M2 Image, Développement et Technologies 3D (ID3D)
UE Animation, Corps Articulés et Moteurs physiques
Partie - Simulation par modèles physiques
Cours 5 - Position Based Dynamics (PBD)

Florence Zara

LIRIS - équipe Origami
Université Claude Bernard Lyon 1

http://liris.cnrs.fr/florence.zara
E-mail: florence.zara@liris.cnrs.fr

Plan du cours

- Rappel de l'approche classique de la dynamique Newtonienne
- Présentation de l'approche *Position Based Dynamics* pour simuler des objets
- Application à la simulation d'objets déformables : contraintes de type ressorts
- Application à la simulation de fluides : contraintes adaptées aux fluides

Dynamique Newtonienne - simulation de particules

Initialisation

Position initiale $x_i(t_0)$ des particules

Vitesse initiale $v_i(t_0)$ des particules

Masse m_i des particules

Pas de temps dt de la simulation

Boucle de simulation

1. **Calcul des forces $F_i(t)$ appliquées sur les particules**

   Forces extérieures : gravité, vent, interaction, etc.

   Forces internes au modèle physique : forces dues à des ressorts (objets déformables)

2. **Calcul des accélérations**

   $a_i(t) = F_i(t) / m_i$ (seconde loi de Newton)

3. **Intégration pour obtenir les nouvelles vitesses et positions (méthode d'Euler symplectique)**

   $v_i(t+dt) = v_i(t) + dt\ a_i(t)$

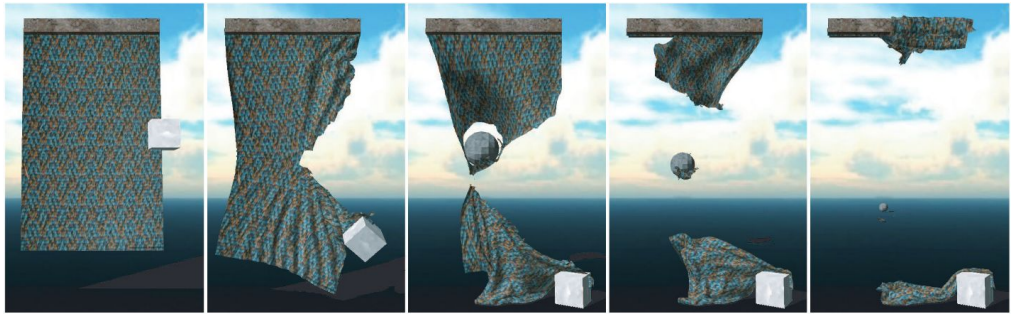
   $x_i(t+dt) = x_i(t) + dt\ v_i(t+dt)$

Approche basée sur le calcul de l'ensemble des forces exercées sur les particules pour calculer les accélérations qui seront intégrées pour obtenir nouvelles vitesses et positions des particules

Approche Position Based Dynamics (PBD)

Alternative à l'approche classique de la dynamique Newtonienne
(qui a été présentée dans les cours précédents)

Premier papier en 2006 en Computer Graphics



Matthias Müller, Bruno Heidelberger, Marcus Hennix, John Ratcliff, Position Based Dynamics, 2006

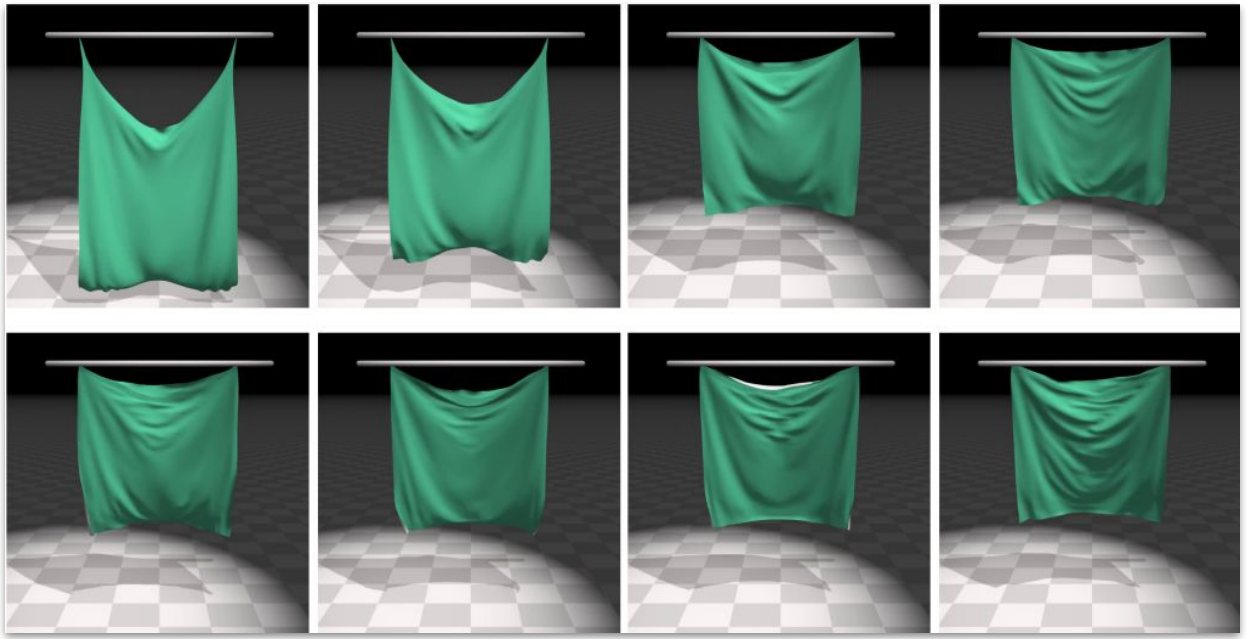
Et des extensions au fil des années

- Position Based Dynamics (2006)
- Hierarchical Position Based Dynamics (2008)
- XPBD: Position-Based Simulation of Compliant Constrained Dynamics (2016)
- A Constraint-based Formulation of Stable Neo-Hookean Materials (2021)

Simulation d'objets déformables, fluides, solides

Extended Position Based Dynamics

Miles Macklin, Matthias Muller, Nuttapong Chentanez (2016)

