Algorithmique & Programmation Orientée Objet Semestre 2 ST

Les matrices de variables, applications pratiques

Problématique

- Emploi de "matrices" de variables dans le cadre de nombreux problèmes
- Matrice: Tableau à deux indices

Déclaration de tableaux à deux indices en langage algorithmique et en langage Java

- Chapitre "Tableaux de variables" pour la syntaxe en langage algorithmique ainsi que pour la syntaxe en langage Java.

Matrices pour les Mathématiques

- Matrice: Tableau de variables à deux indices
  - Composantes du même type
  - Premier indice: Numéro de ligne
  - Second indice: Numéro de colonne

x x x x x

x x x x x

x x x x x

nombre lignes = 3, nombre colonnes = 5

- Vecteur: Tableau de variables à deux indices dont l'une des 2 tailles est égale à 1
  - Matrice n*1
  - Matrice 1*n

x

x

x

Matrice 3*1
nombre lignes = 3, nombre colonnes = 1

x x x x x

Matrice 1*3
nombre lignes = 1, nombre colonnes = 5

- Matrices et vecteurs: "Objets" mathématiques de même catégorie: Tableaux à 2 dimensions
  - Si tailles compatibles, objets associables au moyen d'opérations matricielles:
    - Addition
    - Soustraction
    - Multiplication (composition)

- En programmation informatique, vecteur usuellement géré de manière simplifiée en utilisant un tableau à une dimension
  -> Pas de second indice toujours égal à 0 à "traîner"

- Matrice carrée: Matrice avec des nombres de lignes et de colonnes identiques

- En mathématiques utilisation fréquente des notions de matrice et de vecteur:
  - Ensemble de valeurs:
    Un tableau unidimensionnel de taille arbitraire
    Exemple: Un ensemble de lettres { a,v,r,z,h,c,s }

```
char [] ev = { 'a','v','r','z','h','c','s' };
```

  - Position dans un espace:
    Un tableau unidimentionnel de taille égale à la dimension de l'espace

Exemples: Coordonnées 2D $\begin{pmatrix} x \\ y \end{pmatrix}$, coordonnées 3D $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$

```
double [] c2D = new double[2];
double [] c3D = new double[3];
```

- Transformation géométrique:
  Un tableau bidimentionnel carré de taille égale à la dimension de l'espace
  Exemple: Rotation 3D d'angle Θ_y autour de l'axe Oy $\begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix}$

```
double [][] rot3D = { {  Math.cos(thetay),0.0,Math.sin
(thetay) },
                      {
0.0,1.0,              0.0 },
                      { -Math.sin(thetay),0.0,Math.cos
(thetay) } };
```

- Equation linéaire:
  Un tableau unidimentionnel de taille égale à 1 + le nombre de variables de l'équation linéaire
  Exemple: Equation d'un plan dans un espace 3D a x + b y + c z + d = 0

```
double [] equationLineaire = { a, b, c, d };
```

- Système de ne équations linéaires à nv inconnues:
  Un tableau bidimentionnel de taille ne*(nv+1) ou un tableau bidimentionnel de taille ne*nv et un tableau unidimentionnel de taille ne

  Exemple: Equation d'une droite dans un espace 3D
  $\begin{cases} a_1x + b_1y + c_1z + d_1 = 0 \\ a_2x + b_2y + c_2z + d_2 = 0 \end{cases}$

```
double [][] systemeEquationsLineaires = { { a1, b1, c1,
d1 },
                                          { a2, b2, c2,
d2 } };
```

- ...

- Utilisation fréquente de matrices carrées

- **Remarque**: Utilisation de tableaux et de matrices possiblement contradictoire avec la règle consistant à définir des types agrégés pour structurer le stockage des données
  -> Arbitrer entre faciliter la lecture et la compréhension et faciliter le développement dans le cadre de l'utilisation de formules mathématiques avec indices

Exemples

- **Addition de deux matrices**

- On considère deux matrices M1 et M2 de tailles identiques n*m.
  La matrice M somme de M1 et M2 est la matrice formée des composantes $m_{ij} = m1_{ij}+m2_{ij}$ pour i de 1 à n et j de 1 à m.

Exemple:
$$\begin{pmatrix} 2 & 3 & 5 & 1 \\ -1 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 5 & 2 & 5 & 1 \\ -1 & 4 & 5 & 0 \\ 0 & 4 & 0 & 4 \\ 1 & 5 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 5 & 10 & 2 \\ -2 & 8 & 8 & 0 \\ 0 & 4 & 0 & 4 \\ 2 & 7 & 0 & 0 \end{pmatrix}$$

```
{ Action de calcul de la somme de deux          }
{ matrices d'entiers de tailles compatibles     }
{ m1 : La première matrice à sommer             }
{ m2 : La seconde matrice à sommer              }
{ ms : La matrice résultat                      }

action somme(-> entier [][] m1,
             -> entier [][] m2,
             entier [][] ms ->)
   entier i,j
   pour i de 0 à longueur(1,m1)-1 faire
     pour j de 0 à longueur(2,m1)-1 faire
      ms[i][j] <- m1[i][j]+m2[i][j]
     fait
   fait
finAction

{ Fonction de calcul et retour                  }
{ de la somme de 2 matrices d'entiers           }
{ de tailles compatibles                        }
{ m1 : La première matrice à sommer             }
{ m2 : La seconde matrice à sommer              }

entier [][] somme(-> entier [][] m1,
                  -> entier [][] m2)
   entier [longueur(1,m1)][longueur(2,m1)] ms
  somme(m1,m2,ms)
   retourner ms
finFonction
```

```
/* Fonction de calcul de la somme               */
/* de 2 matrices de int de tailles compatibles  */
/* m1 : La première matrice à sommer            */
/* m2 : La seconde matrice à sommer             */
/* ms : La matrice résultat                     */
```

```
static void somme(int [][] m1,int [][] m2,int [][] ms) {
   for ( int i = 0 ; i < m1.length ; i++ ) {
     for ( int j = 0 ; j < m1[0].length ; j++ ) {
      ms[i][j] = m1[i][j]+m2[i][j]; } }
}

/* Fonction de calcul et retour de la somme     */
/* de 2 matrices de int de tailles compatibles */
/* m1 : La première matrice à sommer            */
/* m2 : La seconde matrice à sommer             */

static int [][] somme(int [][] m1,int [][] m2) {
   int [][] ms = new int[m1.length][m1[0].length];
  somme(m1,m2,ms);
   return ms;
}
```

SommeMatrices.lda
SommeMatrices.java
Exemple d'exécution

- **Mathématiques matricielles: Produit matrice par vecteur**

- On considère un vecteur $\vec{V}$ de taille n et une matrice carrée M de taille n*n.
  Le vecteur $\vec{W}$ produit de M par $\vec{V}$ (noté $\vec{W} = M.\vec{V}$) est calculé selon la formule:
  $w_i = \sum_{k=1}^{n} m_{ik} v_k$ pour i de 1 à n
  où les w_i sont les composantes du vecteur $\vec{W}$, les m_{ik} sont les composantes de la matrice M et les v_k sont les composantes du vecteur $\vec{V}$.

- Présentation intuitive du calcul du produit matrice-vecteur: La composante i du vecteur résultat est le produit de la $i^{ème}$ ligne de la matrice par le vecteur $\vec{V}$.

Exemple: $\begin{pmatrix} 2 & 3 & 5 & 1 \\ -1 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 4 \\ -2 \\ 0,5 \end{pmatrix} = \begin{pmatrix} 2,5 \\ 10 \\ 0 \\ 8 \end{pmatrix}$

```
{ Fonction de calcul et retour du produit       }
{ d'une matrice carree de reel de taille n.n    }
{ n quelconque                                  }
{ par un vecteur (tableau) de reel de taille n  }
{ Le retour est un tableau de reel de taille n  }
```

```
{ m : La matrice de reel                                    }
{ v : Le vecteur de reel                                    }

reel [] fonction produitMatriceVecteur(-> reel [][] m,
                                       -> reel [] v)
   entier n <- longueur(v)
   reel [n] w
   entier i
   entier j
   pour i de 0 à n-1 faire
    w[i] <- 0.0
     pour j de 0 à n-1 faire
      w[i] <- w[i] + m[i][j]*v[j]
     fait
   fait
   retourner w
fin fonction
```

```
/* Fonction de calcul et retour du produit       */
/* d'une matrice carree de reel de taille n.n    */
/* par un vecteur (tableau) de reel de taille n  */
/* n quelconque                                  */
/* Le retour est un tableau de reel de taille n  */
/* m : La matrice de reel                        */
/* v : Le vecteur de reel                        */

static double [] produitMatriceVecteur(double [][] m,
                                       double [] v) {
   int n = v.length;
   double [] w = new double[n];
   for ( int i = 0 ; i < n ; i++ ) {
    w[i] = 0.0;
     for ( int j = 0 ; j < n ; j++ ) {
      w[i] = w[i] + m[i][j]*v[j]; } }
   return w;
}
```

ProduitMatriceVecteur.lda
ProduitMatriceVecteur.java
Exemple d'exécution

- **Mathématiques matricielles: Produit matrice par matrice**

- On considère une matrice M1 de taille n*m et une matrice M2 de taille m*p. La matrice M produit de M1 par M2 (noté M = M1.M2) est une matrice de taille n*p et est calculée au moyen de la formule:

$$m_{ij} = \sum_{k=1}^{m} m1_{ik} m2_{kj} \text{ pour i de 1 à n et j de 1 à p}$$

où les m_{ij} sont les composantes de la matrice M, les $m1_{ik}$ sont les composantes de la matrice M1 et les $m2_{kj}$ sont les composantes de la matrice M2.

- Présentation intuitive du calcul du produit matrice-matrice: La composante ij de la matrice résultat est le produit de la $i^{ème}$ ligne de la première matrice par la $j^{ème}$ colonne de la seconde matrice.

Exemple: $\begin{pmatrix} 1 & -1 & 2 & -3 \\ 0 & 0 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 5 & 0 & 1 \\ 4 & 0 & 0 \\ 3 & 1 & 2 \\ -2 & 3 & -1 \end{pmatrix} = \begin{pmatrix} 13 & -7 & 8 \\ -1 & 7 & 0 \end{pmatrix}$

```
{ Fonction de calcul et retour du produit     }
{ d'une matrice de reel de taille n.m         }
{ par une matrice de reel de taille m.p       }
{ Le retour est une matrice de reel           }
{ de taille n.p                               }
{ n, m et p quelconques                       }
{ m1 : La première matrice de reel            }
{ m2 : La seconde matrice de reel             }

reel [][] fonction produitMatriceMatrice(-> reel [][] m1,
                                         -> reel [][] m2)
   entier n <- longueur(1,m1)
   entier m <- longueur(2,m1)
   entier p <- longueur(2,m2)
   reel [n][p] r
   entier i
   entier j
   entier k
   pour i de 0 à n-1 faire
     pour j de 0 à p-1 faire
      r[i][j] <- 0.0
       pour k de 0 à m-1 faire
        r[i][j] <- r[i][j] + m1[i][k]*m2[k][j]
       fait
     fait
   fait
   retourner r
fin fonction
```

```
/* Fonction de calcul et retour du produit     */
/* d'une matrice de double de taille n.m       */
/* par une matrice de double de taille m.p     */
/* Le retour est une matrice de double         */
/* de taille n.p                               */
/* n, m et p quelconques                       */
```

```
/* m1 : La première matrice de double           */
/* m2 : La seconde matrice de double            */

static double [][] produitMatriceMatrice(double [][] m1,
                                         double []
[] m2) {
   int n = m1.length;
   int m = m2.length;
   int p = m2[0].length;
   double [][] r = new double[n][p];
   for ( int i = 0 ; i < n ; i++ ) {
     for ( int j = 0 ; j < p ; j++ ) {
      r[i][j] = 0.0;
       for ( int k = 0 ; k < m ; k++ ) {
        r[i][j] = r[i][j] + m1[i][k]*m2[k][j]; } } }
   return r;
}
```

ProduitMatriceMatrice.lda
ProduitMatriceMatrice.java
Exemple d'exécution

- **Algèbre linéaire: Résolution d'un système de n équations linéaires à n inconnues**

- On considère le système de n équations linéaires à n inconnues:

$$\begin{cases} a_{11} * v_1 + a_{12} * v_2 + \dots + a_{1n} * v_n = b_1 \\ a_{21} * v_1 + a_{22} * v_2 + \dots + a_{2n} * v_n = b_2 \\ \\ a_{n1} * v_1 + a_{n2} * v_2 + \dots + a_{nn} * v_n = b_n \end{cases}$$

où les inconnues sont les v_i et les constantes sont les a_{ij} et les b_i.

- Présentation possible sous la forme matricielle A.V = B:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

où A est une matrice carrée de taille n*n, V et B sont des vecteurs de taille n.

- Résolution de ce système:
  - Détermination du vecteur des v_i qui vérifie l'ensemble de ces équations et donc du vecteur V qui vérifie le produit matriciel.

- Trois possibilités: V n'existe pas, V existe et est unique, V existe et il y en a une infinité.
- Exemples d'applications pratiques:
  - En deux dimensions: Trouver la position de l'intersection de deux droites
  - En trois dimensions: Trouver la position de l'intersection entre une droite et un plan
  - En dimension n avec n très grand (> 1000000): En informatique graphique, calculer l'éclairage d'une scène par radiosité

- Méthode de résolution par pivot de Gauss

  - Une première étape de transformation par "triangularisation":
  - Transformation de la matrice A par traitement (substitution) de ses lignes de manière à remplir sa 1/2 matrice triangulaire inférieure stricte avec des 0.0
  - Transformation concourante du vecteur B pour que le système d'équations A.V = B reste équivalent
    - n-1 itérations numérotées i consistant à soustraire à chaque ligne j à partir de la ligne i+1 le produit de la ligne i par le facteur fact tel que fact*m[i][i] = m[i][j] -> fact = m[i][j]/m[i][i]
    - Calcul de fact impossible si, lors d'une itération, m[i][i] = 0.0 (division par zéro)
      -> Permutation la ligne i avec la première ligne k où m[k][i] est différent de 0.0
      -> Si tous les m[k][i] égaux à 0.0 alors pas de solution unique (aucune solution ou une infinité de solutions)

Exemple:

Les matrices A et B sont $\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 4.0 & 0.0 & 5.0 & 1.0 & -1.0 \\ 3.0 & 1.0 & 2.0 & 0.0 & 3.0 \\ -2.0 & 3.0 & -2.0 & 16.0 & 3.0 \\ 6.0 & 5.0 & 0.0 & -3.0 & 19.0 \end{bmatrix}$ et $\begin{bmatrix} 1.0 \\ 2.0 \\ -1.0 \\ 4.0 \\ 3.0 \end{bmatrix}$.

