Rapport de projet

Audric Schiltknecht

Mars 2006

Résumé

Simulation d'un système de particules en `C`

Table des matières

1 Présentation générale

1.1 Introduction

L'objectif de ce projet est la réalisation d'une simulation physique d'un système de particule. Ce système pourra être soumis à plusieurs contraintes modélisant des forces physiques (dont le poids, les forces de frottement, etc).

1.2 Spécifications

Le programme devra simuler un système physique soumis à plusieurs contraintes.

1.2.1 Les contraintes physiques

Le système subira les forces suivantes :

Poids Chaque particule sera dotée d'une masse, et le système possèdera un vecteur gravité

Force de frottement Les particules subiront l'effet de la force de frottements fluides

1.2.2 Contraintes supplémentaires

En plus, on pourra paramétrer le système de telle sorte que les contraintes suivantes s'appliquent :

Ancre La particule est fixée à une position déterminée

Tige La distance entre deux particules est fixée

Chaine La distance entre deux particules est inférieure à une borne donnée

Espacement La distance entre deux particules est supérieure à une borne donnée

Ressort Les deux particules sont liées par un ressort

1.2.3 Obstacles

On pourra aussi rajouter des obstacles au système. Les obstacles seront alors représentés par des polygones.

Les contraintes qui devront être traitées dans le projet sont : le poids, la force de frottement, la contrainte ancre et la contrainte tige. Les autres sont optionnelles.

D'autre part, on devra utiliser le module « liste polymorphe »fourni et étudié en TP.

1.3 Format de fichier

D'autre part, le système sera décrit à l'aide d'un fichier au format défini.

Remarque : Toute ligne commençant par le caractère '#' sera ignorée

- `g1 g2` : Le vecteur gravité (double double)
- `k` : La viscosité du milieu (double)
- `nombre_particules` : Le nombre de particules du système (int)
- `x1 y1 m1` : Coordonnées de la particule et sa masse (double double double)
- ...
- `nombre_ancres` : Le nombre d'ancres du système (int)
- `x1 y1 np1` : Les coordonnées de l'ancre, ainsi que le numéro de la particule à laquelle elle s'applique (double double int)
- ...
- `nombre_tiges` : Le nombre de tiges du système
- `longueur1 np1 np2` : La longueur de la tige, ainsi que le numéro des particules liées par la tige (double int int)
- ...
- `nombre_chaines` : Le nombre de chaines
- `longueur1 np1 np2` : La longueur de la chaine, ainsi que le numéro des particules liées (double int int)
- ...
- `nombre_espaces` : Le nombre d'espaces
- `longueur1 np1 np2` : La longueur de l'espace, ainsi que le numéro des particules liées (double int int)
- ...
- `nombre_ressorts` : Le nombre de ressorts
- `longueur1 coef1 np1 np2` : La longueur, le coefficient de raideur du ressort ainsi que le numéro des particules liées (double double int int)
- ...

2 Structure de données

On présentera ici l'ensemble des structures de données, partant des structures simples pour obtenir l'ensemble du système.

2.1 Le point - Le vecteur

C'est la structure la plus élémentaire, simplement composée de deux champs permettant de représenter en coordonnées cartésiennes un point dans un espace à deux dimensions :

```
1 typedef struct {
    double x;
    double y;
  } Point;
```

Listing 1 – Structure Point

La structure `vecteur` est identique, seul le nom des champs est modifié : `x` devient `norme_x` et `y` `norme_y`.

2.2 La particule

Structure de base dans ce projet, j'ai adopté le type suivant :

```
typedef struct {
       int id;    /* Numéro de création de la particule */
       Point* position;  /* La position de la particule en T */
       Point* position_p;  /* La position de la particule en T-1 */
       double masse;   /* Masse de la particule */
       T_LIST forces;  /* Les forces appliquées sur la particule */
       } Particule;
```

Listing 2 – Structure Particule

La liste de forces est une liste « polymorphique »qui servira à stocker les forces créées par les ressorts. On justifiera la nécessité de stocker la position précédente par la suite (voir page 7, équation 1)

2.3 Les contraintes

Il nous est possible d'appliquer sur les particules des contraintes. Ces contraintes seront modélisées de la façon suivante : un (ou plusieurs) champ(s) de la structure contiendra un pointeur sur la particule concernée, et les autres champs seront spécifique à la contrainte en question.

2.3.1 L'ancre

Ainsi, une ancre est simplement décrite de la façon suivante :

```
typedef struct {
       Particule* particule;  /* La particule à ancrer */
       Point* position;  /* Sa position */
       } Ancre;
```

Listing 3 – Contrainte ancre

La modification de la position d'une ancre se fera en modifiant les coordonnées de son champ `position`, la particule sera alors modifiée lors de la résolution des contraintes.