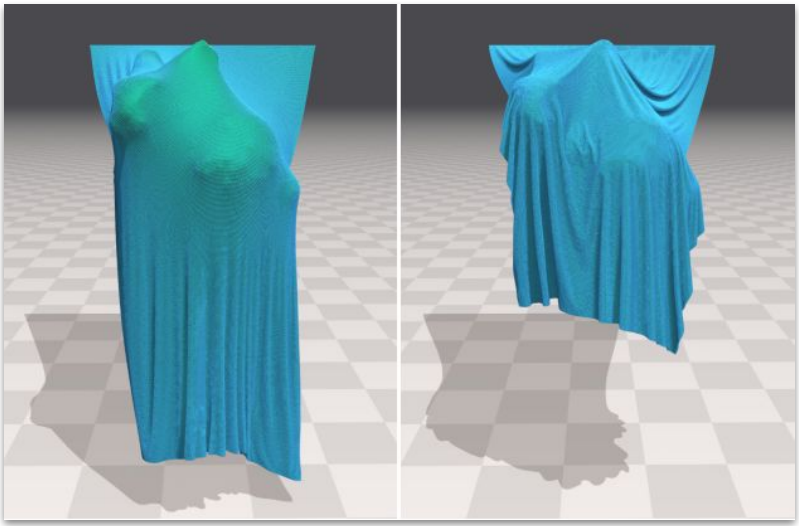


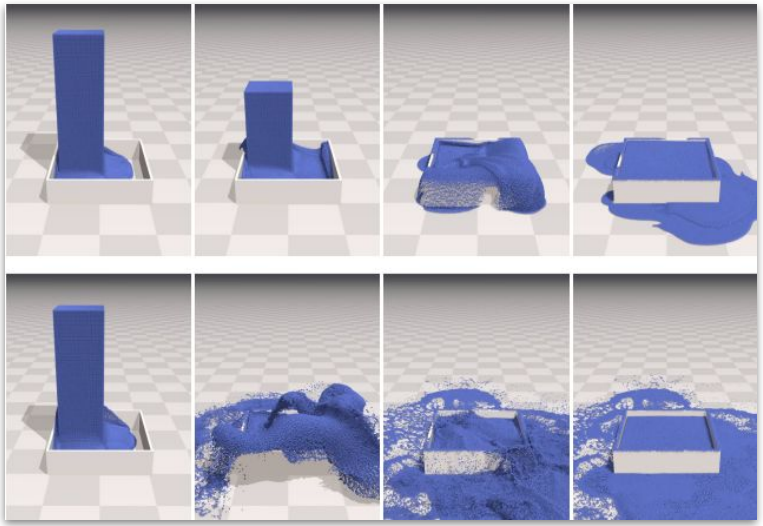
Simulation de textiles (surfaces déformables)

Simulation d'objets déformables, fluides, solides

Small Steps in Physics Simulation

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, Matthias Müller (2019)



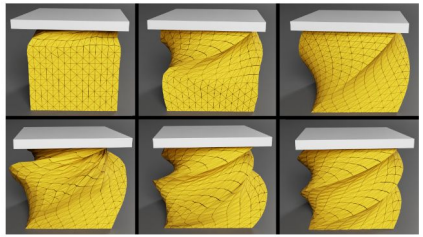


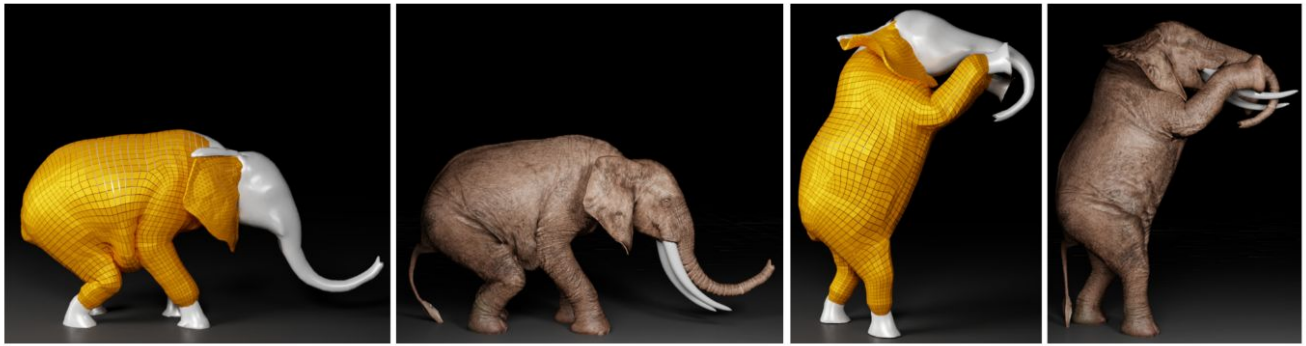
Simulation de textiles et fluides
Intérêt de pouvoir utiliser des petits pas de simulation

Simulation d'objets déformables, fluides, solides

A Constraint-based Formulation of Stable Neo-Hookean Materials

Miles Macklin, Matthias Müller (2021)



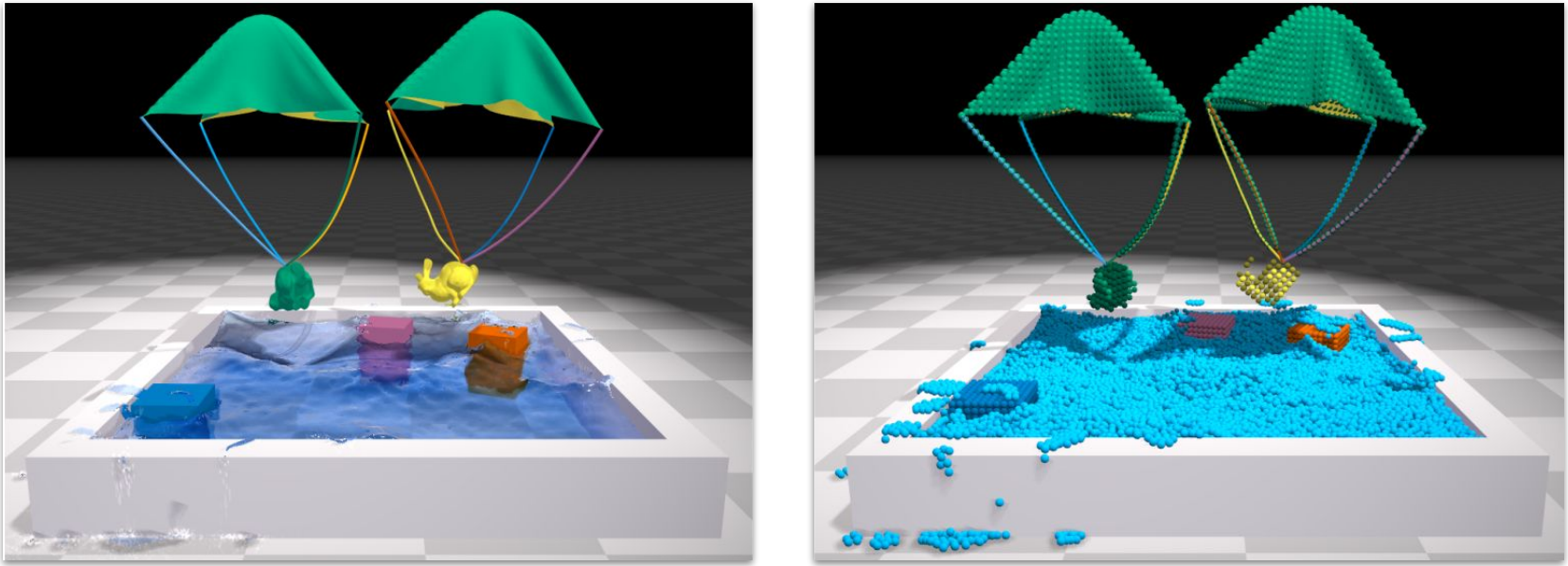


Simulation d'objets 3D déformables basée sur des lois de comportement (Néo-Hooke)

Simulation d'objets déformables, fluides, solides

Unified Particle Physics for Real-Time Applications

Miles Macklin *Matthias Müller* *Nuttapong Chentanez* *Tae-Yong Kim* (2014)



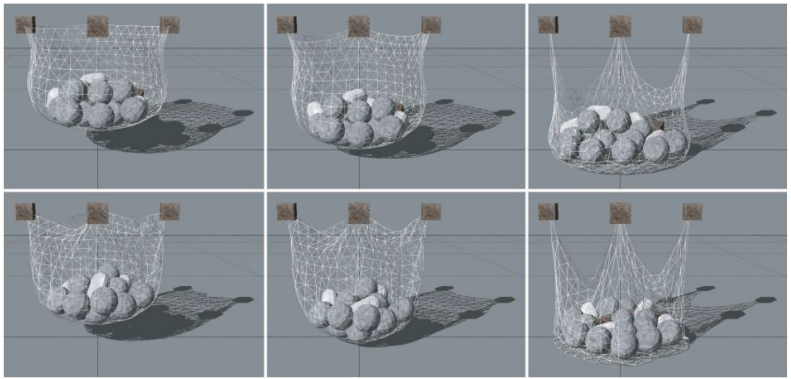
Simulation de fluides basée particules

Algorithme complet - *Position Based Dynamics* (PBD)