1. Etape n°1

I. Traitement de la ligne 2 par soustraction de 4.0 fois la ligne 1:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 3.0 & 1.0 & 2.0 & 0.0 & 3.0 \\ -2.0 & 3.0 & -2.0 & 16.0 & 3.0 \\ 6.0 & 5.0 & 0.0 & -3.0 & 19.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -2.0 \\ -1.0 \\ 4.0 \\ 3.0 \end{bmatrix}.$$

II. Traitement de la ligne 3 par soustraction de 3.0 fois la ligne 1:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ -2.0 & 3.0 & -2.0 & 16.0 & 3.0 \\ 6.0 & 5.0 & 0.0 & -3.0 & 19.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -2.0 \\ -4.0 \\ 4.0 \\ 3.0 \end{bmatrix}.$$

III. Traitement de la ligne 4 par soustraction de -2.0 fois la ligne 1:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 14.0 & 5.0 \\ 6.0 & 5.0 & 0.0 & -3.0 & 19.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -2.0 \\ -4.0 \\ 6.0 \\ 3.0 \end{bmatrix}.$$

IV. Traitement de la ligne 5 par soustraction de 6.0 fois la ligne 1:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 14.0 & 5.0 \\ 0.0 & 5.0 & -6.0 & 3.0 & 13.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -2.0 \\ -4.0 \\ 6.0 \\ -3.0 \end{bmatrix}.$$

2. Etape n°2

I. Permutation des lignes 2 et 3 car la composante d'indice (2,2) est égale à 0.0:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 3.0 & 0.0 & 14.0 & 5.0 \\ 0.0 & 5.0 & -6.0 & 3.0 & 13.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -4.0 \\ -2.0 \\ 6.0 \\ -3.0 \end{bmatrix}.$$

II. Traitement des lignes 4 et 5:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 0.0 & 3.0 & 5.0 & 5.0 \\ 0.0 & 0.0 & -1.0 & -12.0 & 13.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -4.0 \\ -2.0 \\ 18.0 \\ 17.0 \end{bmatrix}.$$

3. Etape n°3

I. Traitement des lignes 4 et 5:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 0.0 & 0.0 & -10.0 & 20.0 \\ 0.0 & 0.0 & 0.0 & -7.0 & 8.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -4.0 \\ -2.0 \\ 24.0 \\ 15.0 \end{bmatrix}.$$

4. Etape n°4

I. Traitement de la ligne 5:

$$\begin{bmatrix} 1.0 & 0.0 & 1.0 & -1.0 & 1.0 \\ 0.0 & 1.0 & -1.0 & 3.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 5.0 & -5.0 \\ 0.0 & 0.0 & 0.0 & -10.0 & 20.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & -6.0 \end{bmatrix} \text{ et } \begin{bmatrix} 1.0 \\ -4.0 \\ -2.0 \\ 24.0 \\ -1.8 \end{bmatrix}.$$

- Une seconde (et dernière) étape: Calcul de V
  - v_n d'ores et déjà calculable car sur la ligne n de la matrice A, il ne reste que des 0.0 sauf pour la dernière valeur d'indice (n,n)
    -> $v_n = b_n/a_{nn}$
  - Calcul de v_{n-1} possible car, v_n étant connu, toutes les valeurs nécessaires à son calcul sont maintenant connues.
  - Calcul de v_{n-2} possible car, v_n et v_{n-1} étant connus, toutes les valeurs nécessaires à son calcul sont maintenant connues.
  - Poursuite du calcul des v du calcul des v_i jusqu'à v_1.

Sur l'exemple précédent:

- v_5 = -1.8/-6.0 = 0.3
- v_4 = (24.0 - 20.0*0.3)/-10.0 = -1.8
- v_3 = (-2.0 + 5.0*0.3 + 5.0*1.8)/1.0 = 8.5

- v_2 = (-4.0 + 0.0*0.3 + 3.0*1.8 + 1.0*8.5)/1.0 = 9.9
- v_1 = (1.0 - 1.0*0.3 - 1.0*1.8 - 1.0*8.5 + 0.0*9.9)/1.0 = -9.6

```
{ Resolution d'un systeme de n equations       }
{ lineaires a n inconnues                      }
{ par la methode du pivot de Gauss             }

{ Recherche et permutation entre lignes        }
{ de coefficients dans le cadre                }
{ de la résolution d'un système                }
{ de n équations linéaires à n inconnues       }
{ par pivot de Gauss                           }
{ l : Le numéro de la ligne à permuter         }
{ a : La matrice de réels à traiter            }
{ b : Le tableau de réels à traiter            }
{     concourrament à la matrice a             }

action permutation(-> entier l,
                   -> réel [][] a ->,
                   -> réel [] b ->)
   entier n <- longueur(b)
   entier i
   reel aux
   entier ll <- l
   tant que a[ll][l] == 0.0 faire
    ll <- ll+1
   fait
   pour i de l à n-1 faire
    aux <- a[l][i]
    a[l][i] <- a[ll][i]
    a[ll][i] <- aux
   fait
  aux <- b[l]
  b[l] <- b[ll]
  b[ll] <- aux
fin action

{ Première partie de la méthode de résolution }
{ d'un système de n équations linéaires        }
{ à n inconnues par pivot de Gauss:            }
{ Triangularisation de la matrice              }
{ des coefficients et modification             }
{ concourrante du vecteur                      }
{ a : La matrice de réels à transformer        }
{     par triangularisation                    }
{ b : Le tableau de réels modifié              }
{     concourrament à la matrice a             }

action transformation(-> réel [][] a ->,
```

```
                          -> réel [] b ->)
   entier n <- longueur(b)
   entier i
   entier j
   entier k
   reel facteur
   pour i de 1 à n-1 faire
     si a[i-1][i-1] == 0.0 alors
      permutation(i-1,a,b)
     fsi
     pour j de i à n-1 faire
      facteur <- a[j][i-1]/a[i-1][i-1]
       pour k de i-1 à n-1 faire
        a[j][k] <- a[j][k] - a[i-1][k]*facteur
       fait
      b[j] <- b[j] - b[i-1]*facteur
     fait
   fait
fin action

{ Seconde partie de la méthode de résolution  }
{ des systèmes de n équations linéaires       }
{ à n inconnues: Extraction du résultat       }
{ Le retour est un tableau de réels           }
{ a : La matrice de réels triangulaire carrée }
{ b : Le tableau de réels                     }

réel [] fonction extraction(-> réel [][] a,
                               -> réel [] b)
   entier n <- longueur(v)
   reel [n] v
   entier i
   entier j
  v[n-1] <- b[n-1]/a[n-1][n-1]
   pour i de n-2 à 0 pas -1 faire
    v[i] <- b[i]
     pour j de n-1 à j+1 pas -1 faire
      v[i] <- v[i] - v[j]*a[i][j]
     fait
    v[i] <- v[i]/a[i][i]
   fait
   retourner v
fin fonction

{ Fonction de calcul et retour de la solution }
{ du systeme d'équations linéaires a.v = b    }
{ a est une matrice de coefficients           }
{ b est un vecteur de coefficients            }
{ v est le vecteur à déterminer               }
{ a, b et v sont de taille n.n, n et n        }
```

```
{ n quelconque                                    }
{ Méthode utilisée : pivot de Gauss               }
{ a : La matrice de réels (carrée)                }
{ b : Le tableau de réels                         }

réel [] fonction resolution(-> réel [][] a,
                            -> réel [] b)
   reel [longueur(v)] v
  transformation(a,b)
  v <- extraction(a,b)
   retourner v
fin fonction
```

```
/* Fonction de calcul et retour d'une copie     */
/* d'un tableau de double a 1 indice            */
/* t : Le tableau à cloner                      */

static double [] clone(double [] t) {
   int n = t.length;
   double [] nt = new double[n];
   for ( int i = 0 ; i < n ; i++ ) {
    nt[i] = t[i]; }
   return nt;
}

/* Fonction de calcul et retour d'une copie     */
/* d'un tableau de double a 2 indices           */
/* t : La matrice à cloner                      */

static double [][] clone(double [][] t) {
   int n = t.length;
   int m = t[0].length;
   double [][] nt = new double[n][m];
   for ( int i = 0 ; i < n ; i++ ) {
     for ( int j = 0 ; j < m ; j++ ) {
      nt[i][j] = t[i][j]; } }
   return nt;
}

/* Recherche et permutation entre lignes        */
/* de coefficients dans le cadre                */
/* de la résolution d'un système                */
/* de n équations linéaires à n inconnues       */
/* par pivot de Gauss                           */
/* l : Le numéro de la ligne à permuter         */
/* a : La matrice de réels à traiter            */
/* b : Le tableau de réels à traiter            */
/*     concourrament à la matrice a             */

static void permutation(int l,double [][] a,double [] b) {
```

```
    int n = b.length;
    double aux;
    int ll = l;
    while ( a[ll][l] == 0.0 ) {
     ll++; }
    for ( int i = l ; i < n ; i++ ) {
     aux = a[l][i];
     a[l][i] = a[ll][i];
     a[ll][i] = aux; }
   aux = b[l];
   b[l] = b[ll];
   b[ll] = aux;
}

/* Première partie de la méthode de résolution */
/* d'un système de n équations linéaires       */
/* à n inconnues par pivot de Gauss:           */
/* Triangularisation de la matrice             */
/* des coefficients et modification            */
/* concourrante du vecteur                     */
/* a : La matrice de réels à transformer       */
/*     par triangularisation                   */
/* b : Le tableau de réels modifié             */
/*     concourrament à la matrice a            */

static void transformation(double [][] a,double [] b) {
    int n = b.length;
    for ( int i = 1 ; i < n ; i++ ) {
      if ( a[i-1][i-1] == 0.0 )
       permutation(i-1,a,b);
      for ( int j = i ; j < n ; j++ ) {
        double facteur = a[j][i-1]/a[i-1][i-1];
        for ( int k = i-1 ; k < n ; k++ ) {
         a[j][k] = a[j][k] - a[i-1][k]*facteur; }
       b[j] = b[j] - b[i-1]*facteur;  } }
}

/* Seconde partie de la méthode de résolution  */
/* des systèmes de n équations linéaires       */
/* à n inconnues: Extraction du résultat       */
/* Le retour est un tableau de réels           */
/* a : La matrice de réels triangulaire carrée */
/* b : Le tableau de réels                     */

static double [] extraction(double [][] a,double [] b) {
    int n = b.length;
    double [] v = new double[n];
   v[n-1] = b[n-1]/a[n-1][n-1];
    for ( int i = n-2 ; i >= 0 ; i-- ) {
      v[i] = b[i];
```

```
      for ( int j = n-1 ; j > i ; j-- ) {
        v[i] = v[i] - v[j]*a[i][j]; }
     v[i] = v[i]/a[i][i]; }
    return v;
}

/* Fonction de calcul et retour de la solution */
/* du systeme d'équations linéaires a.v = b    */
/* a est une matrice de coefficients           */
/* b est un vecteur de coefficients            */
/* v est le vecteur à déterminer               */
/* a, b et v sont de taille n.n, n et n        */
/* n quelconque                                */
/* Méthode utilisée : pivot de Gauss           */
/* Attention: Les deux tableaux a et b         */
/* sont modifiés au cours de la résolution     */
/* Le retour est un tableau de réels           */
/* a : La matrice de réels (carrée)            */
/* b : Le tableau de réels                     */

static double [] resolution(double [][] a,
                            double [] b) {
  transformation(a,b);
   return extraction(a,b);
}

/* Fonction de calcul et retour de la solution */
/* du systeme d'équations linéaires a.v = b    */
/* a est une matrice de coefficients           */
/* b est un vecteur de coefficients            */
/* v est le vecteur à déterminer               */
/* a, b et v sont de taille n.n, n et n        */
/* n quelconque                                */
/* Méthode utilisée : pivot de Gauss           */
/* Une copie des deux tableaux a et b          */
/* supports des coefficients est réalisée      */
/* pour ne pas détruire ces informations       */
/* au cours de la résolution                   */
/* Le retour est un tableau de réels           */
/* a : La matrice de réels (carrée)            */
/* b : Le tableau de réels                     */

static double [] resolutionGauss(double [][] a,
                                 double [] b) {
   return resolution(clone(a),clone(b));
}
```

PivotGauss.lda
PivotGauss.java
Exemple d'exécution

Matrices pour la Physique

- Phénomènes statiques ou en évolution au cours du temps
- Large appel à l'informatique comme outil de simulation
- Pourquoi simuler?
  - Moins cher que les tests réels (crash d'une automobile, analyse d'un écoulement turbulent, ...)
  - Réalisation de prévisions (météo, évolution climatique, crue, ...)
  - Analyse de phénomènes qu'il n'est pas possible de tester dans la réalité (essai nucléaire, circulation automobile dans une ville, évacuation d'un stade de football, formation du système solaire, explosion d'une étoile en super-novae, ...)
  - ...
- Phénomènes à simuler fréquement décrits par des équations (continues ou non) pour lesquelles il n'existe pas de méthode de résolution analytique (i.e. pas de solution présentable sous la forme d'une formule simple, voire complexe)
- Invention de méthodes numériques de résolution permettant de trouver une solution à partir d'une discrétisation du champ de travail
  - champ unidimentionnel: appel à des tableaux,
  - champ bidimentionnel: utilisation de matrices,
  - dimension 3: tableaux à 3 indices,

