Fondamenti di Java
Smoothed Particle Hydrodynamics

Salvatore Mastrangelo

February 19, 2026

1 Introduzione

Smoothed Particle Hydrodynamics (SPH)èun metodo computazionale usato per simulare fluidi, spesso usato in simulazioni aerodinamiche. È basato sull' interazione tra macro-particelle e permette una simulazione in real time, ma richiede un grande numero di particelle per fornire risultati affidabili. È spesso usato in simulazioni la cui interattivitáèpreferita alla precisione fisica.

2 Equazioni Governanti

L'interazione tra particelleèdi tipo locale, e si estende fino ad una distanza prefissata chiamata *Smoothing Radius*. L'intensitá dell'interazione dipende da una funzione di smoothing, chiamata *Kernel Function*, che èuna funzione ad area unitaria e decrescente con la distanza. Spesso é definita come:

$$W(|\mathbf{r} - \mathbf{r}'|, h)$$

dove:

- $\mathbf{r}, \mathbf{r}'$: posizioni delle particelle
- h: smoothing radius

Tutte le grandeezze fisiche, come densitá, e forze di pressione, sono calcolate come:

$$A(\mathbf{r}) = \int A(\mathbf{r}')\, W\left(|\mathbf{r} - \mathbf{r}'|, h\right)\, dV(\mathbf{r}')$$

che puó essere discretizzata come:

$$A(\mathbf{r}) = \sum_j V_j A_j\, W\left(|\mathbf{r} - \mathbf{r}_j|, h\right)$$

dove:

- $A(\mathbf{r})$ rappresenta tale grandezza fisica in un punto $\mathbf{r}$.
- $V_j$èil volume associato alla particella j.
- $A_j$èil valore della grandezza fisica per la particella j.

Ad esempio, per il calcolo della densitá:

$$\rho(\mathbf{r}) = \sum_j V_j \rho_j \, W\left(|\mathbf{r} - \mathbf{r}_j|, h\right) = \sum_j m_j \, W\left(|\mathbf{r} - \mathbf{r}_j|, h\right)$$

3 Difficoltá Implementative

Tutte le ottimizzazioni elencate sono state implementate dopo una profilazione dei tempi di esecuzione di ciascuna funzione che contribuisce all'evoluzione del sistema, eseguita tramite l'ausilio della funzione `System.nanoTime()`, che ritorna il tempo in nanosecondi dall'avvio della macchina virtuale Java. Tutte le profilazioni sono state eseguite su un calcolatore con le seguenti specifiche:

- Java: OpenJDK 11.0.30
- CPU: Core 7 ultra 255hx
- RAM: 32GB

Per ottimizzare le prestazioni, in tutti i loop che iterano sulle particelle sono stati usati dei `.parallelStream().forEach(p -> {...})` per sfruttare l'alta parallelizzabilitá del problema, essendo tutte le operazioni tra loro indipendenti.

3.1 Ricerca dei Neighbors

Su ogni particella vengono applicate forze di pressione, viscositá e forze esterne, come la gravitá. Escludendo quest'ultima, tutte le forze necessitano di una ricerca dei Neighbors, dove una particella si definisce nel neighborhood di un'altra se dista meno di h, lo smoothing radius. Una ricerca esaustivaèstata usata in prima implementazione per semplicitá, maèrisultata eccessivamente lenta per numeri di particelle superiori a qualche migliaio. Per risolvere questo problema dopo l'update di ogni singola particella vengono calcolati degli indici in una griglia bidimensionale (`Particle.updateCell()`), in cui ogni cella ha lato di lunghezza h. Successivamente, viene riempito un vettore di arrayList con gli indici delle particelle che si trovano in ogni cella (`Fluid.updateCellMatrix()`), affinché in fase di ricerca sia sufficiente acquisire gli indici delle particelle nelle celle adiacenti a quella della particella in questione.

3.2 Calcolo delle Forze

Le forze di pressione e viscositá rappresentavano un collo di bottiglia significativo, poiché per ciascuna particella bisogna iterare sulle vicine. Per ridurre il numero di ricerche dei neighbors, le due funzioni di calcolo delle forze sono state inglobate in un'unica funzione (`Fluid.applyMergedForcesParallel()`), che ha portato ad un miglioramento delle prestazioni di circa il 10-15%.

3.3 Riduzione della Pressione sul GC

In prima implementazione, le funzioni di calcolo delle forze utilizzavano delle funzioni di supporto per operazioni tra vettori, implementate in `Vector2D`, che restituivano una nuova is-

tanza ad ogni call. 5 nuove istanze venivano create per ciascun vicino per ciascuna particella, portando ad un numero di oggetti creati per frame maggiore di 2.4 milioni per circa 4000 particelle. Tale pressione sul GC portava ad un rallentamento per frame di circa 2.5ms. Per risolvere questo problema, sono state implementate delle funzioni di supporto che modificano i vettori invece di ritornarne uno nuovo (`Vector2D.addThis(·)`, `Vector2D.scaleThis(·)`, `Vector2D.normalizeThis(·)`)

4 Possibili Sviluppi Futuri

Possibili Sviluppi sono articolabili sia riguardo l'accuratezza fisica sia riguardo la performance della simulazione.

4.1 Tensione Superficiale

La simulazione attuale non include fenomeni di tensione superficiale, modellizzabili come forze attrattive tra particelle vicine. L'attrazione in questione vedrebbe coinvolta una terza kernel function che impone una forza attrattiva tra particelle, che prevale a distanze maggiori rispetto alla forza di pressione, e cheètrascurabile a distanze ravvicinate, dove prevale l'ingombro della particella rappresentato come forza di pressione.

4.2 Parallelizzazione mediante GPU

L'attuale implementazione prevede solo una parallelizzazione a livello CPU per il calcolo delle grandezze fisiche, ma essendo la task fortemente parallelizzabile, sarebbe possibile sviluppare dei kernel per GPU. Ciononostante, questa idea richiederebbe una riscrittura della funzione di ricerca del neighborhood, che andrebbe ad utilizzare strutture dati piú adatte e tecniche di calcolo di hash per ciascuna cella.

4.3 Trasposizione in 3D

Pur essendo implementata in 2D, la trasposizione in 3 dimensioni sarebbe relativamente semplice, richiedendo solo una riscrittura delle kernel functions e delle funzioni di supporto di calcolo tra vettori. Un problema piú complesso potrebbe porsi nell'utilizzo della griglia per i neighbor, che in 3D risulterebbe molto piú dispendiosa in termini di memoria.