3rd Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2006)
C. Mendoza, I. Navazo (Editors)

Position Based Dynamics

Matthias Müller Bruno Heidelberger Marcus Hennix John Ratcliff



Based on this data and a time step Δt, the dynamic object is simulated as follows:

(1) **forall** vertices i
(2) initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
(3) **endfor**
(4) **loop**
(5) **forall** vertices i **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
(6) dampVelocities$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(7) **forall** vertices i **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
(8) **forall** vertices i **do** generateCollisionConstraints$(\mathbf{x}_i \rightarrow \mathbf{p}_i)$
(9) **loop** solverIterations **times**
(10) projectConstraints$(C_1, \ldots, C_{M+M_{coll}}, \mathbf{p}_1, \ldots, \mathbf{p}_N)$
(11) **endloop**
(12) **forall** vertices i
(13) $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
(14) $\mathbf{x}_i \leftarrow \mathbf{p}_i$
(15) **endfor**
(16) velocityUpdate$(\mathbf{v}_1, \ldots, \mathbf{v}_N)$
(17) **endloop**

Approche PBD - Méthode basée contraintes

Objet représenté par N particules/sommets i et M contraintes

Contraintes vont permettre de définir les propriétés physiques de l'objet (objet déformable, fluide, rigide, etc.)

- Contraintes = restriction cinétiques = fonction des positions et des dérivées
- Dans le cadre de l'approche PBD : uniquement fonction des positions
- Contraintes unilatérales : $C(x_{i1}, \ldots, x_{inj}) = 0$ *pour la particule i*
- Contraintes bilatérales : $C(x_{i1}, \ldots, x_{inj}) \geqslant 0$ *pour la particule i*
- nj = nombre de contraintes associées à la particule i

Pour chaque contrainte :

il y a un paramètre de raideur associé $k \in [0, 1]$ (modulation de la contrainte)

Approche PBD : consiste à modifier les positions afin de tenir compte de ces contraintes

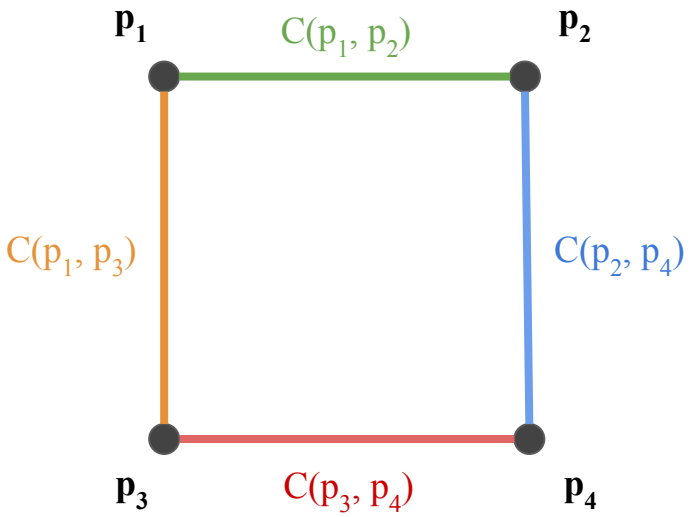
Approche PBD - Méthode basée contraintes

Particule/sommet i $\in$ [1, . . . , N]

position : x_i
vitesse : v_i
masse : m_i

Contraintes j $\in$ [1, . . . , M]

cardinalité : n_j
indices : $\{i_1, \ldots i_{n_j}\}, i_k \in [1, \ldots N]$
fonction : $C_j(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_{n_j}}) \rightarrow \mathbb{R}$
raideur : k $\in$ [0,1]
condition : *égalité (C = 0)* ou *inégalité (C ≥ 0)*



Algorithme - Position Based Dynamics (PBD) - Initialisation

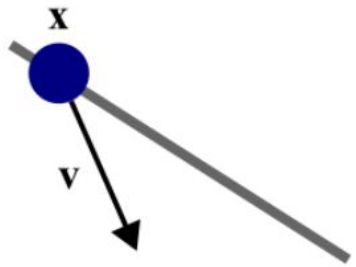
Première étape de l'algorithme

Initialisation des positions x_i et vitesses v_i des particules de l'objet

Définition des masses m_i des particules pour définir $w_i = 1/m_i$

Rq : Étape nécessaire car schéma du 2nd ordre en temps (sinon trop de variables inconnues)

```
forall vertices i do
        initialize x_i = x_i^0
        initialize v_i = v_i^0
        initialize w_i = 1 / mi
endfor
```



Les accélérations ne sont pas calculées (et donc pas mises à jour dans l'algorithme)
L'algorithme va travailler directement sur les positions x_i des particules

Algorithme - Position Based Dynamics (PBD) - Simulation

Seconde étape de l'algorithme

On va entrer dans la boucle de simulation

Le temps continu est discrétisé avec un pas de temps Δt

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Schéma d'intégration d'Euler symplectique appliqué aux vitesses

normalement :

$v(t+dt) = v(t) + dt\, a(t)$

$= v(t) + dt\, f(t)/m$

ici : $v(t+dt) = v(t) + dt\, fext(t)/m$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

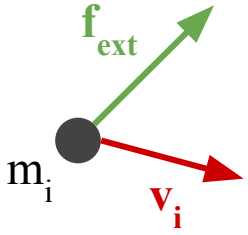
$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop



$w_i = 1 / m_i$ $f_{ext}\, w_i$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

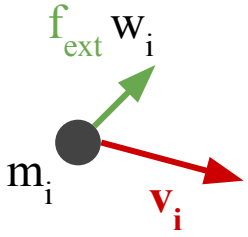
$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop



$w_i = 1 / m_i$ $f_{ext}\ w_i$

$v_i = v_i + \Delta t\, f_{ext}\, w_i$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

$w_i = 1 / m_i$ $f_{ext}\, w_i$

$v_i = v_i + \Delta t\, f_{ext}\, w_i$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Amortissement des vitesses

$v_i = v_i + ???$

$v_i = v_i - k_{damp} v_i$ $\quad k_{damp} \in [0,1]$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

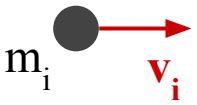
$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Amortissement des vitesses



$v_i = v_i + ???$

$v_i = v_i - k_{damp} v_i$ $\quad k_{damp} \in [0,1]$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Schéma d'intégration d'Euler symplectique appliqué aux positions

Obtention de **prédictions p_i** qui ne sont pas directement assignées aux positions

Algorithme manipule x_i et p_i pour les positions

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

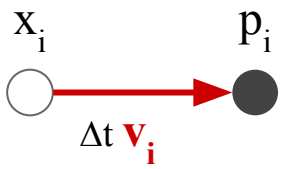
$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop



Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

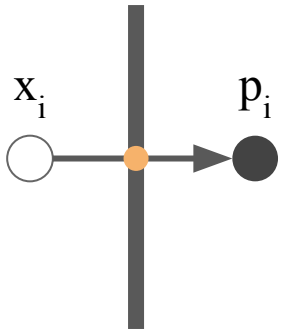
endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Génération des contraintes externes non permanentes (collisions)

???