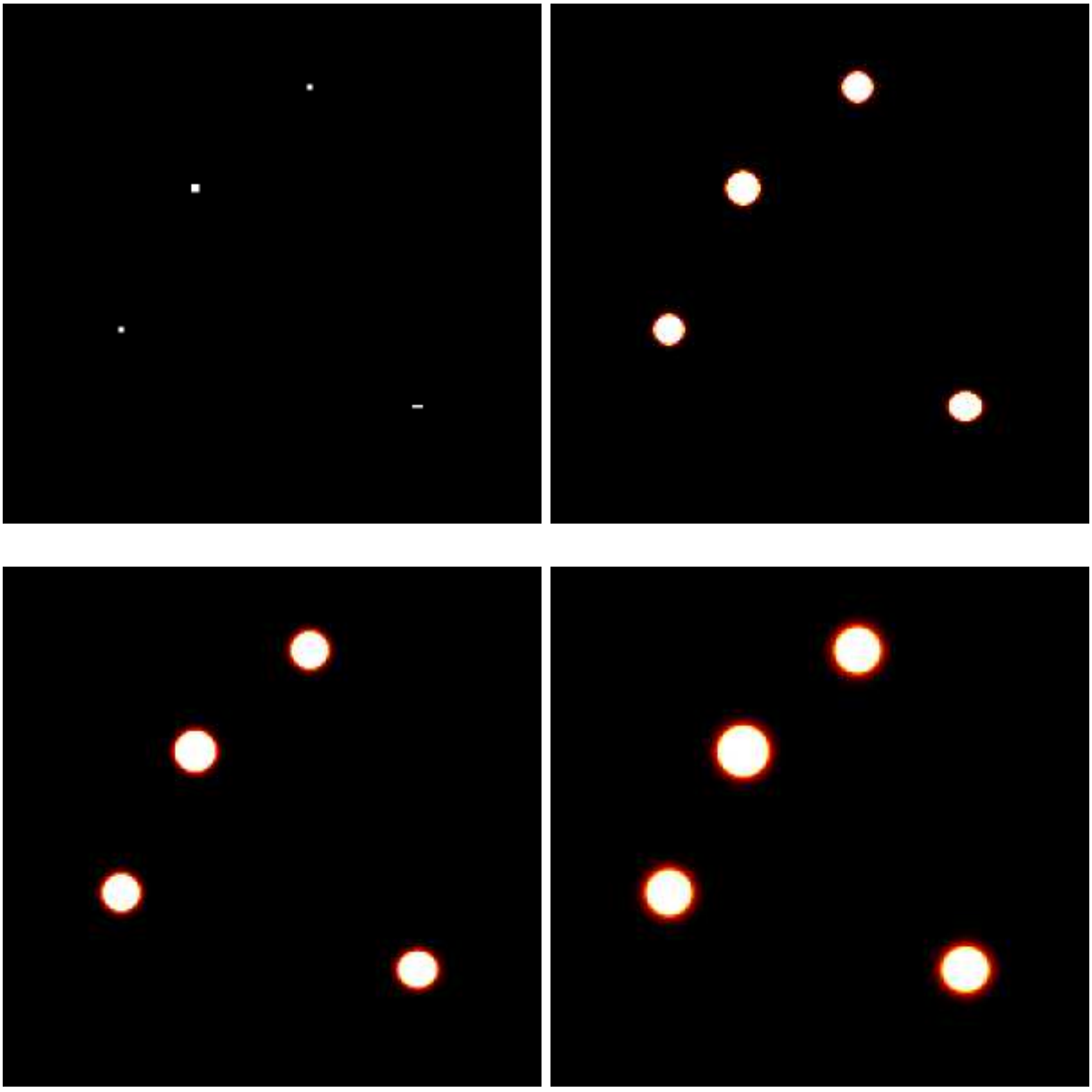
  et d'une discrétisation du temps s'il s'agit d'un problème à évolution temporelle.

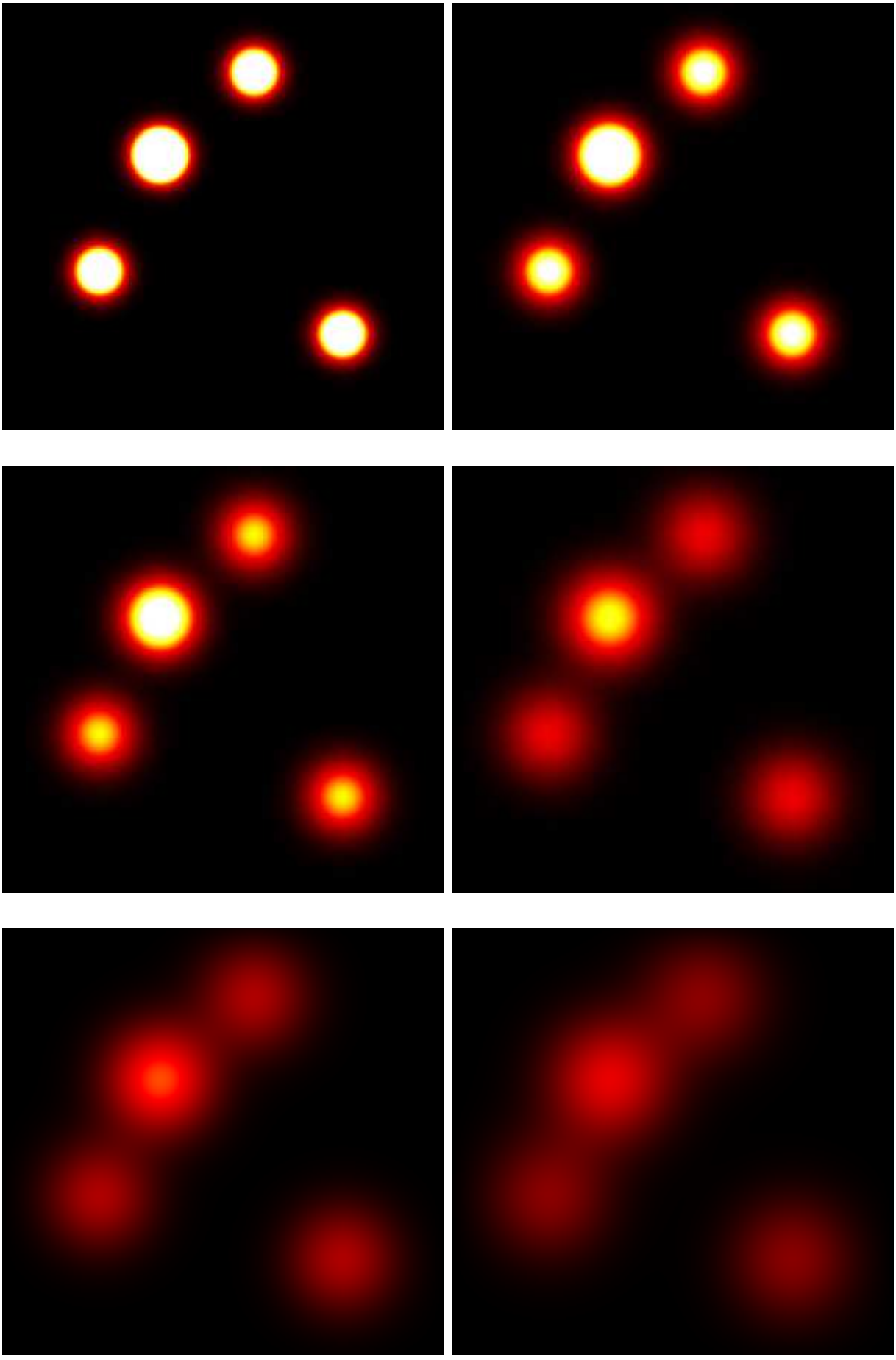
Exemples

- Propagation de la chaleur au sein d'une barre métallique
  - Problème en dimension 1
  - Décomposition (discrétisation) de la barre en une suite de morceaux pour chacun desquels la température est considérée uniforme
  - Valeurs stockées et gérées dans un tableau à 1 indice
  - Calcul des transferts de chaleur entre morceaux voisins
  - Evolution au cours du temps par calcul d'états successifs séparés dans le temps par un "pas de temps" généralement constant
  - Augmentation du pas de discrétisation spatial et/ou du pas de discrétisation temporel

-> Obtention de résultats plus précis au prix d'un temps de calcul plus long

- Propagation de la chaleur au sein d'une plaque métallique
  - Problème en 2 dimensions
  - Décomposition de la plaque en une grille de cellules pour chacune desquelles la température est considérée uniforme
  - Calcul des transferts de chaleur entre cellules voisines
  - Evolution au cours du temps par calcul d'états successifs séparés dans le temps par un "pas de temps" généralement constant
  - Valeurs stockées dans un tableau à 2 indices



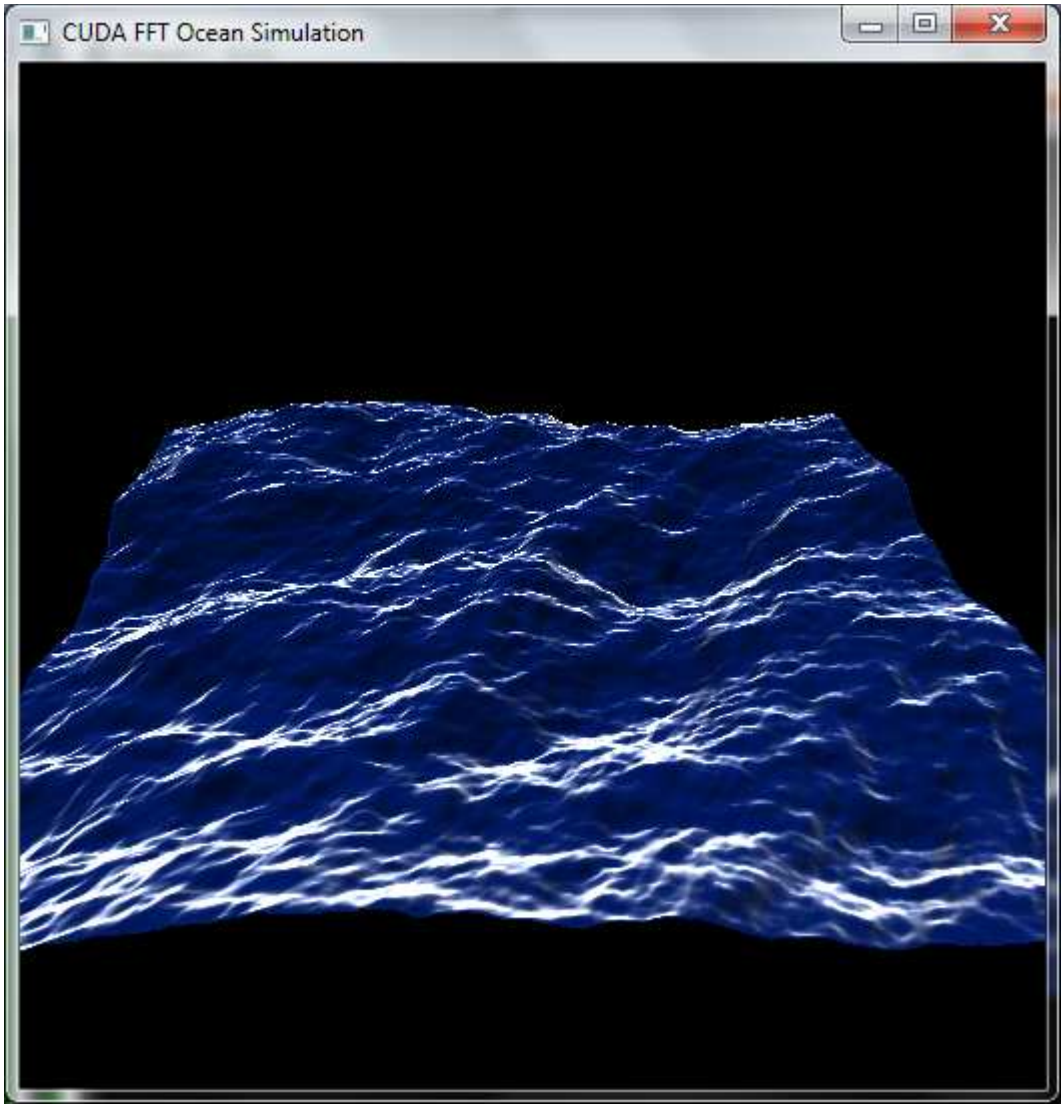


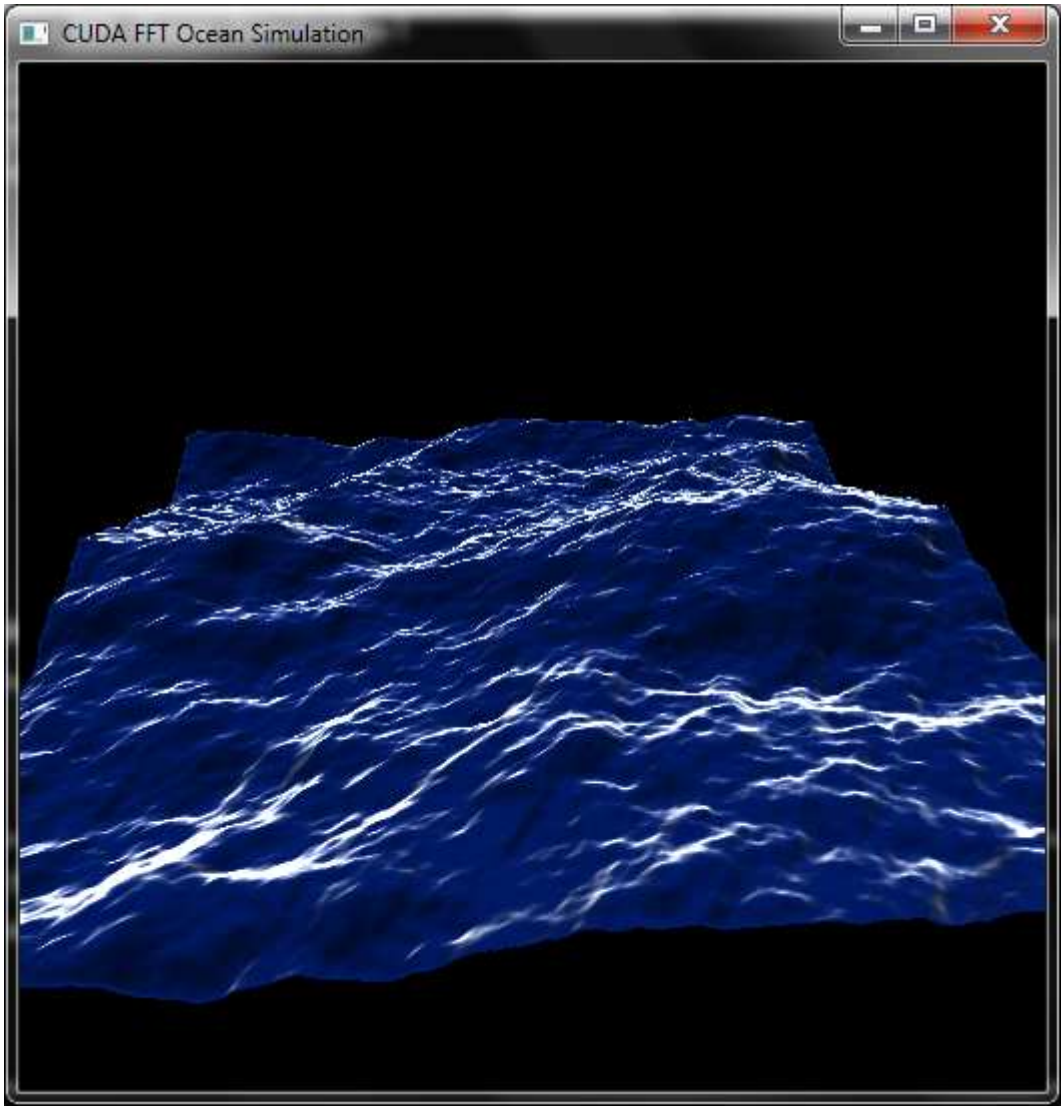
Simulation de l'évolution de la température d'une plaque métallique sur laquelle sont créés des points chauds

