    }
    Vector2d fgrav = G * MASS / pi.rho;
    pi.f = fpress + fvisc + fgrav;
  }
}

void Update(void)
{
  ComputeDensityPressure();
  ComputeForces();
  Integrate();

```

```
  glutPostRedisplay();
}
```

3.3. Gestionnaire d'entrées utilisateur.

Comme mentionné précédemment, nous utilisons GLUT pour faciliter l'interaction entre le programme et l'utilisateur. Celui-ci peut effectuer deux actions simples en appuyant sur une touche : ajouter de petits blocs de fluide à déposer et réinitialiser la simulation. Ces actions sont assignées respectivement à la barre d'espace et à la touche r. On a également mis en place un compteur global, utilisé pour limiter le nombre de particules.

```
void Keyboard(unsigned char c, __attribute__((unused)) int x, __attribute__((unused)) int y)
{
  switch (c)
  {
  case ' ':
    if (particles.size() >= MAX_PARTICLES)
    {
      cout << "Nombre maximum de particules atteint" << endl;
    }
    else
    {
      unsigned int placed = 0;
      for (float y = VIEW_HEIGHT / 1.5f - VIEW_HEIGHT / 5.f; y < VIEW_HEIGHT / 1.5f + VIEW_HEIGHT / 5.f; y
          += H * 0.95f)
      {
        for (float x = VIEW_WIDTH / 2.f - VIEW_HEIGHT / 5.f; x <= VIEW_WIDTH / 2.f + VIEW_HEIGHT / 5.f; x
            += H * 0.95f)
        {
          if (placed++ < BLOCK_PARTICLES && particles.size() < MAX_PARTICLES)
          {
            particles.push_back(Particle(x, y));
          }
        }
      }
    }
    break;
  case 'r':
  case 'R':
    particles.clear();
    InitSPH();
    break;
  }
}
```

3.4. Résultat.

On obtient ainsi une simulation de fluides globalement satisfaisante, même si elle présente quelques limites : la tension superficielle n'a pas implémentée, aucune tentative n'a été faite pour paralléliser les calculs, l'équation des gaz parfaits entraîne une mauvaise application de l'incompressibilité et le schéma d'intégration numérique reste grossier et contribue à l'instabilité des paramètres. Le lecteur trouvera une simulation vidéo sur le site.

Code ImplémentationSPH.

```
#include <GLUT/glut.h>
#include <iostream>
#include <vector>
#include <eigen3/Eigen/Dense>
using namespace std;
using namespace Eigen;

// Constantes
const static Vector2d G(0.f, -10.f); // Forces gravitationnelles externes
const static float REST_DENS = 300.f; // Reste
const static float GAS_CONST = 2000.f; // Constante pour l'equation d'etat
const static float H = 15.f;  // Rayon
const static float HSQ = H * H;  // Rayon au carre pour optim
const static float MASS = 4.f;  // Toutes les particules ont meme masse
const static float VISC = 200.f; // Constante de viscosite
const static float DT = 0.001f; // Pas de temps integration

// Adaptation 2D de SPH par Muller
const static float POLY6 = 4.f / (M_PI * pow(H, 8.f));
const static float SPIKY_GRAD = -10.f / (M_PI * pow(H, 5.f));
const static float VISC_LAP = 40.f / (M_PI * pow(H, 5.f));

// Parametres simulation
const static float EPS = H; // Epsilon bord
const static float BOUND_DAMPING = -1.f;

//Structure des particules, inclut leur position, vitesse et force, et inclut la densité et les forces de
    pression pour SPH
struct Particle
{
  Particle(float _x, float _y) : x(_x, _y), v(0.f, 0.f), f(0.f, 0.f), rho(0), p(0.f) {}
  Vector2d x, v, f;
  float rho, p;
};

static vector<Particle> particles;

// Interactions
const static int MAX_PARTICLES = 3500; // Nombre de particules maximales
const static int DAM_PARTICLES = 0; // Nombre de particules au début
const static int BLOCK_PARTICLES = 234; // Nombre de particules par blocs

// Parametres de rendu glut
const static int WINDOW_WIDTH = 800;
const static int WINDOW_HEIGHT = 800;
const static double VIEW_WIDTH = 1.5 * 800.f;
const static double VIEW_HEIGHT = 1.5 * 600.f;

void InitSPH(void)
{
  for (float y = EPS; y < VIEW_HEIGHT - EPS * 2.f; y += H)
  {
    for (float x = VIEW_WIDTH / 4; x <= VIEW_WIDTH / 2; x += H)
    {
      if (particles.size() < DAM_PARTICLES)
      {
```

```
        float jitter = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        particles.push_back(Particle(x + jitter, y));
      }
      else
      {
        return;
      }
    }
  }
}

void Integrate(void)
{
  for (auto &p : particles)
  {
    // Integration par la methode d'Euler
    p.v += DT * p.f / p.rho;
    p.x += DT * p.v;