$C \leftarrow C \cup C_{col}(p_i)$

Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

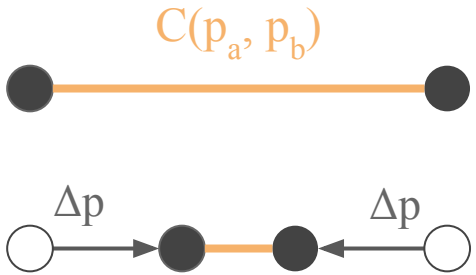
endfor

velocityUpdate($v_1, \ldots, v_N$)

endloop

Correction des prédictions p_i en fonction des contraintes

Utilisation d'un solveur itératif
(détaillé par la suite)



Boucle de simulation de la méthode PBD

While(True)

forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

dampVelocities($v_1, \ldots, v_N$)

forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

loop solverIterations times

projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)

endloop

forall vertices i

$v_i \leftarrow (p_i - x_i)/\Delta t$

$x_i \leftarrow p_i$

endfor

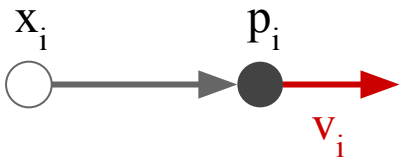
velocityUpdate($v_1, \ldots, v_N$) ← frictions, collisions, etc.

endloop

Mise à jour finale avec l'intégration de Verlet

$$\mathbf{v_i} = \frac{\mathbf{p_i} - \mathbf{x_i}}{\Delta t}$$

$$\mathbf{x_i} = \mathbf{p_i}$$



Algorithme - Position Based Dynamics (PBD)

Initialisation

forall vertices i do
    initialize $x_i = x_i^0$
    initialize $v_i = v_i^0$
    initialize $w_i = 1 / m_i$
endfor

While(True)
    forall vertices i do $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

    dampVelocities($v_1, \ldots, v_N$)

    forall vertices i do $p_i \leftarrow x_i + \Delta t\, v_i$

    forall vertices i do generateCollisionConstraints($x_i \rightarrow p_i$)

    loop solverIterations times
        projectConstraints($C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N$)
    endloop

    forall vertices i
        $v_i \leftarrow (p_i - x_i)/\Delta t$
        $x_i \leftarrow p_i$
    endfor
    velocityUpdate($v_1, \ldots, v_N$)

endloop

Boucle de simulation

Résumé de l'algorithme PBD

1. **Initialisation des positions et vitesses** (nécessaire car schéma du 2nd ordre en temps)

2. **Schéma d'intégration d'Euler symplectique appliqué aux vitesses et positions**
   Calcul des forces/interactions **extérieures** *pas de calculs des accélérations*
   Calcul des vitesses v_i en fonction de ces forces extérieures *méthode d'Euler*
   Amortissement des vitesses
   Calcul des prédiction p_i *méthode d'Euler - pas directement assignées aux positions*
   algorithme manipule x_i et p_i pour les positions

3. Considère les **contraintes extérieures non-permanentes** (comme les collisions)
   Définition des contraintes de collisions

4. Utilisation **solveur itératif pour corriger les positions prédites** afin de satisfaire les contraintes
   M contraintes internes à l'objet et M_{coll} contraintes de collisions $p_i = p_i + \Delta p_i$

5. Utilisation des prédictions pi pour **mettre à jour les vitesses v_i et les positions x_i**
   $v_i = (p_i - v_i) / dt$ approximation de la dérivée par rapport à ces 2 "positions"
   $x_i = p_i$

Ce qui reste à comprendre de cette approche PBD

Quelles sont les contraintes et comment elles se définissent ?

Comment sont résolues ces contraintes ?

Résolution des contraintes - Utilisation d'un solveur itératif

Objectif : Corriger les positions prédites p_i afin de satisfaire des contraintes

Résolution d'un système ayant M équations avec 3N inconnues :

M contraintes connues, N le nombre de particules qui ont une position $x \in \mathbb{R}^3$

Pour la particule i, on a la contrainte *non-linéaire* de la forme : $C(x_{i1}, \ldots, x_{inj}) = 0$ ou $C(x_{i1}, \ldots, x_{inj}) \geqslant 0$

La **méthode de Gauss-Seidel** est choisie pour résoudre le système, par contre elle résout uniquement des systèmes d'équations *linéaires* de la forme $A\,x = b$

=> Linéarisation à faire des fonctions de contraintes dans le voisinage de la solution

= Passage de la formulation de la contrainte de la forme $c(x) = 0$ à la forme $A\,x = 0$

Résolution des contraintes - Linéarisation des contraintes

Dans le cas d'une contrainte d'égalité $c(x) = 0$, la contrainte est linéarisée en approximant comme suit :

Rappel développement limité d'ordre 1 au voisinage de t_1: $f(t_1) \approx f(t_0) + f'(t_0)(t_1 - t_0)$ *sur intervalle* $[t_0, t_1]$

Pour la contrainte, on obtient : $C(x) \approx C(p) + \nabla C(p)(x-p) = 0$ *Position x, position prédite p*

$C(p + \Delta p) \approx C(p) + \nabla C(p) \cdot \Delta p = 0$ *En fonction des prédictions p*

$C(\mathbf{x}+\Delta\mathbf{x}) \approx C(\mathbf{x}) + \nabla C(\mathbf{x}) \cdot \Delta\mathbf{x} = 0$ *En fonction des positions x*

Cela nous donne une formulation linéaire de la contrainte : $C(\mathbf{x}) = -\nabla C(\mathbf{x}) \cdot \Delta\mathbf{x}$

Le système de contraintes est alors ré-écrit pour obtenir le système linéaire suivant :

$$\begin{aligned} \nabla C_1(\mathbf{x}) \cdot \Delta\mathbf{x} &= -C_1(\mathbf{x}) \\ &\dots \\ \nabla C_M(\mathbf{x}) \cdot \Delta\mathbf{x} &= -C_M(\mathbf{x}), \end{aligned}$$

Résolution du système pour obtenir la solution Δx qui correspond à la correction de la position **x** tenant compte de toutes ses contraintes

Résolution des contraintes - Restriction de la solution

Si on n'a pas une contrainte d'égalité, la solution $\Delta\mathbf{x}$ doit être restreinte dans la direction de ∇C :

$$\Delta\mathbf{x} = \lambda\, \mathbf{M}^{-1}\, \nabla C(\mathbf{x})^T$$

M = matrice des masses des particules, elles sont inversées pour pondérer les corrections proportionnelles aux masses inverses

Il suffit alors simplement de déterminer les multiplicateurs de Lagrange λ (facteur scalaire) pour obtenir la solution $\Delta\mathbf{x}$ correspondant à la correction

C(x) a été ré-écrit comme : $-C(x) = \nabla C(x) \cdot \Delta x$ avec $\Delta x = \lambda\, M^{-1} \nabla C(x)^T$

Par substitution on trouve :

$$\lambda = \frac{C(\mathbf{x})}{\nabla C(\mathbf{x}) \mathbf{M}^{-1} \nabla C(\mathbf{x})^T}$$

$$\lambda = \frac{C(\mathbf{x})}{|\nabla C(\mathbf{x})|^2 \mathbf{M}^{-1}}$$

Résolution des contraintes - En résumé

Objectif : Position x doit être modifiée pour prendre en compte des contraintes

Méthode :