Exemple d'exécution

Simulation de la surface océanique

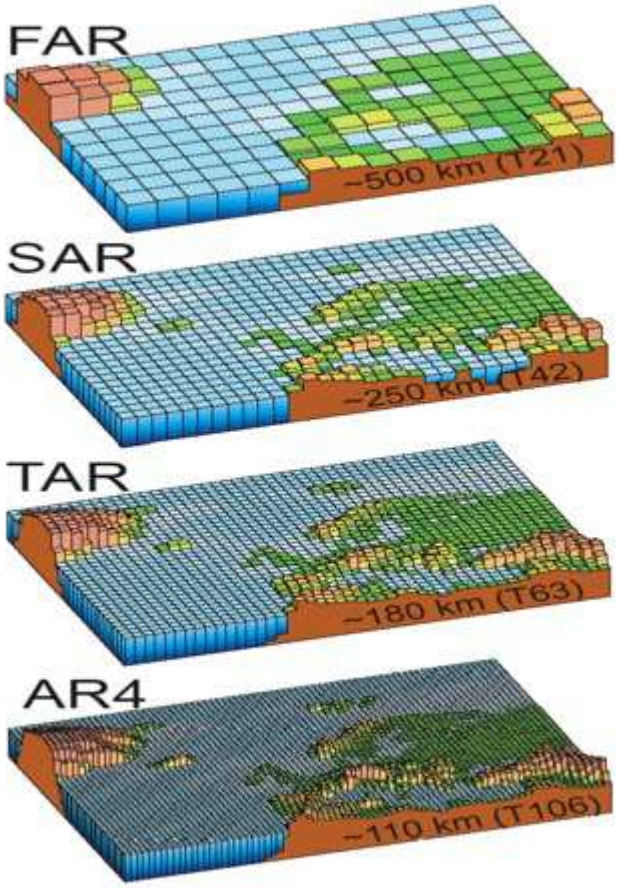
- Problème en 2 dimensions
- Simulation à base d'ensembles d'ondes sinusoïdales de longueurs d'onde et amplitudes différentes se déplaçant au cours du temps et s'additionnant les unes aux autres



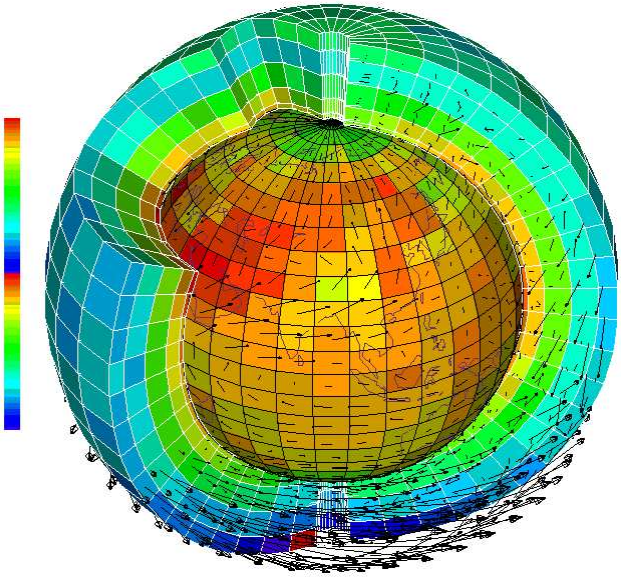


Simulation de surface océanique de surface océanique
Kit de développement CUDA (NVidia)

- Evolution climatique
  - Premiers modèles: Modèles en deux dimensions
  - Surface de la Terre décomposée en longitude et en latitude en un maillage "carré"
  - Informations relatives aux cellules de ce maillage stockées dans une matrice de variables
  - Etude de l'évolution du climat en considérant chaque cellule comme une entité atomique de caractéristiques uniformes
  - Modèles plus complexes: Trois dimensions

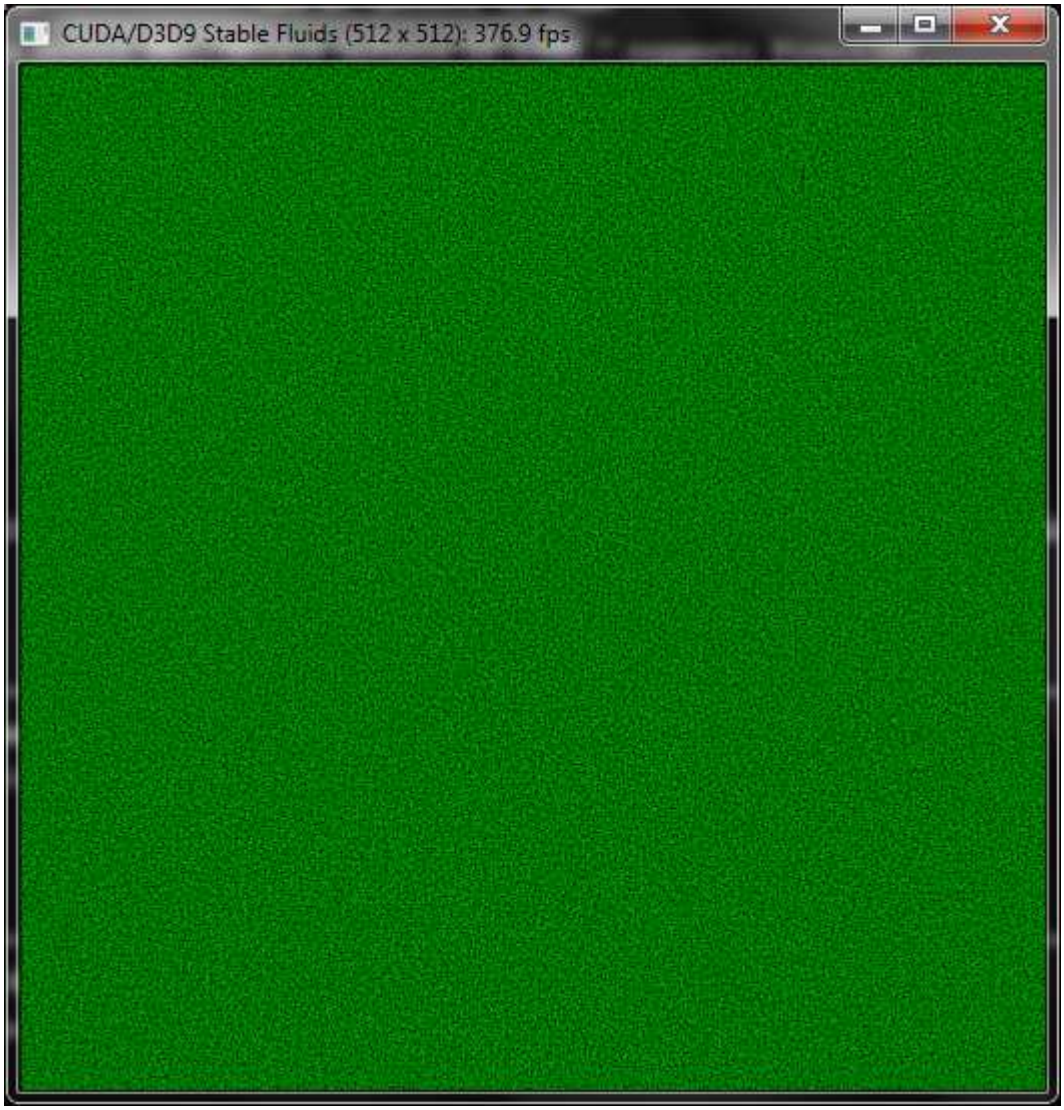


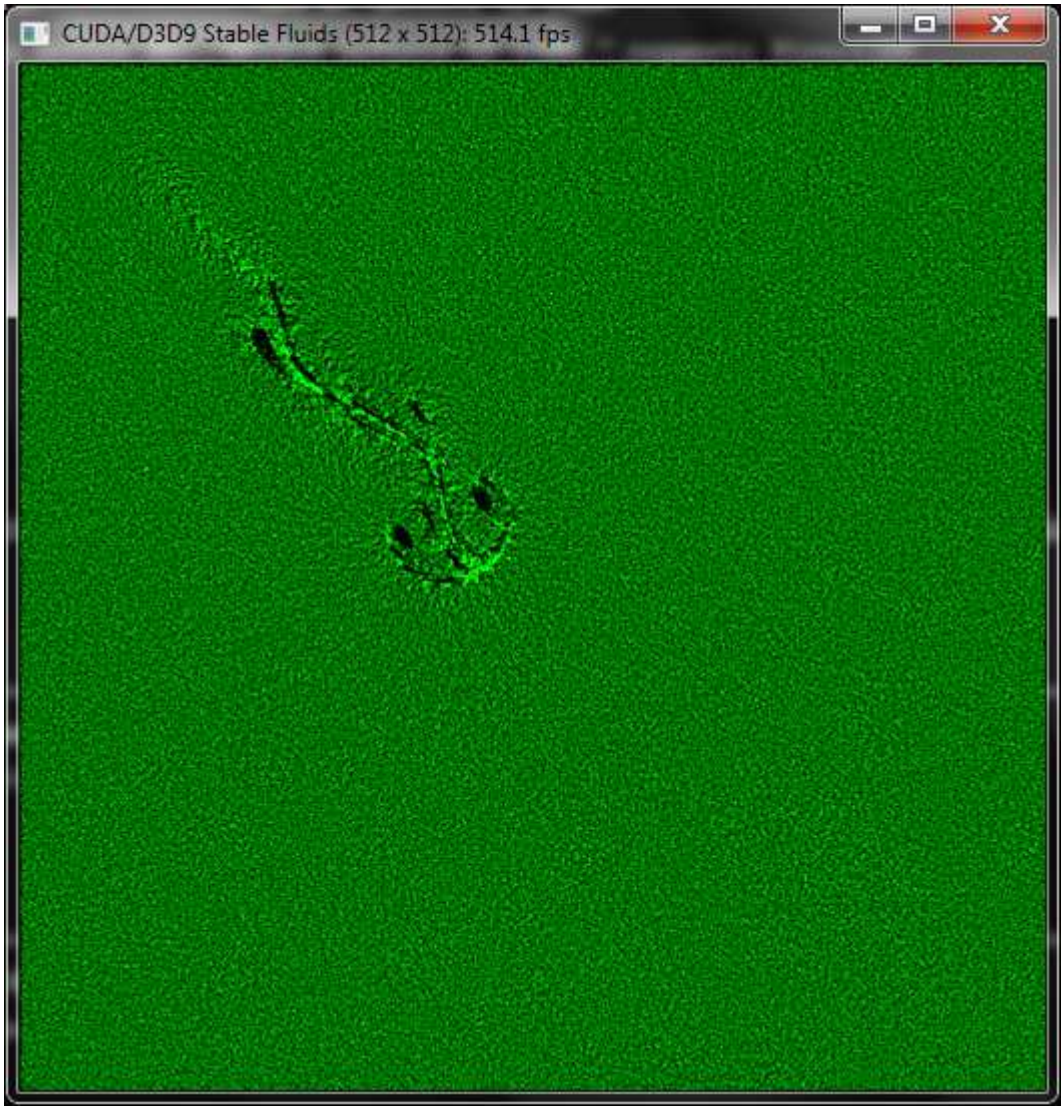
Décomposition de l'atmosphère en un maillage 2D de plus en plus fin
ces dernières années avec la disponibilité d'ordinateurs de plus en plus puissants