2.3.2 La tige

Une tige fixe une longueur entre deux particules :

```
typedef struct {
       Particule* e1;   /* La première extrémité */
       Particule* e2;   /* La seconde extrémité */
```

```
        double longueur; /* La longueur de la tige */
5       } Tige;
```

Listing 4 – Contrainte tige

2.3.3 La chaine - l'espacement

Les types chaines et espacement sont en fait les mêmes que le type tige.

2.3.4 Ressort

Le type ressort contient en plus le coefficient de raideur du ressort :

```
typedef struct {
        Particule* e1;   /* Première extrémité du ressort */
        Particule* e2;   /* Seconde extrémité */
        double longueur;  /* Longueur au repos du ressort */
5       double coefficient;  /* Coefficient de raideur du ressort */
        } Ressort;
```

Listing 5 – Contrainte tige

2.4 Le système

Le système est la représentation numérique du système physique. Cette structure contiendra l'ensemble des données du système : particules, contraintes, etc.

J'ai donc choisi la structure suivante :

```
typedef struct {

3       /* Les paramètres physiques du systeme */
        Vecteur* gravite; /* Le vecteur gravité */
        double coef; /* Coefficient de viscosité */

        /* La liste des éléments du système */
8       T_LIST particules; /* Liste de particules */

        T_LIST tiges; /* Liste de tiges */

        T_LIST ancres; /* Liste d'ancres */
13
        T_LIST chaines; /* Liste de chaines */

        T_LIST espaces; /* Liste d'espaces */

18      T_LIST ressorts; /* Liste de ressorts */
```

```
/* Les données nécessaires pour l'interaction avec la souris */
      Ancre* deplace; /* Ancre qui sera déplacée par la souris */

      int destructible; /* Indique si l'ancre précédente peut être
          détruite */

      } Systeme;
```

Listing 6 – Structure système

où `T_LIST` est le type « liste polymorphique »du module liste fourni.

3 Algorithme

3.1 Calcul des forces

3.1.1 Équation de Verlet

Pour faire évoluer la particule au fil du temps, on utilisera l'équation suivante, dite « équation de Verlet » :

$$\vec{X}_{t+1} = 2 * \vec{X}_t - \vec{X}_{t-1} + \vec{a}\Delta t^2 \tag{1}$$

avec Δt intervalle de temps entre deux mesures.

Cette équation nécessite donc de connaître l'accélération. On peut calculer ce vecteur grâce à la formule suivante :

$$\sum \vec{f_i} = m\vec{a} \tag{2}$$

où m est la masse de la particule.

3.1.2 Force de frottement

Il faudra aussi prendre en compte les forces de frottements, données par la formule suivante :

$$\vec{f} = -k\|\vec{v}\|\vec{v} \tag{3}$$

Cette formule suppose de connaître la vitesse, que l'on obtiendra par l'équation :

$$\vec{v} = \frac{\Delta\vec{X}}{\Delta t} = \frac{\vec{X}_t - \vec{X}_{t-1}}{\Delta t} \tag{4}$$

3.2 Algorithme de calcul

À chaque pas de temps, on effectuera les calculs suivants :
- On calcule la vitesse de la particule
- On calcule les forces de frottements

– On somme les forces pour calculer l'accélération
– On intègre le mouvement par l'équation de Verlet (équation 1)
– Résolution des contraintes

3.2.1 Calcul de la vitesse

La vitesse n'est pas difficile à calculer. Le code suivant effectue ce calcul :

```
Vecteur* calculer_vitesse (Particule* particule, double dt){

  /* On utilise la formule DV = DP/DT */
  double norme_x = particule -> position -> x - particule ->
      position_p -> x;
  double norme_y = particule -> position -> y - particule ->
      position_p -> y;


  /* Remarque: On utilise MULT car sinon la vitesse tend vers l'
      infini */
  norme_x /= dt*MULT;
  norme_y /= dt*MULT;

  return creer_vecteur(norme_x, norme_y);
}
```

Listing 7 – Calcul de la Vitesse

Remarque On utilise une macro `MULT`, car lors des tests, j'ai constaté que la vitesse tendait très rapidement vers $+\infty$. En divisant chacune des composantes de la vitesse par cette constante, on réduit cet effet indésirable.

3.2.2 Calcul des forces

Le calcul de la vitesse ayant été effectué, il ne reste plus qu'à calculer les forces de frottement.

Il ne reste plus qu'à sommer toutes ces forces pour obtenir l'accélération. C'est le rôle de la fonction `calculer_acceleration`

```
Vecteur* calculer_acceleration (Particule* particule, double dt,
    double gx, double gy, double k) {

  double nx = 0; /* Norme en X de la somme des forces */
  double ny = 0; /* Norme en Y de la somme des forces */

  Vecteur* vitesse = calculer_vitesse(particule,dt);

  int ret; /* Valeur de retour des fonctions du module liste */
```

```
  /* T_LIST qui contiendra la liste des forces */
  T_LIST forces = particule -> forces;

  /* Vecteur contenant la tête de la liste des forces */
  Vecteur* vect;

  /* On calcule la somme des forces dues aux ressorts */
  while(!is_empty(forces)){
    /* On récupère la tete de la liste */
    vect = (Vecteur*)head(forces,&ret);

    nx += vect -> norme_x;
    ny += vect -> norme_y;

    forces = tail(forces, &ret);
  }

  /* Calcul de la force de friction */
  nx += -k*norme(vitesse)*(vitesse->norme_x);
  ny += -k*norme(vitesse)*(vitesse->norme_y);

  nx /= particule->masse;
  ny /= particule->masse;

  /* On rajoute la gravité */
  nx += gx;
  ny += gy;

  return creer_vecteur(nx,ny);
}
```