- Calcul du multiplicateur de Lagrange λ
  - Besoin de calculer la contrainte et gradient de la contrainte

$$\lambda = \frac{C(\mathbf{x})}{\nabla C(\mathbf{x}) \mathbf{M}^{-1} \nabla C(\mathbf{x})^T}$$

$$\lambda = \frac{C(\mathbf{x})}{|\nabla C(\mathbf{x})|^2 \mathbf{M}^{-1}}$$

- Calcul de la correction qui permet de tenir compte de la contrainte :
  - $\Delta x = \lambda M^{-1} \nabla C(x)^T$

- Mise à jour de la position x pour prendre en compte la correction :
  - $x = x + \Delta x$

Retour dans le cadre de l'algorithme PBD

On a obtenu les prédictions p_i des positions

forall vertices i do $\mathbf{p_i \leftarrow x_i + \Delta t\, v_i}$

On définit ensuite les contraintes à respecter

forall vertices i do generateCollisionConstraints$(\mathbf{x_i \rightarrow p_i})$

On souhaite ensuite qu'elles satisfassent les contraintes énoncées

loop solverIterations times
projectConstraints$(\mathbf{C_1, \ldots, C_{M+Mcoll}, p_1, \ldots, p_N})$
endloop

- Calcul du multiplicateur de Lagrange λ
- Calcul de la correction Δp_i
- Mise à jour de la prédiction : $p_i = p_i + \Delta p_i$

Retour dans le cadre de l'algorithme PBD

Pour la particule i, on a ainsi :

Si masses uniformes

$$\Delta p_i = -\lambda \nabla_{p_i} C(p_1, \ldots, p_{n_j})^T$$
$$= -\lambda \nabla_{p_i} C(p)^T$$

Si masses non-uniformes

$$\Delta p_i = -\lambda w_i \nabla_{p_i} C(p_1, \ldots, p_{n_j})^T$$
$$= -\lambda w_i \nabla_{p_i} C(p)^T \qquad \text{avec } w_i = 1/m_i$$

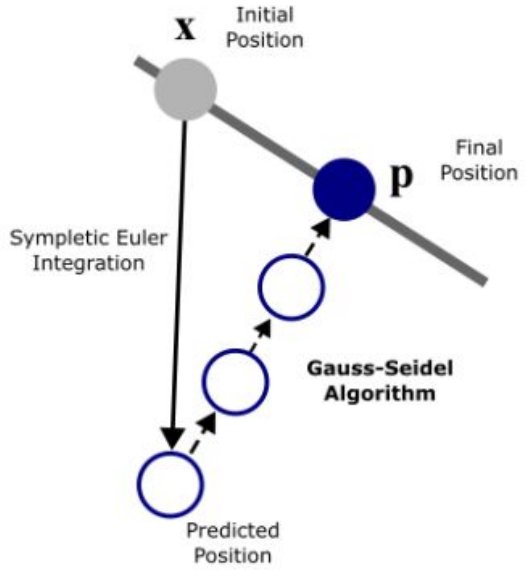
$$\lambda = \frac{C(p)}{\sum_j w_j \| \nabla_{p_j} C(p) \|^2}$$

Pour calculer le multiplicateur de Lagrange,
on a besoin de calculer la contrainte et le gradient de la contrainte

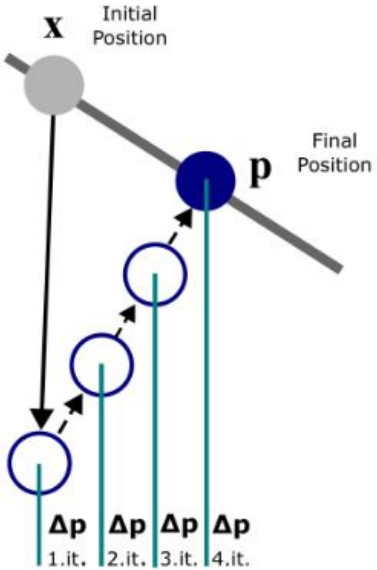
A la fin des itérations, prédiction p_i corrigée avec $p_i = p_i + \Delta p_i$
pour satisfaire toutes les contraintes

Résolution des contraintes - Principe du solveur itératif

Méthode de Gauss-Seidel est itérative : à chaque itération, les nouvelles valeurs calculées remplacent les anciennes valeurs



Algorithme de Gauss-Seidel en action :
La position prédite p est déplacée itérativement pour satisfaire la contrainte (la ligne grise)



Mise à jour de p de manière itérative pour satisfaire nos contraintes :
La position prédite p est déplacée itérativement de $\Delta \boldsymbol{p}$ pour satisfaire la contrainte (la ligne grise)

Ce qui reste à comprendre de cette approche PBD

Quelles sont les contraintes et comment elles se définissent ?

~~Comment sont résolues ces contraintes ?~~

Contrainte de distance entre deux particules

Contrainte de distance définie par : $C(p_1, p_2) = \| p_1 - p_2 \| - d$

Contrainte d'égalité : $C(p_1, p_2) = 0 \Rightarrow \| p_1 - p_2 \| = d$

Contrainte d'inégalité : $C(p_1, p_2) \geq 0 \Rightarrow \| p_1 - p_2 \| \geq d$

On peut l'écrire aussi de cette façon :

$$C(p_1, p_2) = [(p_1 - p_2)^T . (p_1 - p_2)]^{0.5} - d$$

Contrainte de distance entre deux particules

Correction à chaque itération : $\Delta p_i = -\lambda\, w_i\, \nabla_{p_i} C(p)^T$ $\qquad \lambda = \dfrac{C(p)}{\sum_j w_j \|\nabla_{p_j} C(p)\|^2}$

$$\nabla_{p1} C(p1, p2) = \frac{1}{2(\|p1 - p2\|^2)^{0.5}}\, 2\,(p1 - p2)$$

$$\nabla_{p1} C(p1, p2) = n \quad \text{et} \quad \nabla_{p2} C(p1, p2) = -n \text{ où } n = \frac{p1 - p2}{\|p1 - p2\|}$$

$$\Delta p1 = -\frac{w_1(\|\, p1 - p2 \,\| - d)}{w1 + w2}\, \frac{p1 - p2}{\|p1 - p2\|}$$

$$\Delta p2 = +\frac{w_2(\|\, p1 - p2 \,\| - d)}{w1 + w2}\, \frac{p1 - p2}{\|p1 - p2\|}$$