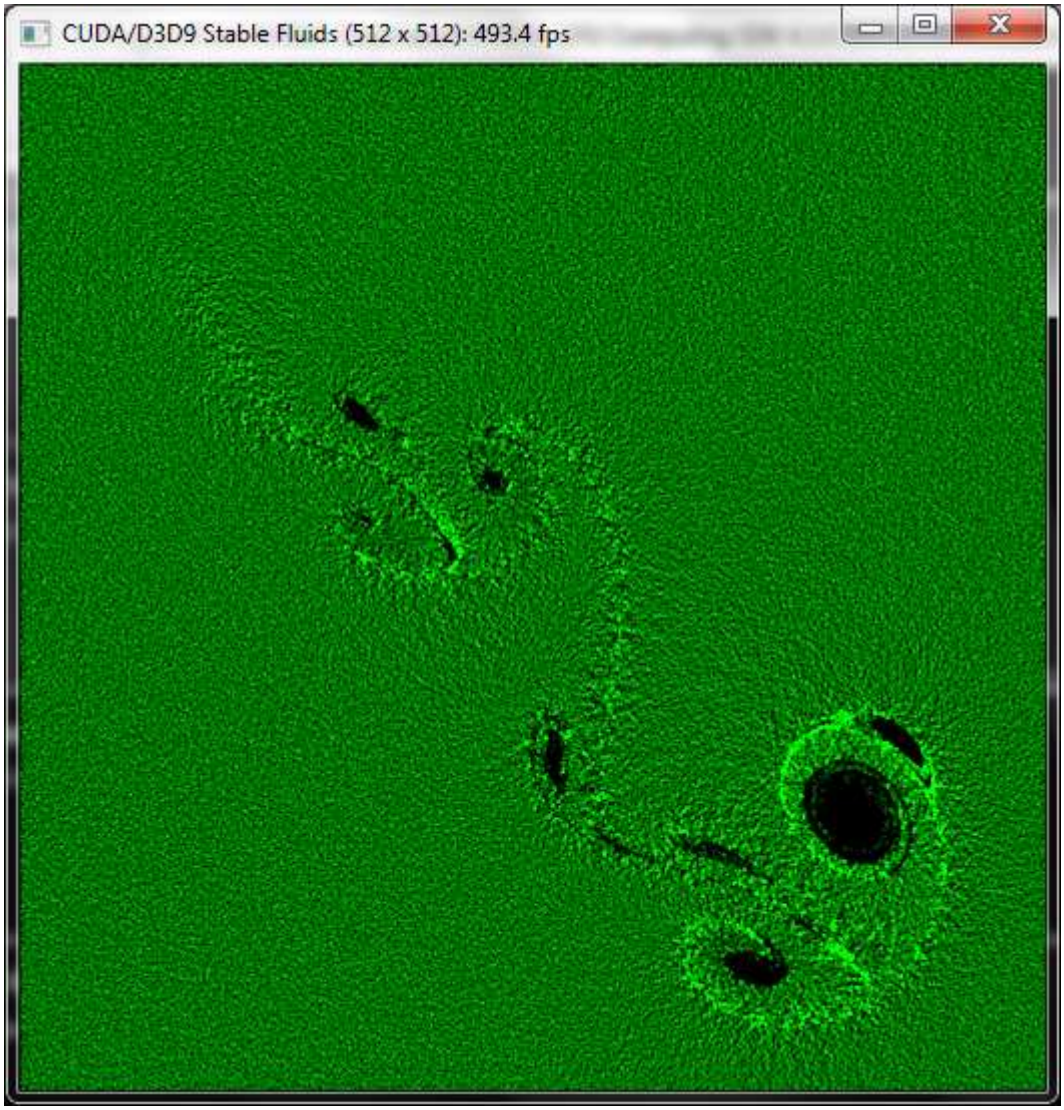


Décomposition de l'atmosphère en un maillage 3D formé de volumes élémentaires

- Analyse des turbulences dans un fluide 2D
  - Problème en deux dimensions
  - Résolution des équations de mécanique des fluides (Navier-Stockes) complexe et extrêmement coûteuse en temps de calcul

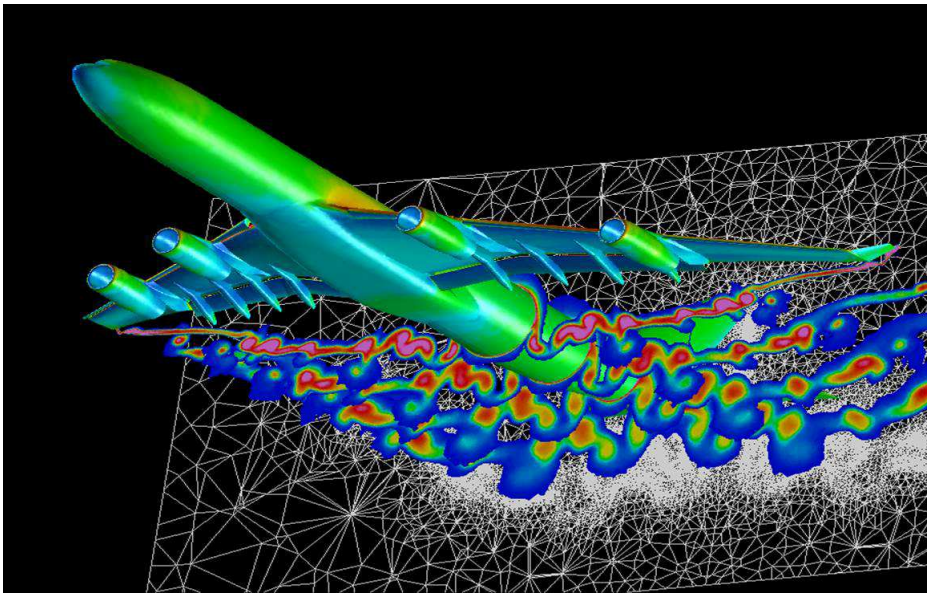

CUDA/D3D9 Stable Fluids (512 x 512): 376.9 fps


CUDA/D3D9 Stable Fluids (512 x 512): 514.1 fps

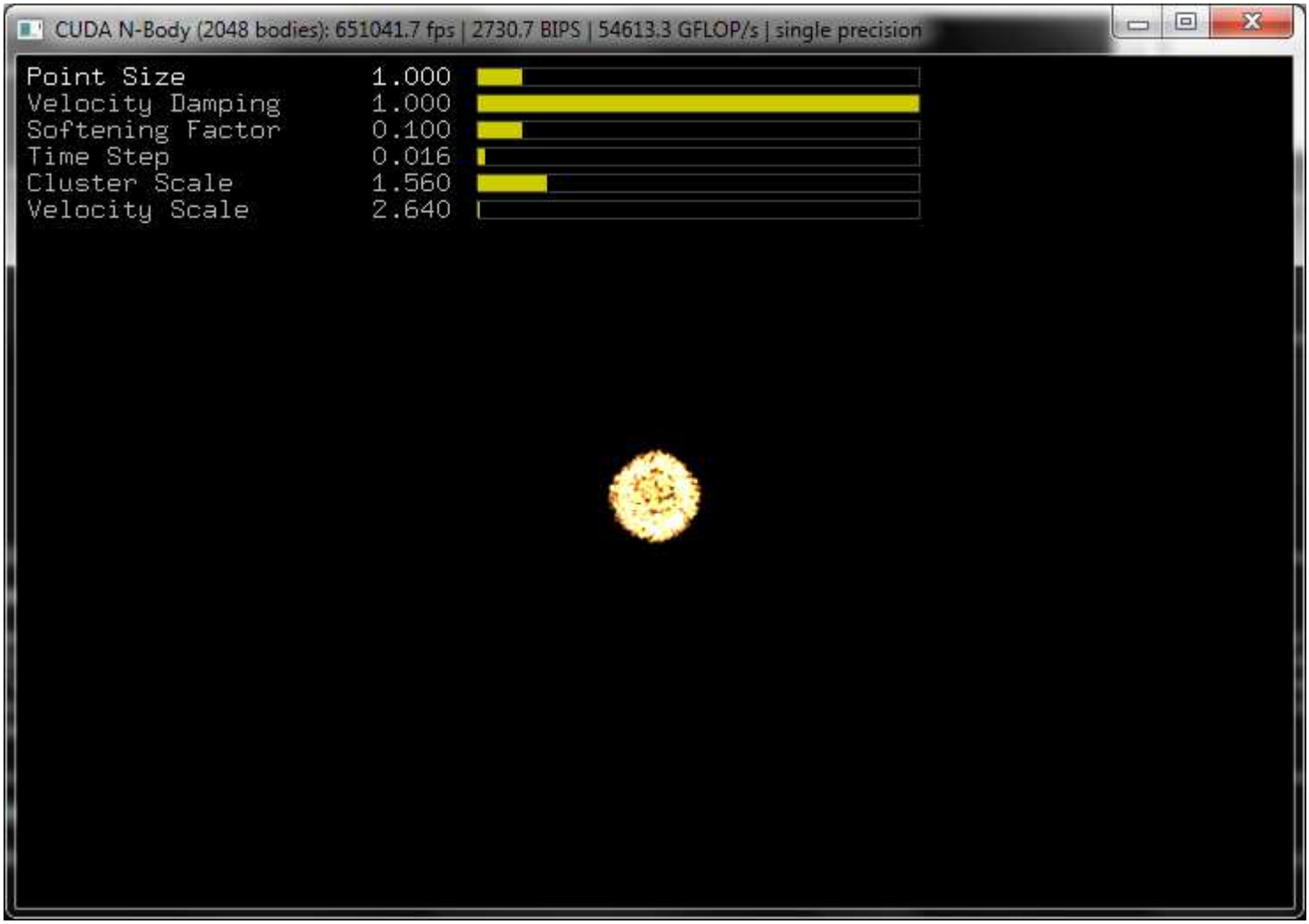


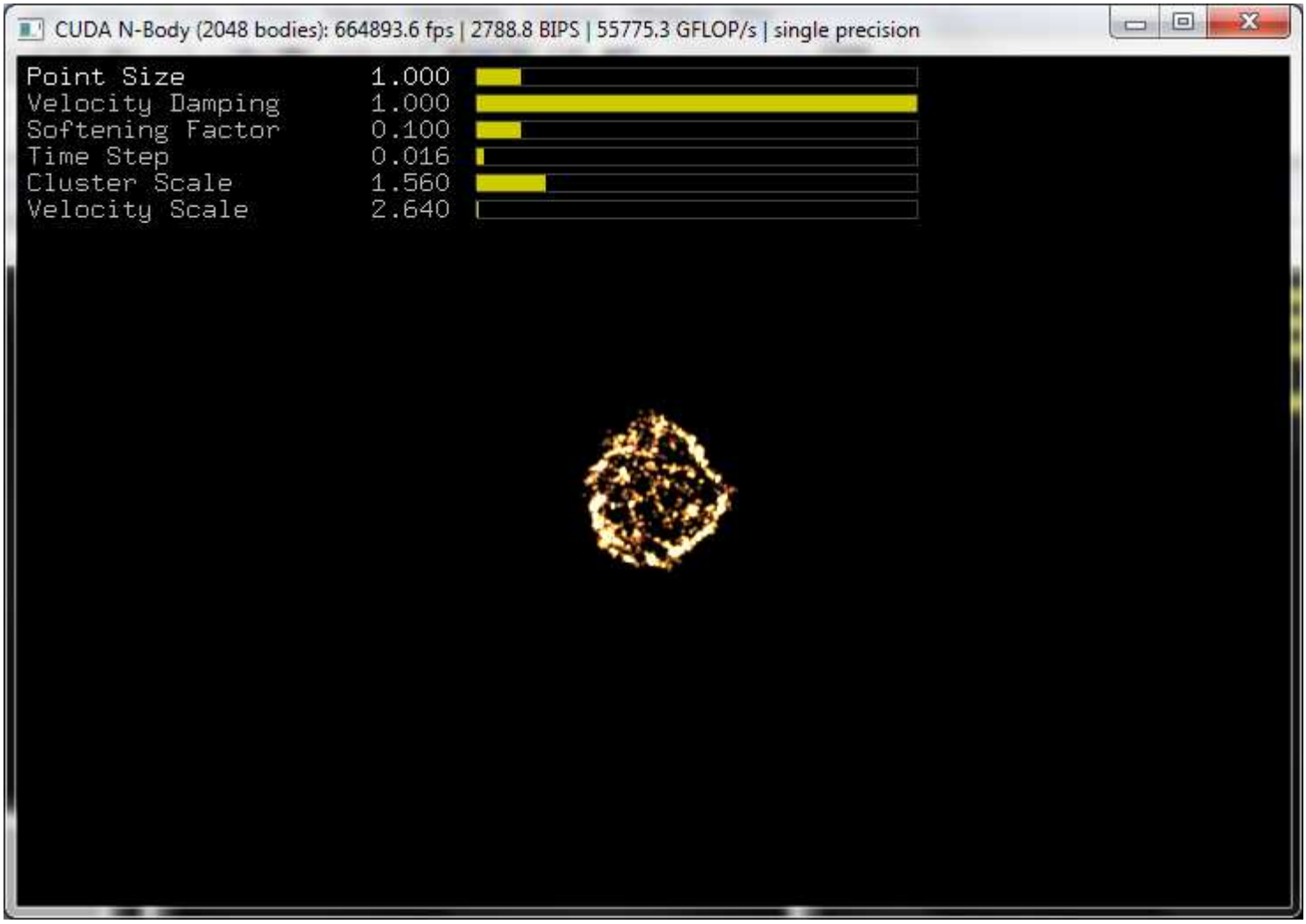
Simulation de fluides
Kit de développement CUDA (NVidia)