    // Conditions au bord
    if (p.x(0) - EPS < 0.f)
    {
      p.v(0) *= BOUND_DAMPING;
      p.x(0) = EPS;
    }
    if (p.x(0) + EPS > VIEW_WIDTH)
    {
      p.v(0) *= BOUND_DAMPING;
      p.x(0) = VIEW_WIDTH - EPS;
    }
    if (p.x(1) - EPS < 0.f)
    {
      p.v(1) *= BOUND_DAMPING;
      p.x(1) = EPS;
    }
    if (p.x(1) + EPS > VIEW_HEIGHT)
    {
      p.v(1) *= BOUND_DAMPING;
      p.x(1) = VIEW_HEIGHT - EPS;
    }
  }
}

void ComputeDensityPressure(void)
{
  for (auto &pi : particles)
  {
    pi.rho = 0.f;
    for (auto &pj : particles)
    {
      Vector2d rij = pj.x - pi.x;
      float r2 = rij.squaredNorm();

      if (r2 < HSQ)
      {
        pi.rho += MASS * POLY6 * pow(HSQ - r2, 3.f);
      }
    }
    pi.p = GAS_CONST * (pi.rho - REST_DENS);
```

```
  }
}

void ComputeForces(void)
{
  for (auto &pi : particles)
  {
    Vector2d fpress(0.f, 0.f);
    Vector2d fvisc(0.f, 0.f);
    for (auto &pj : particles)
    {
      if (&pi == &pj)
      {
        continue;
      }

      Vector2d rij = pj.x - pi.x;
      float r = rij.norm();

      if (r < H)
      {
        // Contribution des forces de pression
        fpress += -rij.normalized() * MASS * (pi.p + pj.p) / (2.f * pj.rho) * SPIKY_GRAD * pow(H - r, 3.f);
        // Contribution des forces de viscosite
        fvisc += VISC * MASS * (pj.v - pi.v) / pj.rho * VISC_LAP * (H - r);
      }
    }
    Vector2d fgrav = G * MASS / pi.rho;
    pi.f = fpress + fvisc + fgrav;
  }
}

void Update(void)
{
  ComputeDensityPressure();
  ComputeForces();
  Integrate();

  glutPostRedisplay();
}

void InitGL(void)
{
  glClearColor(1.f, 1.f, 1.f, 0); // Couleur du fond de la fenetre
  glEnable(GL_POINT_SMOOTH);
  glPointSize(H / 2.f);
  glMatrixMode(GL_PROJECTION);
}

void Render(void)
{
  glClear(GL_COLOR_BUFFER_BIT);

  glLoadIdentity();
  glOrtho(0, VIEW_WIDTH, 0, VIEW_HEIGHT, 0, 1);

  glColor4f(1.0f, 0.0f, 0.0f, 0.0f); // Couleur des particules
  glBegin(GL_POINTS);
  for (auto &p : particles)
```

```
  {
    glVertex2f(p.x(0), p.x(1));
  }
  glEnd();
  glutSwapBuffers();
}

void Keyboard(unsigned char c, __attribute__((unused)) int x, __attribute__((unused)) int y)
{
  switch (c)
  {
  case ' ':
    if (particles.size() >= MAX_PARTICLES)
    {
      cout << "Nombre maximum de particules atteint" << endl;
    }
    else
    {
      unsigned int placed = 0;
      for (float y = VIEW_HEIGHT / 1.5f - VIEW_HEIGHT / 5.f; y < VIEW_HEIGHT / 1.5f + VIEW_HEIGHT / 5.f; y
          += H * 0.95f)
      {
        for (float x = VIEW_WIDTH / 2.f - VIEW_HEIGHT / 5.f; x <= VIEW_WIDTH / 2.f + VIEW_HEIGHT / 5.f; x
            += H * 0.95f)
        {
          if (placed++ < BLOCK_PARTICLES && particles.size() < MAX_PARTICLES)
          {
            particles.push_back(Particle(x, y));
          }
        }
      }
    }
    break;
  case 'r':
  case 'R':
    particles.clear();
    InitSPH();
    break;
  }
}

int main(int argc, char **argv)
{
  glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInit(&argc, argv);
  glutCreateWindow("Implémentation SPH");
  glutDisplayFunc(Render);
  glutIdleFunc(Update);
  glutKeyboardFunc(Keyboard);

  InitGL();
  InitSPH();

  glutMainLoop();
  return 0;
}
```

Makefile.

```
INCLUDE_PATH    =  -I /opt/homebrew/Cellar/eigen/3.4.0_1/include/
LIBRARY_PATH    = -L/usr/local/lib/
OPENGL_LIBS     = -framework OpenGL -framework GLUT


TARGET = SimulationSPH
CC = g++
LD = g++
CFLAGS = -std=c++11 -O3 -Wall -Wno-deprecated -pedantic -Wno-vla-extension $(INCLUDE_PATH) -I./include
    -I./src -DNDEBUG
LFLAGS = -std=c++11 -O3 -Wall -Wno-deprecated -Werror -pedantic $(LIBRARY_PATH) -DNDEBUG
LIBS = $(OPENGL_LIBS)

OBJS = obj/main.o

default: $(TARGET)

all: clean $(TARGET)

$(TARGET): $(OBJS)
  $(LD) $(LFLAGS) $(OBJS) $(LIBS) -o $(TARGET)

obj/main.o: src/main.cpp
  mkdir -p obj
  $(CC) $(CFLAGS) -c src/main.cpp -o obj/main.o

clean:
  rm -f $(OBJS)
  rm -f $(TARGET)
  rm -f $(TARGET).exe
```

Références

[1] Matthias Müller, David Charypar and Markus Gross. Particle-Based Fluid Simulation for Interactive Applications. SIGGRAPH Symposium on Computer Animation, 2000.

[2] Lucas Schuermann. Work on SPH 2D implementation, 2017.