$(a^T.a)' = 2a$

$(f \circ g(x))' = f' \circ g(x)\, g'(x)$

$(x^n)' = n\, x^{n-1}$

Contrainte de distance entre deux particules

$$C(p_1, p_2) = \| p_1 - p_2 \| - d$$

Contrainte d'égalité : $C(p_1, p_2) = 0 \Rightarrow \| p_1 - p_2 \| = d$

Contrainte d'inégalité : $C(p_1, p_2) \geq 0 \Rightarrow \| p_1 - p_2 \| \geq d$

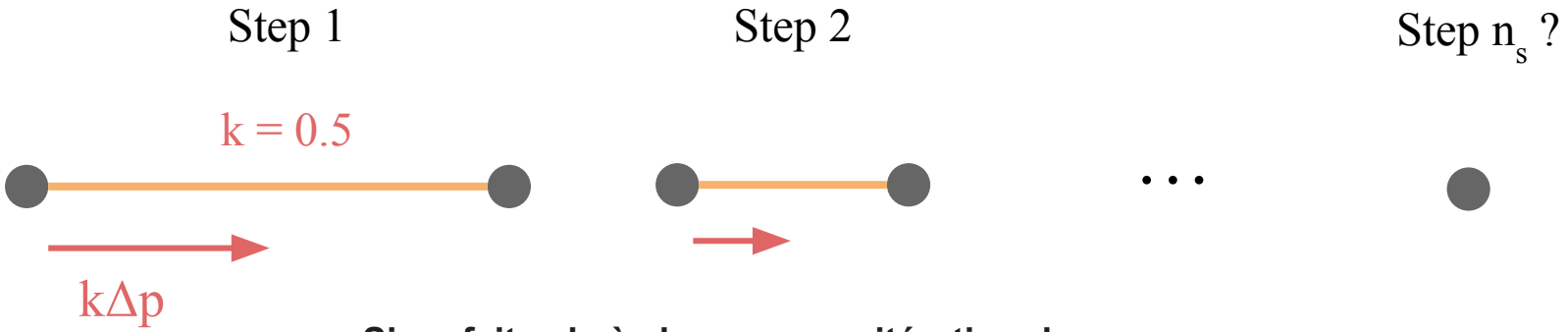
$$\Delta p_1 = - \frac{w_1(\| p_1 - p_2 \| - d)}{w_1 + w_2} \frac{p_1 - p_2}{\|p_1 - p_2\|}$$

$$\Delta p_2 = + \frac{w_2(\| p_1 - p_2 \| - d)}{w_1 + w_2} \frac{p_1 - p_2}{\|p_1 - p_2\|}$$

Correction à apporter sur p_1 et p_2 pour tenir compte de la contrainte de distance $C(p_1,p_2)$

Contrainte de distance entre deux particules - Facteur k

Application de la modération de la contrainte avec raideur k ∈ [0, 1] : k Δp



Si on fait cela à chaque sous-itération du solveur itératif, l'effet de k est non-linéaire

Emploi à la place : $k' = 1 - (1 - k)^{1/ns}$ avec n_s le nombre de sous-itérations du solveur

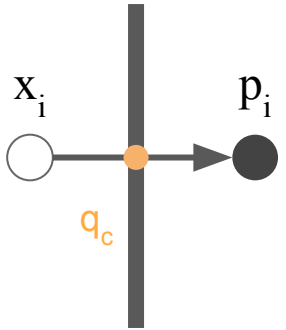
Contrainte de collision entre deux objets rigides

Pour chaque particule i :

- Teste le rayon $x_i \rightarrow p_i$
- Si le rayon rencontre un objet
  - Calcul du point d'impact q_c
  - Calcul de la normale à la surface n_c en ce point
  - Définition de la contrainte de collision :

$$C(p) = (p - q_c) \cdot n_c$$

Détection de collision



Amortissement pour les objets rigides

(1) $\mathbf{x}_{cm} = (\sum_i \mathbf{x}_i m_i)/(\sum_i m_i)$
(2) $\mathbf{v}_{cm} = (\sum_i \mathbf{v}_i m_i)/(\sum_i m_i)$
(3) $\mathbf{L} = \sum_i \mathbf{r}_i \times (m_i \mathbf{v}_i)$
(4) $\mathbf{I} = \sum_i \tilde{\mathbf{r}}_i \tilde{\mathbf{r}}_i^T m_i$
(5) $\omega = \mathbf{I}^{-1}\mathbf{L}$
(6) **forall** vertices i
(7) $\quad \Delta\mathbf{v}_i = \mathbf{v}_{cm} + \omega \times \mathbf{r}_i - \mathbf{v}_i$
(8) $\quad \mathbf{v}_i \leftarrow \mathbf{v}_i + k_{\text{damping}} \Delta\mathbf{v}_i$
(9) **endfor**

$\mathbf{ri} = \mathbf{x_i} - \mathbf{x_{cm}}$

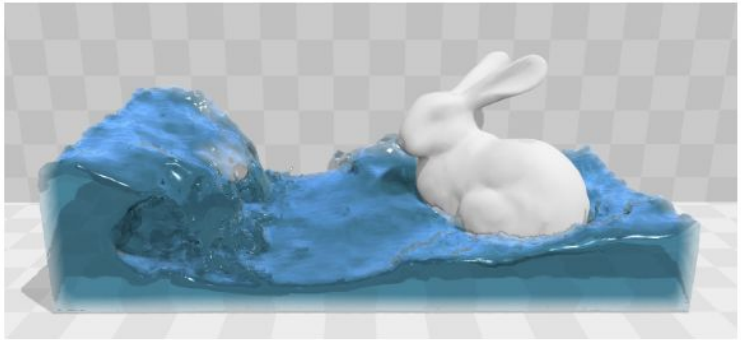
$k_{damping} \in [0, 1]$

$\tilde{\mathbf{r}}_i \mathbf{v} = \mathbf{r}_i \times \mathbf{v}$

Méthode PBD pour simuler les fluides

Position Based Fluid

Miles Macklin, Matthias Müller, 2013



Algorithm 1 Simulation Loop

for all particles i **do**
  apply forces $\mathbf{v}_i \Leftarrow \mathbf{v}_i + \Delta t \mathbf{f}_{ext}(\mathbf{x}_i)$
  predict position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$
end for
for all particles i **do**
  find neighboring particles $N_i(\mathbf{x}_i^*)$
end for
while $iter < solverIterations$ **do**
  for all particles i **do**
    calculate λ_i
  end for
  for all particles i **do**
    calculate $\Delta \mathbf{p}_i$
    perform collision detection and response
  end for
  for all particles i **do**
    update position $\mathbf{x}_i^* \Leftarrow \mathbf{x}_i^* + \Delta \mathbf{p}_i$
  end for
end while
for all particles i **do**
  update velocity $\mathbf{v}_i \Leftarrow \frac{1}{\Delta t}(\mathbf{x}_i^* - \mathbf{x}_i)$
  apply vorticity confinement and XSPH viscosity
  update position $\mathbf{x}_i \Leftarrow \mathbf{x}_i^*$
end for

Approche PBD pour simuler un fluide

Particule/sommet $i \in [1, \ldots, N]$

position : x_i

vitesse : v_i

masse : m_i

densité : ρ_0

Contraintes $j \in [1, \ldots, M]$

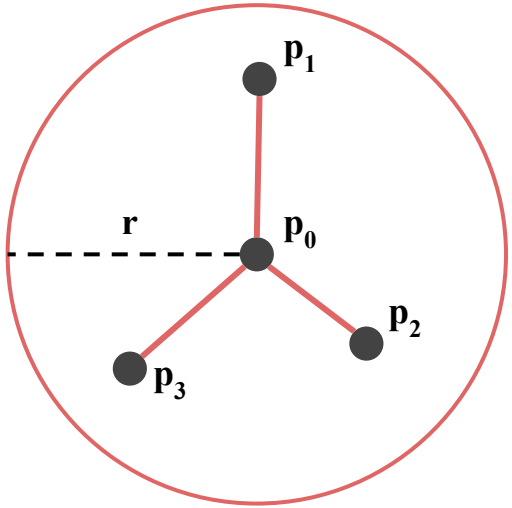
cardinalité : n_i

indices : $\{i_1, \ldots i_{n_j}\}, i_k \in [1, \ldots N]$

fonction : $C_j(\mathbf{x}_{i_1}, \ldots, \mathbf{x}_{i_{n_j}}) \rightarrow \mathbb{R}$

raideur : $k \in [0,1]$

condition : *égalité* $(C = 0)$ ou *inégalité* $(C \geq 0)$



$C(p_0, \{p_1, p_2, p_3\})$

Boucle de simulation PBD pour simuler un fluide

while(True)

forall particles i **do** $v_i \leftarrow v_i + \Delta t\, w_i\, f_{ext}$

forall particles i **do** $p_i \leftarrow x_i + \Delta t\, v_i$

forall particles i **do** **findNeighbours**(p_i)

loop solverIterations **times**

forall particles i **do**

solveConstraints()

checkCollisions() & solveCollisions()

forall particles i **do**

$p_i \leftarrow p_i + \Delta p_i$

forall particles i

$v_i \leftarrow (p_i - x_i) / \Delta t$

applyVorcity() & applyViscosity()