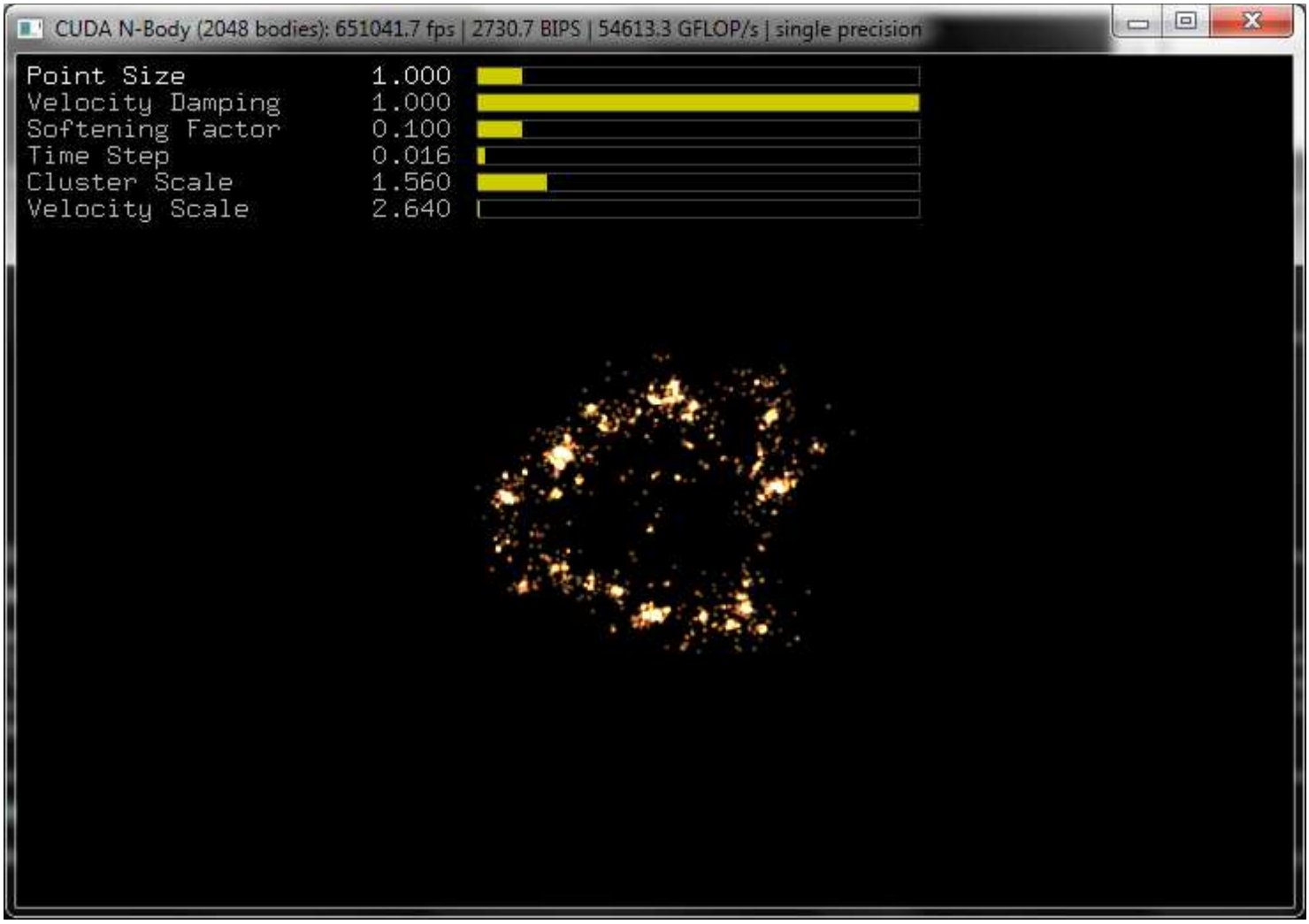
- Analyse des turbulences sur une aile d'avion
  - Problème en trois dimensions
  - Résolution des équations de mécanique des fluides (Navier-Stockes)
  - Historiquement, problème géré en dimension 2 avec simulation numérique sur des profils 2D de l'objet 3D
  - Accroissement de la puissance des ordinateurs -> Passage en 3D avec une aile complète
  - Bientôt possible de simuler l'écoulement sur un avion entier: ailes, volets, fuselage, gouvernes, pylônes, réacteurs, ...

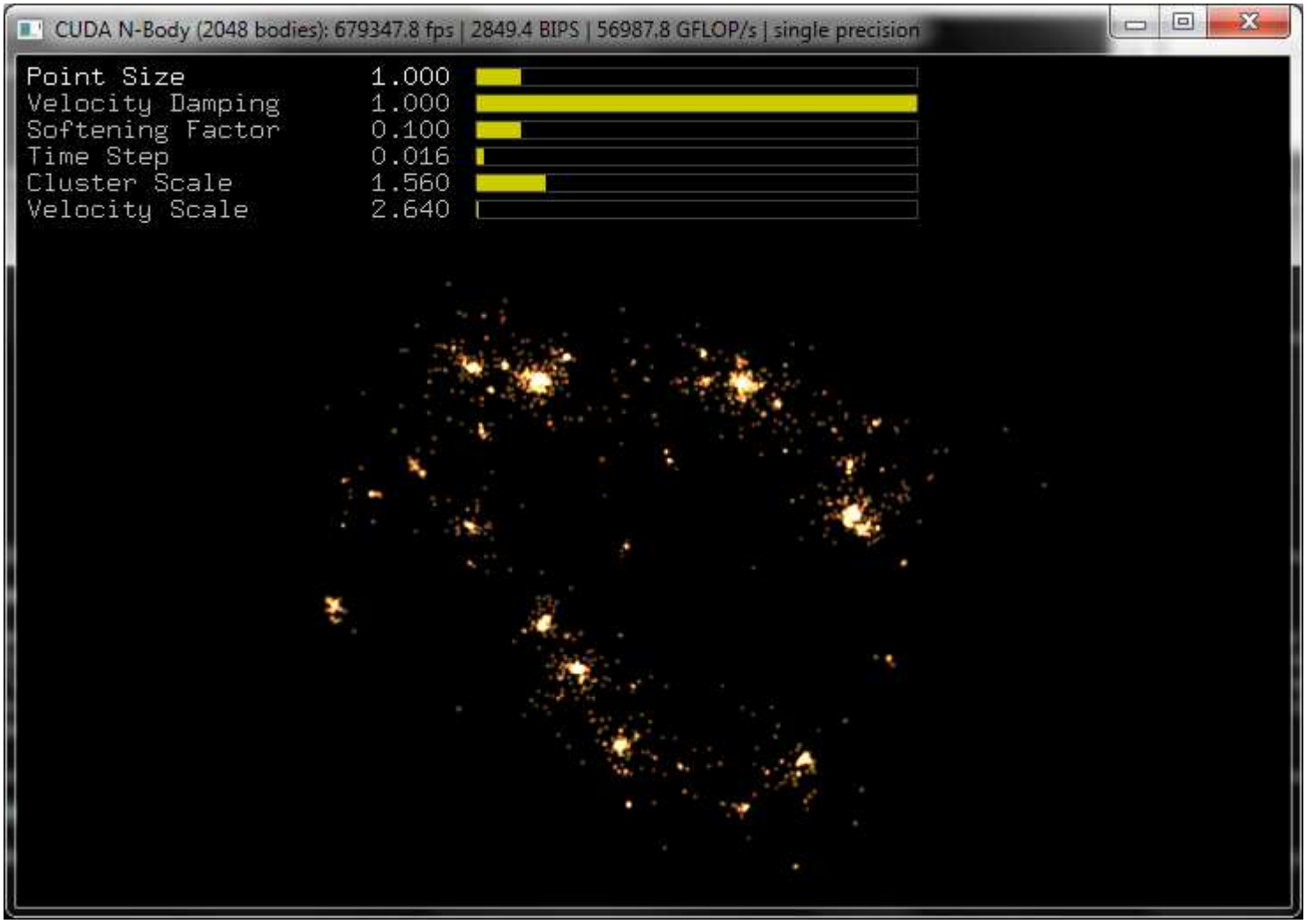


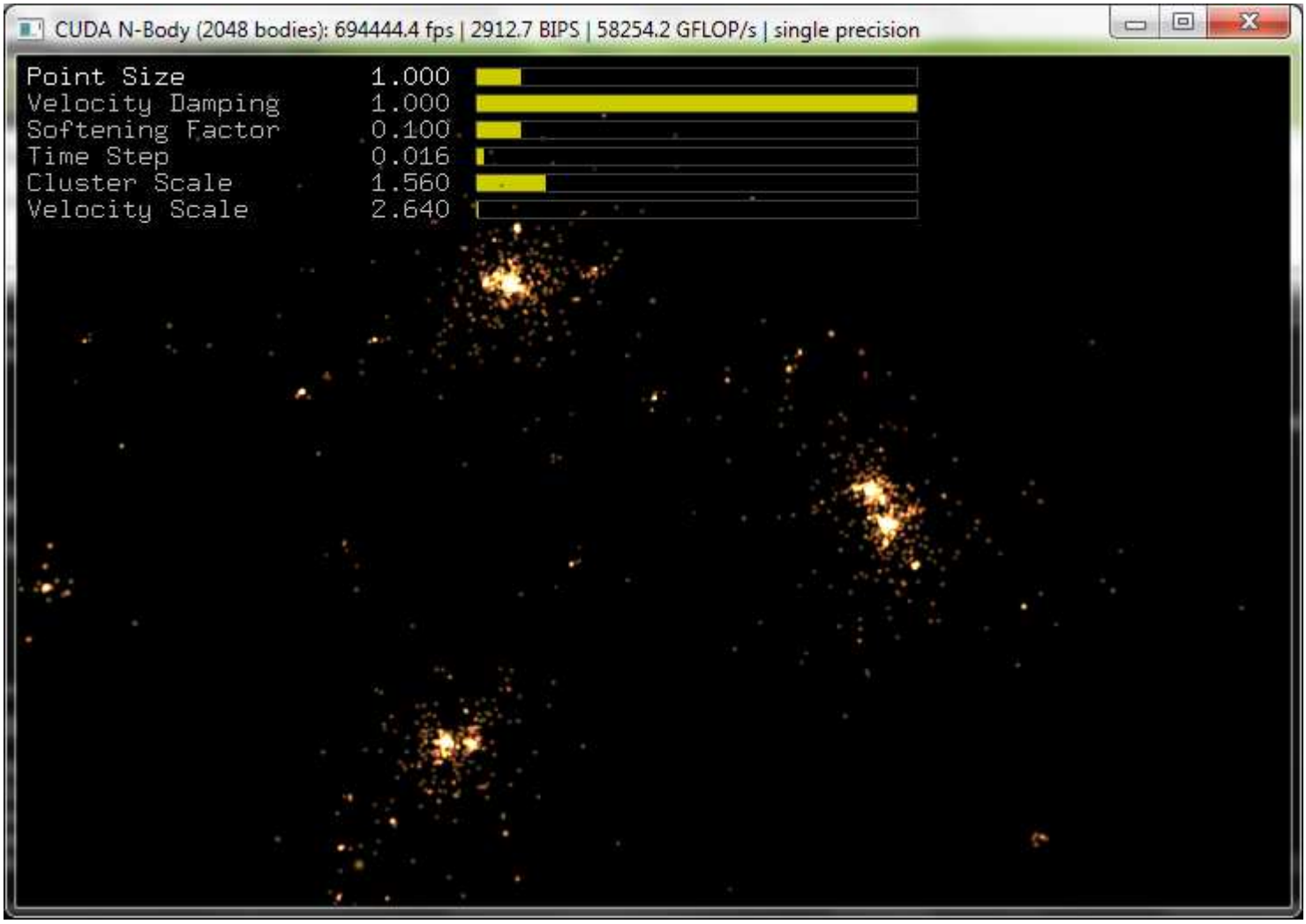
- Problème des n-corps
  - Comment évolue un système de n corps sous l'effet des forces de gravité qu'ils appliquent les uns sur les autres?
  - Pour n masses, 3*n équations différentielles du 2ème ordre
    -> 6*n inconnues



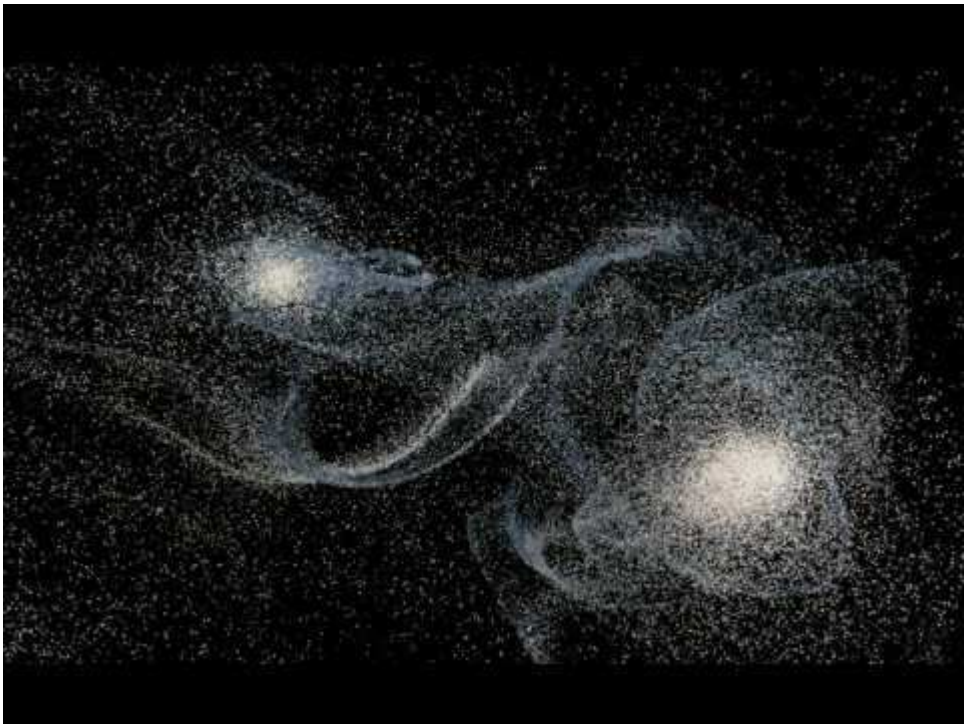

CUDA N-Body (2048 bodies): 664893.6 fps | 2788.8 BIPS | 55775.3 GFLOP/s | single precision
Point Size 1.000
Velocity Damping 1.000
Softening Factor 0.100
Time Step 0.016
Cluster Scale 1.560
Velocity Scale 2.640