$x_i \leftarrow p_i$

endfor

Schéma d'intégration d'Euler symplectique appliqué aux vitesses et positions

Particle Simulation using CUDA (2010)

Contraintes de densité

Contraintes de collision

Applique les corrections

Mise à jour des vitesses avec l'intégration de Verlet

Viscosité du fluide et tourbillon

Mise à jour des positions

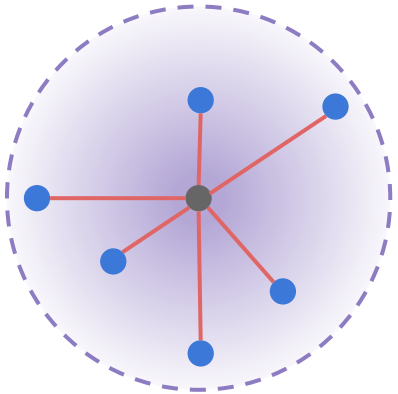
Contraintes pour les fluides et sa correction associée

$$C(x) = \frac{\rho i}{\rho 0} - 1 \leq 0$$

$$\rho i = \sum_j m_j W(p_i - p_j, h)$$

ρ_0 : densité au repos

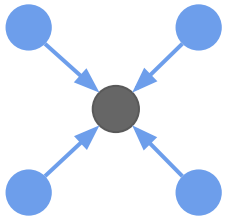
ρi : utilise méthode SPH pour l'estimer



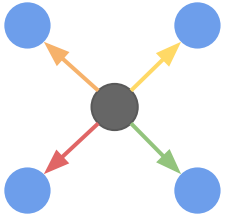
$W \in [0,1]$

Gradient de la contrainte :

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = i \\ -\nabla_{\mathbf{p}_k} W(\mathbf{p}_i - \mathbf{p}_j, h) & \text{if } k = j \end{cases}$$



k = i



k = j

Multiplicateur de Lagrange : $\lambda_i = -\frac{C_i(\mathbf{p}_1, \ldots, \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2}$

Correction : $\Delta \mathbf{p}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$

Quelques améliorations

Instabilité en tension de surface

$$s_{corr} = -k\left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta\mathbf{q}, h)}\right)^n$$

$$\Delta\mathbf{p}_i = \frac{1}{\rho_0}\sum_j \left(\lambda_i + \lambda_j + s_{corr}\right)\nabla W(\mathbf{p}_i - \mathbf{p}_j, h)$$

Ajout d'une pression artificielle provoquant une tension superficielle

Traitement des particules solitaires pour éviter qu'elles n'explosent

Confinement du tourbillon et viscosité

Tourbillon

$$\omega_i = \nabla \times \mathbf{v} = \sum_j \mathbf{v}_{ij} \times \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_j, h)$$

$$\mathbf{f}_i^{vorticity} = \varepsilon\left(\mathbf{N} \times \omega_i\right)$$

Viscosité

$$\mathbf{v}_i^{new} = \mathbf{v}_i + c\sum_j \mathbf{v}_{ij} \cdot W(\mathbf{p}_i - \mathbf{p}_j, h)$$

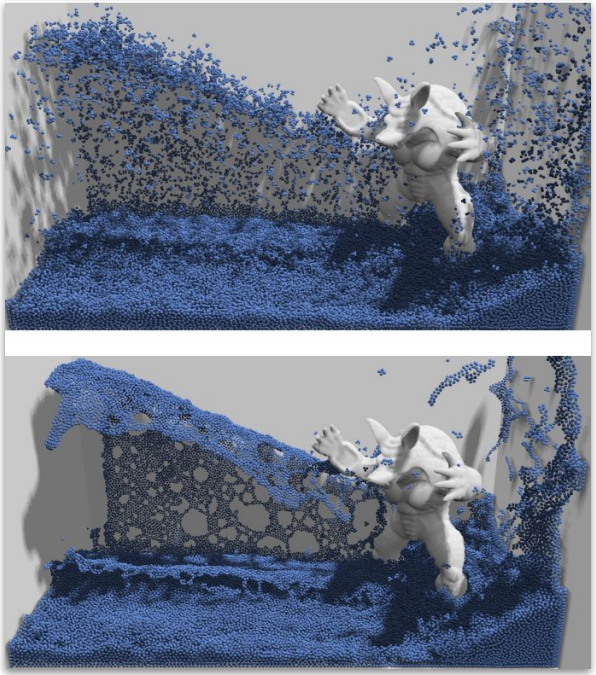
Ajout du mouvement du tourbillon

Augmente la cohérence du mouvement

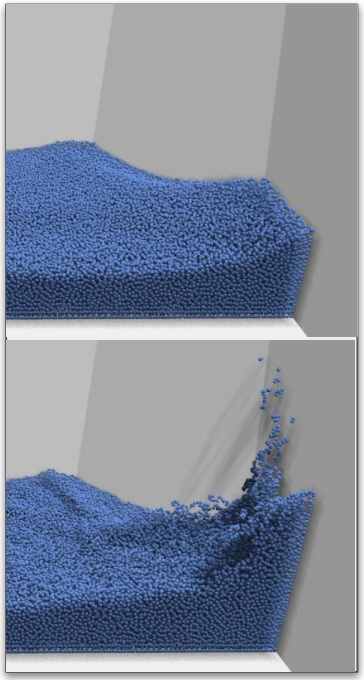
Plus… pour que cela soit joli

Approche PBD pour simuler un fluide

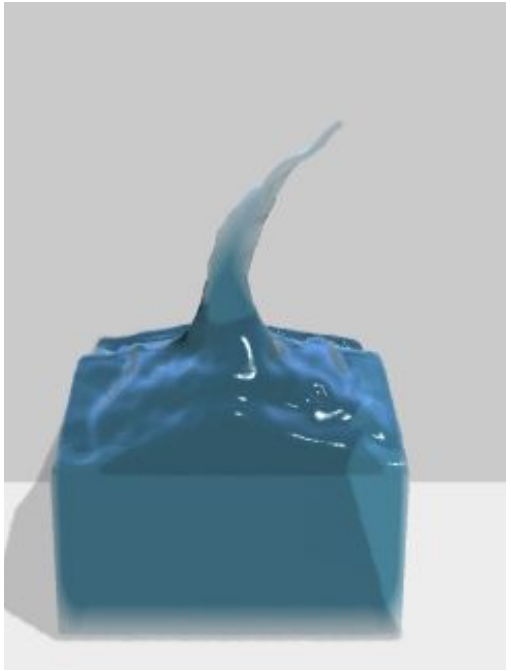
Instabilité



Tourbillon



Rendu



En conclusion

- Nouvelle approche pour faire de la simulation d'objets 2D ou 3D
- Méthode adaptée aux objets déformables, aux objets rigides et aux fluides
- Méthode basée sur des contraintes
- Utilisation d'un solveur itératif pour résoudre les contraintes
- Dans la suite des papiers, de plus en plus proches de la physique

Avantages de l'approche :

- Plus de contrôle en modifiant directement les positions
- Plus de stabilité (à voir)
- Plus simple pour gérer les collisions
- Contraintes peuvent être résolues facilement en parallèle (sur GPU)

Références bibliographiques

Articles de recherche :

- M. Muller, B. Heidelberger, M. Hennix, J. Ratcliff, Position Based Dynamics, VRIPhys 2006 (vidéo du papier de M. Muller)
- M. Macklin, M. Müller, Position Based Fluids, SIGGRAPH 2013
- et tous les suivants : https://matthias-research.github.io/pages/publications/publications.html
- Cours SIGGRAPH 2017 - A survey on Position Based Dynamics

Une page web qui explique plein de choses (méthode PBD, mais beaucoup d'autres choses) :
https://carmencincotti.com/fr/2022-08-01/la-boucle-de-simulation-de-pbd

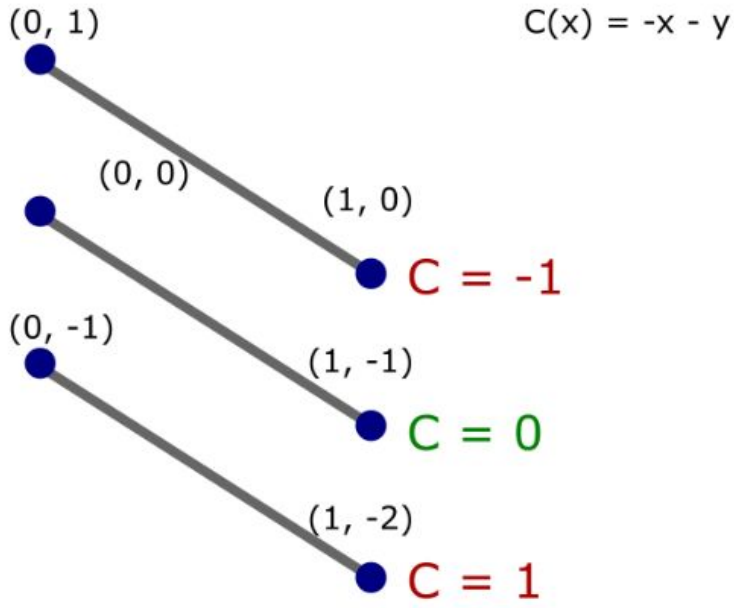
Cours effectués en partant initialement de présentations faites par Bastien Saillant (équipe Origami, LIRIS)

Des choses en plus…

Correction des positions - Méthode de Newton-Raphson

Soit la fonction contrainte $C(x) = -x - y \Rightarrow y = -x - C(x) \Rightarrow$ droite de pente -1

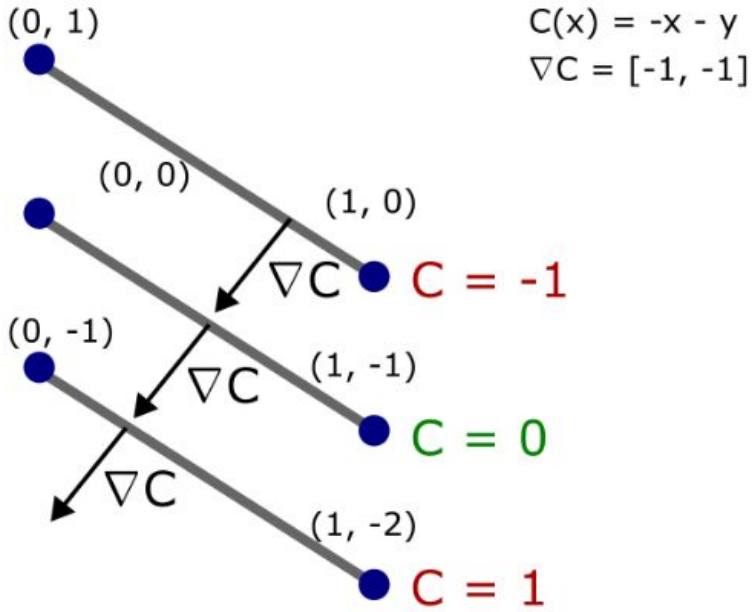
Nous souhaitons qu'un point arbitraire reste sur cette droite indépendamment des forces externes



Correction des positions - Méthode de Newton-Raphson

Nous voulons minimiser $C(x) = 0$

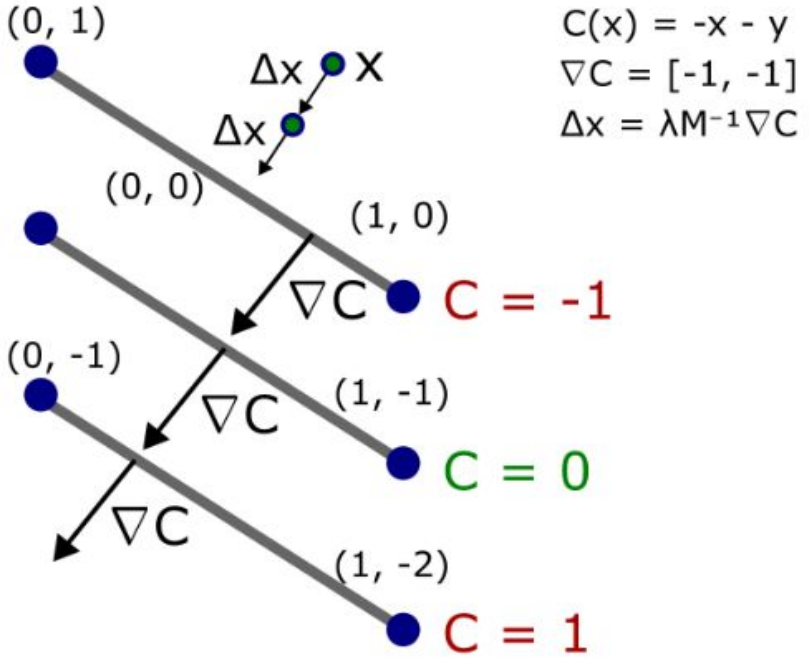
Pour ce faire, on voyage dans le sens du gradient de $C(\mathbf{x})$: $\nabla C(\mathbf{x})$:



Correction des positions - Méthode de Newton-Raphson

Si on positionne un point x sur ce graphique et qu'on souhaite qu'il reste ensuite sur C(x)

Il va voyager dans la direction du gradient de C(x) pour revenir le plus rapidement possible sur la ligne



Gradient d'une fonction

Soit une fonction f à plusieurs variables : f(x,y,z)

Gradient de f = vecteur des dérivées partielles de la fonction f : $\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$

Interprétation du gradient :

Vecteur gradient est la direction et le taux d'augmentation le plus rapide d'une fonction f

Autrement dit le gradient nous indique la direction pour maximiser la valeur de f à chaque entrée donnée

Problème d'optimisation sous contraintes

Objectif : maximiser ou minimiser une fonction de plusieurs variables f(x, y, z, …)

Fonction contrainte de la forme g(x, y, z, …) = c

recherche du point où f(x,y,z,...) et g(x,y,z,...) = c sont tangents

Méthode : trouver les valeurs maximales des variables x, y, z, … en utilisant les gradients

Difficulté : magnitudes de ∇f(x,y,z,..) et ∇g(x,y,z,...) ne sont pas pareils

Correction de cette différence avec un scalaire **λ** appelé **multiplicateur de Lagrange** tel que :

$$\nabla f(x,y,z,..) = \lambda \nabla g(x,y,z,...)$$