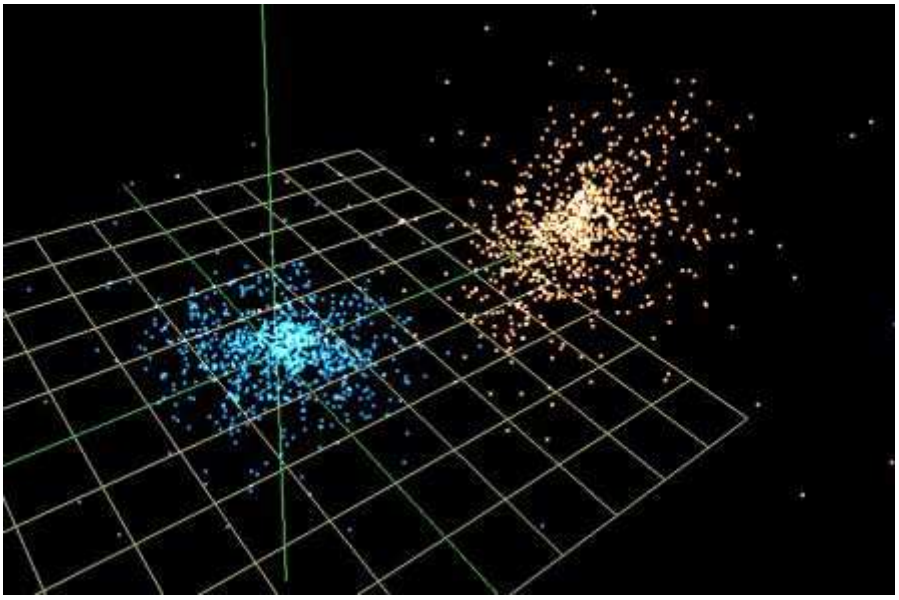

CUDA N-Body (2048 bodies): 651041.7 fps | 2730.7 BIPS | 54613.3 GFLOP/s | single precision
Point Size 1.000
Velocity Damping 1.000
Softening Factor 0.100
Time Step 0.016
Cluster Scale 1.560
Velocity Scale 2.640


CUDA N-Body (2048 bodies): 679347.8 fps | 2849.4 BIPS | 56987.8 GFLOP/s | single precision
Point Size 1.000
Velocity Damping 1.000
Softening Factor 0.100
Time Step 0.016
Cluster Scale 1.560
Velocity Scale 2.640



Résolution du problème des n-corps
Kit de développement CUDA (NVidia)



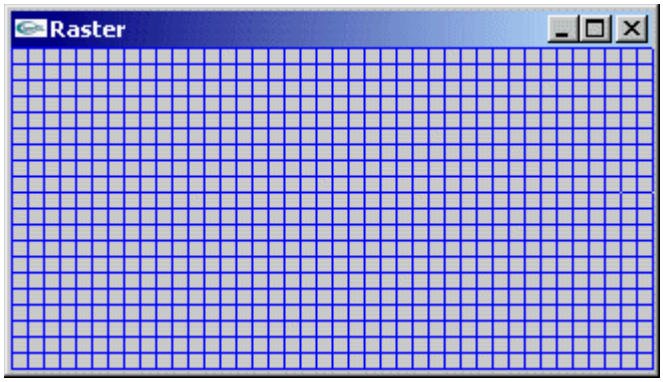


Simulations de collisions de galaxies

Matrices pour l'Informatique

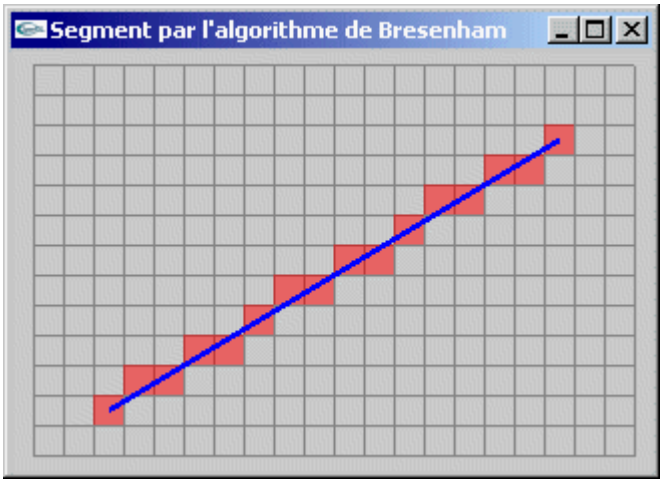
- Utilisation de matrices pour gérer les problèmes de l'informatique fondamentale qui se présentent naturellement sous forme 2D

Exemples

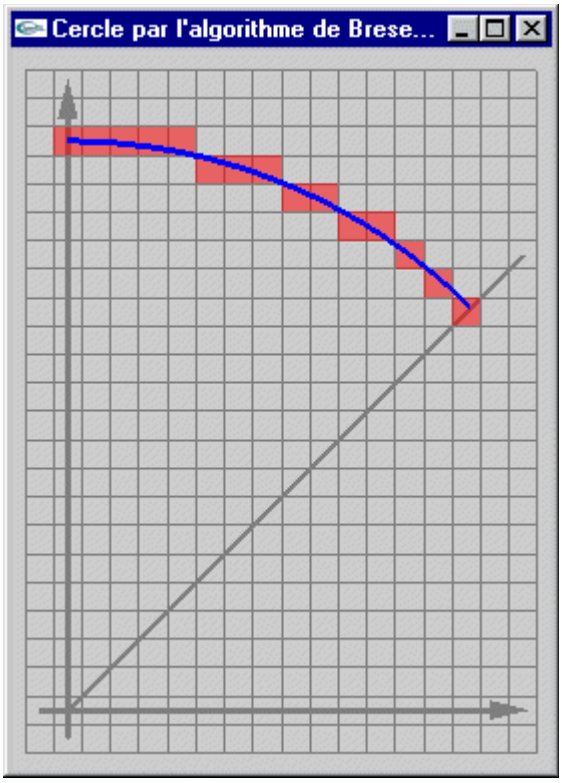
- Les écrans d'ordinateur
  - Technologie "raster" 

  - Une grille rectangulaire de pixels carrés pour lesquels on peut choisir individuellement la couleur
  - "Résolution": Nombres de colonnes et de lignes de pixels: 1024x768 -> 768 lignes de 1024 pixels
  - Autre caractéristique de la résolution: La "profondeur écran"
    - Nombre de bits de données affectés au stockage de la couleur d'un pixel
    - En 24 bits (3 octets), répartition en trois "teintes de base" stockées individuellement sur 8 bits
      -> 8 bits (1 octet) de composante rouge, 8 bits (1 octet) de composante verte et 8 bits (1 octet) de composante bleue
      -> Définition de chaque composante au moyen d'une valeur entière positive codée sur un octet (dans l'intervalle [0,255])

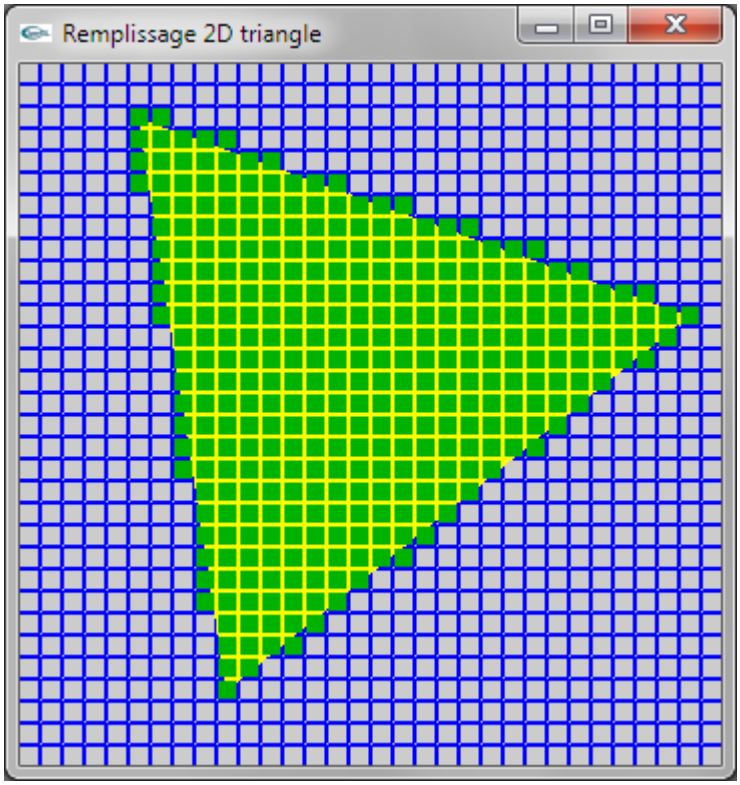
- Couleurs classiques:
  - le noir (0,0,0)
  - le blanc (255,255,255)
  - le rouge saturé (255,0,0)
  - le vert saturé (0,255,0)
  - le bleu saturé (0,0,255)
  - le jaune saturé (255,255,0)
  - le cyan saturé (0,255,255)
  - le magenta saturé (255,0,255)
- Autres couleurs: Choix les valeurs "RVB" adéquates
- Image affichée à l'écran: Une matrice de pixels codés en couleurs RVB stockée en mémoire
- Opération d'affichage de plus bas-niveau: "Allumer" un pixel avec une couleur particulière
- Autres opérations implantées en utilisant cette opération bas-niveau
- Paramètres nécessaires à l'appel de fonction d'allumage de pixel:
  - Position en x: un entier compris dans l'intervalle [0,rx-1] où rx est la résolution d'affichage en x
  - Position en y: un entier compris dans l'intervalle [0,ry-1] où ry est la résolution d'affichage en y
  - Couleur d'affichage
- "Librairie" de fonctions d'affichage graphique: Un grand nombre de fonctions de gestion de l'affichage en mode graphique et donc d'écriture dans la matrice de pixels:
  - Une ou plusieurs fonctions pour configurer la résolution d'affichage
  - Une ou plusieurs fonctions pour afficher un pixel selon une couleur
  - D'autres fonctions d'affichage plus élaborées:
    - Vidage la zone d'affichage avec une couleur (couleur de fond)
    - Traçage de segments de droite
    - Remplissage de polygones
    - Elimination des parties cachées
    - ...



Dessin d'un segment de droite



Dessin d'un arc de cercle



Remplissage d'une facette triangulaire

Exemple d'affichage graphique bitmap

- Deux techniques de stockage d'images dans des fichiers:
  - Vectoriel: Images stockées sous la forme d'objets graphiques de type segment de droite, rectangle, rectangle rempli, cercle, disque, ...
  - Bitmap: Images stockées sous la forme d'une grille rectangulaire de pixels décrits individuellement par une couleur

  - Déclinaison de ces techniques de diverses manières avec pour conséquence l'apparition de divers formats de fichiers:
    - Exemples de formats vectoriels: Postscript (.ps) et SVG (.svg, Scalable Vector Graphic)
    - Exemples de formats bitmap: JPEG, BMP, GIF, PNG
      - Compression ou non
      - Si compression, perte de qualité ou non
      - Format 8 bits, 24 bits ou 32 bits
      - ...

- Traitement particulier applicable aux images bitmaps: Réalisation d'opérations de traitement d'image visant à en changer les caractéristiques:
  - Afficher en fausses couleurs
  - Augmenter/diminuer la luminosité
  - Augmenter/diminuer le contraste

- Flouter
- Extraire les contours
- Négativer
- ...

- Traitement mathématique de chaque pixel de la matrice de pixels représentant l'image
  - Calcul de la nouvelle valeur d'un pixel à partir de son ancienne valeur -> Filtrage spectral
  - Calcul de la nouvelle valeur d'un pixel à partir de son ancienne valeur ainsi que des valeurs d'autres pixels (généralement les pixels voisins) -> Filtrage matriciel

Image originale

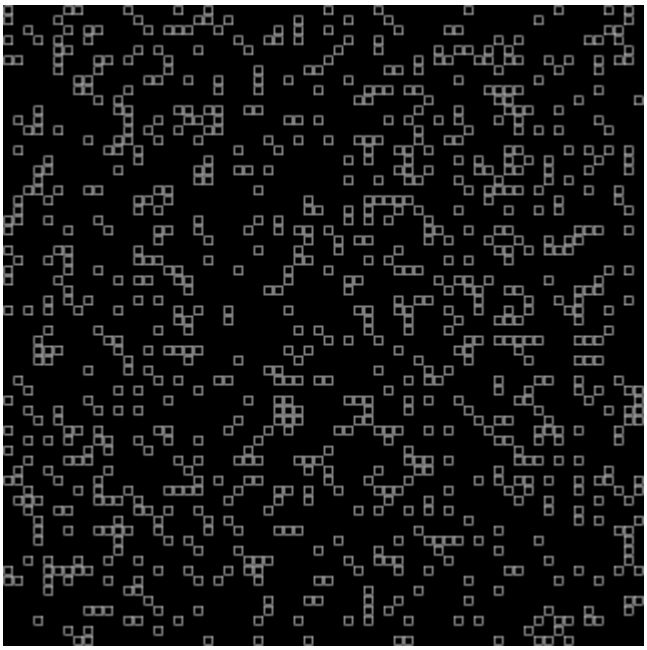
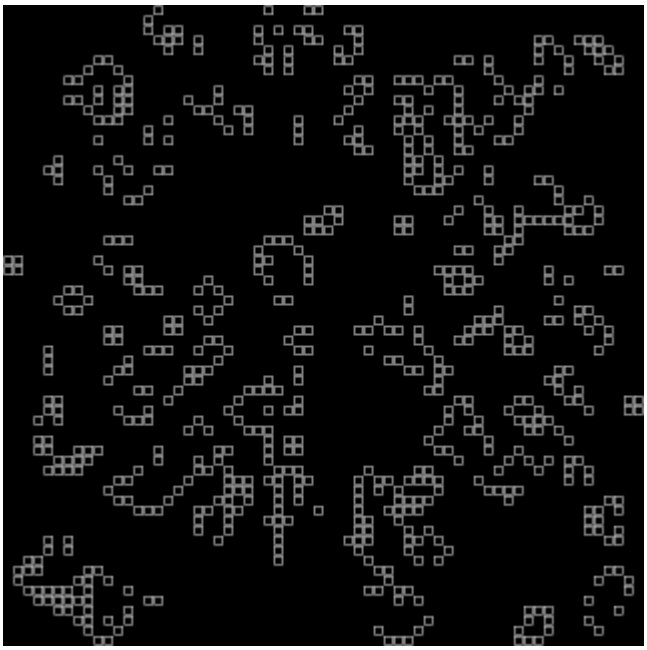
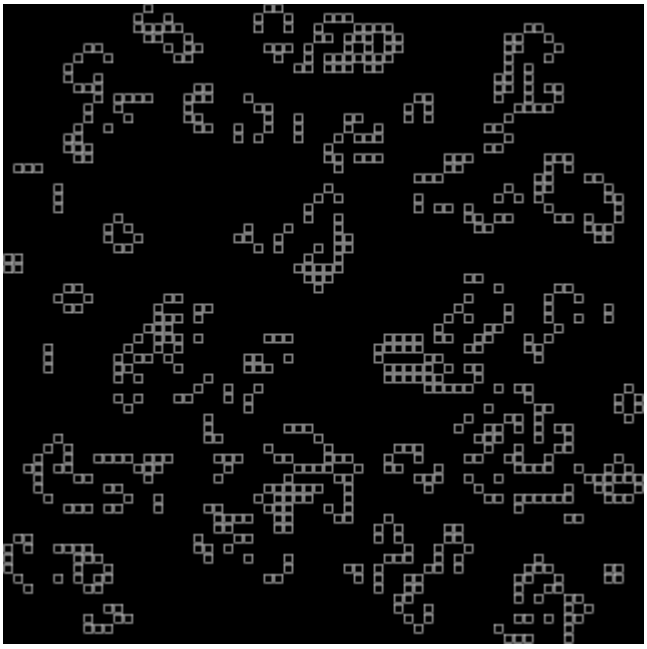
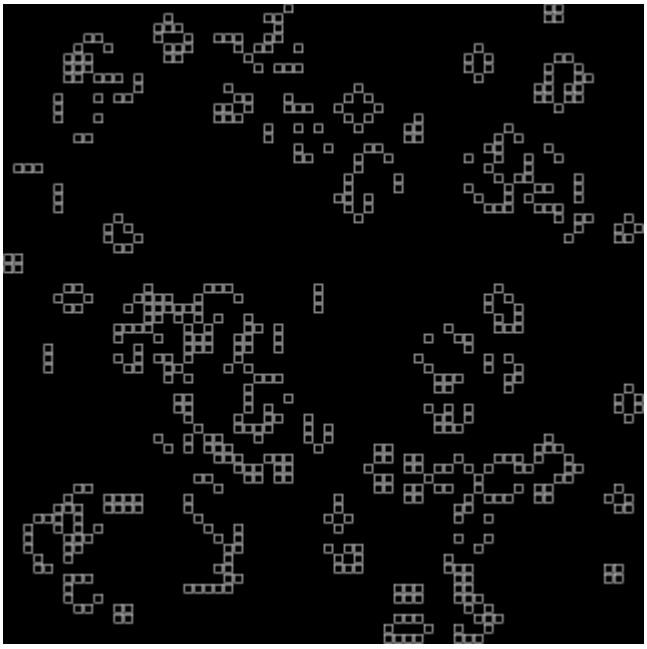
Image floutée
Remplacement de chaque pixel par la moyenne de ce pixel et des pixels voisins

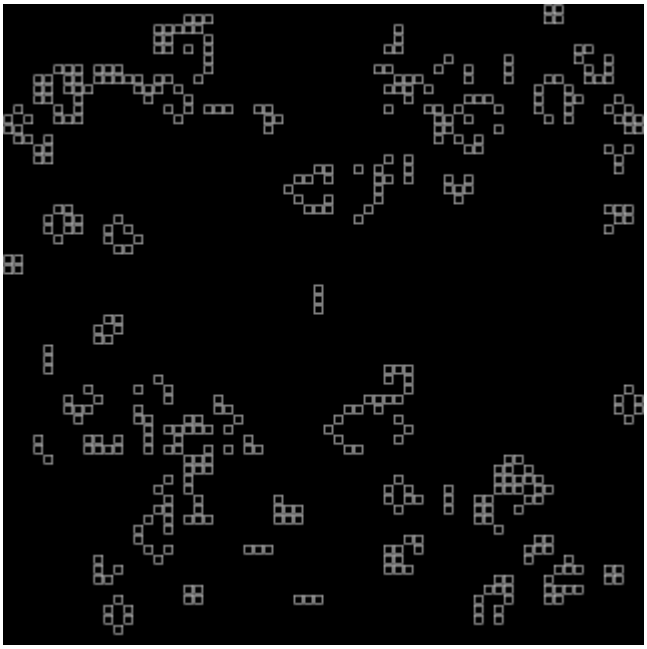
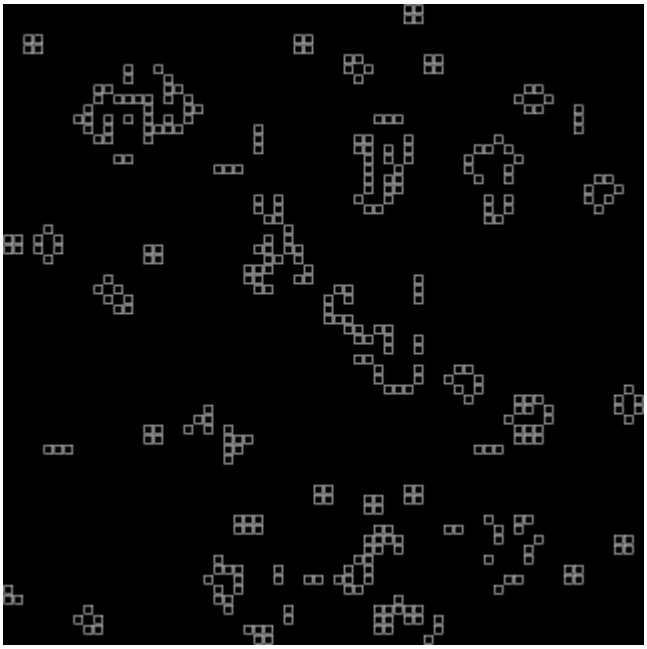
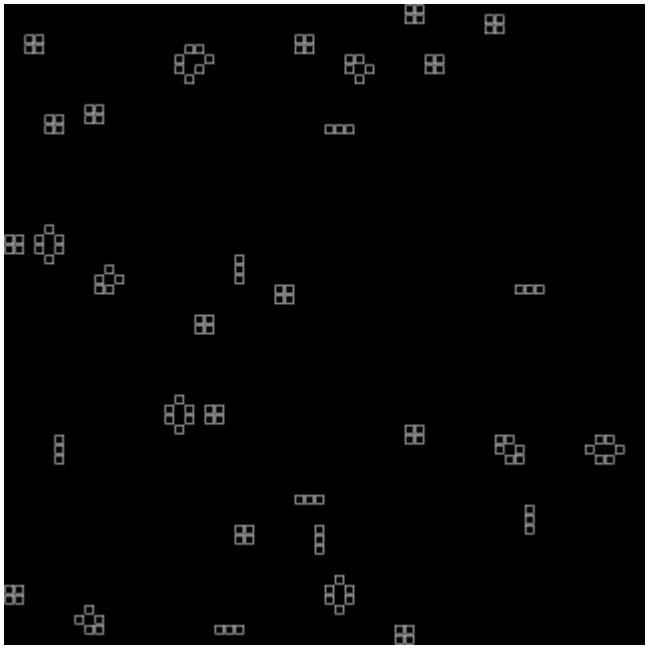
Image après extraction de contours
Gradiant calculé sur les pixels voisins

Exemple de traitement d'image

- Jeu de la vie
  - Une grille de booléens initialisés à vrai ou à faux
  - Chaque cellule évolue en fonction du contenu de ses 8 voisines
    - Si 2 voisins à vrai, la cellule ne change pas
    - Si 3 voisins à vrai, la cellule passe à vrai
    - Dans tous les autres cas, la cellule passe à faux

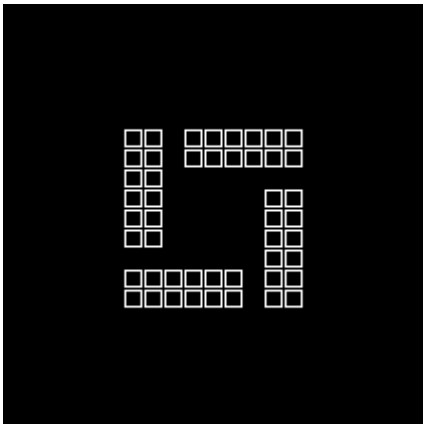
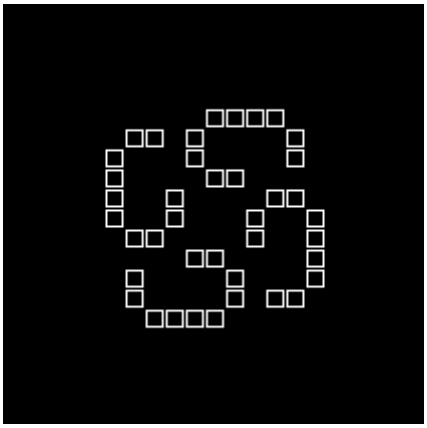
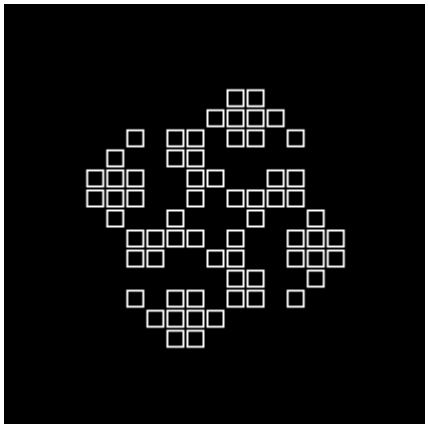
- Automate cellulaire
- Evolution d'un eco-système

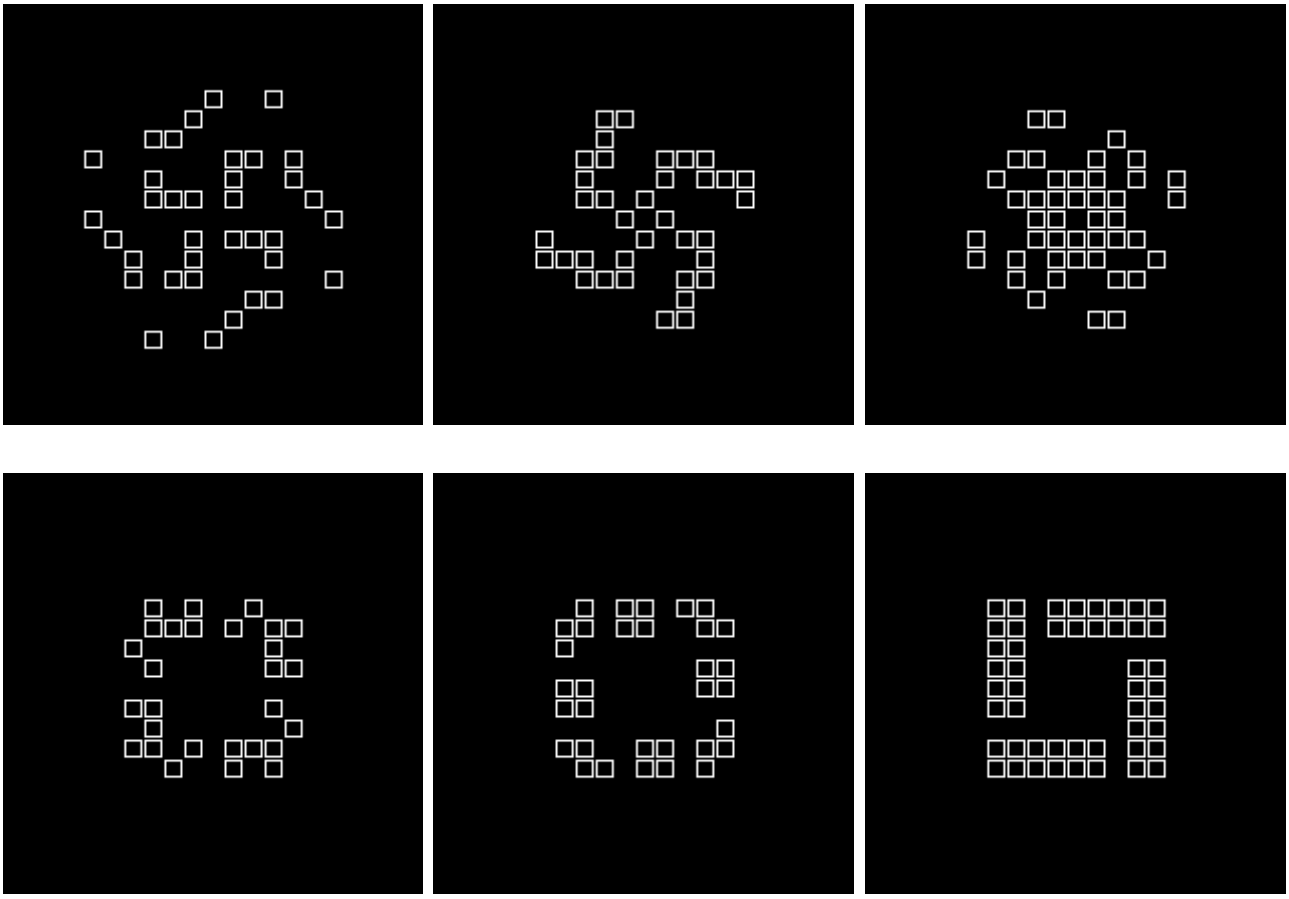





Placement aléatoire -> Stabilisation au bout d'un certain nombre d'étapes

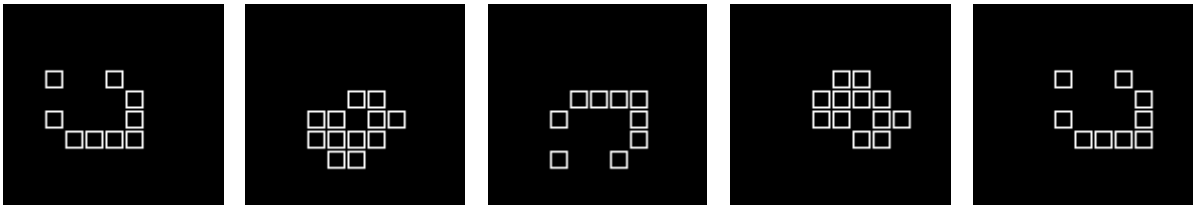
Exemple d'exécution



Evolution cyclique en place: Galaxie

Exemple d'exécution



Evolution cyclique avec déplacement régulier: Vaisseau

Exemple d'exécution

- Jeux 2D

Tétris

Candy Crush

Auteur: Nicolas JANEY
UFR Sciences et Techniques
Université de Besançon
16 Route de Gray, 25030 Besançon
nicolas.janey@univ-fcomte.fr