Listing 8 – Calcul de l'accélération

3.2.3 Intégration de Verlet

Cette procédure va intégrer le mouvement, et mettre à jour la position de la particule en utilisant l'équation de Verlet (1).

```
void integrer(Particule* particule, double dt, Vecteur* g, double k)
    {

  double gx = g -> norme_x;
  double gy = g -> norme_y;

  /* On calcule la somme des forces appliquées à cette particule */
  Vecteur* a = calculer_acceleration(particule,dt,gx,gy,k);

  /* On récupère la position courante */
  Point* tmp = particule -> position;
```

```
11
    /* Équation de Verlet */
    double n_x = 2*(tmp -> x) - particule -> position_p -> x + dt*dt*(
        a -> norme_x);
    double n_y = 2*(tmp -> y) - particule -> position_p -> y + dt*dt*(
        a -> norme_y);

16  particule -> position = creer_point(n_x,n_y);

    /* On libère la mémoire de la position précédente */
    free(particule -> position_p);

21  /* On met à jour la position précédente */
    particule -> position_p = tmp;

    /* On détruit le vecteur accélération*/
    free(a); a = NULL;
26
}
```

Listing 9 – Calcul de la nouvelle position

Remarque En théorie, il faudrait appeller cette procédure avec un dt valant $50ms$ (on verra par la suite pourquoi). Cependant, lors des tests, il s'avère qu'une telle valeur rend l'affichage assez lent. J'ai donc délibérément augmenté cette valeur, et ai pris $0.5sec$.

3.2.4 Résolution de contraintes

Le principe de la résolution des contraintes est le suivant :
- On résout chaque type de contrainte « type par type »
- On itère l'étape précédente un nombre fixé de fois

Voyons en premier lieu comment résoudre les contraintes selon les types.

Ancre La contrainte de type « ancre »est la plus simple à résoudre, puisqu'il suffit simplement de déplacer la particule associée à l'ancre.

La procédure suivante effectue cette opération :

```
void resoudre_contrainte_ancre(Ancre* ancre) {
       Particule* particule = ancre -> particule;
3      Point* position = ancre -> position;

       /* Pour une ancre: On replace la particule à sa position
           originelle */
       particule -> position -> x = position -> x;
       particule -> position -> y = position -> y;
8
```

```
}
```

Listing 10 – Résolution contrainte type Ancre

Tige La contrainte tige est plus difficile à résoudre. En effet, les deux particules peuvent se trouver à des distances quelconques, et il faut les rapprocher à une distance fixée.

Pour résoudre cette contrainte, je me suis basé sur l'algorithme donné sur la page web suivante : `http://www.teknikus.dk/tj/gdc2001.htm`. Le voici :

```
1  void resoudre_contrainte_tige(Tige* tige) {
     Point* pt1 = tige -> e1 -> position;
     Point* pt2 = tige -> e2 -> position;

     /* On récupère l'inverse des masses des particules */
6    double invm1 = tige -> e1 -> masse;
     double invm2 = tige -> e2 -> masse;

     /* On calcule la distance entre les deux points */
     double delta_x = pt2 -> x - pt1 -> x;
11   double delta_y = pt2 -> y - pt1 -> y;

     /* delta est la norme du vecteur pt2 - pt1 */
     double delta = sqrt(pow(delta_x,2)+pow(delta_y,2));

16   /* La correction à appliquer aux points pour donner longueur fixée
         */
     double diff = (delta - tige -> longueur)/(delta*(invm1 + invm2));

     /* On corrige le premier point */
     pt1 -> x += invm1*delta_x*diff;
21   pt1 -> y += invm1*delta_y*diff;

     /* On corrige le second point */
     pt2 -> x -= invm2*delta_x*diff;
     pt2 -> y -= invm2*delta_y*diff;
26 }
```

Listing 11 – Résolution contrainte type Tige

Chaine - Espace Pour résoudre la contrainte chaine, il suffit d'appliquer l'algorithme de résolution de la contrainte de type tige lorsque la distance entre les deux points est supérieur à la longueur de la chaine.

Pour la contrainte de type espace, il suffit d'appliquer l'algorithme de résolution de contrainte de type tige lorsque la distance entre les deux points est inférieure à la longueur de l'espace.

Pour calculer la distance entre deux points, on utilisera la fonction suivante :

```
double distance (Point* p1, Point* p2) {
  double dx = p2 -> x - p1 -> x;
  double dy = p2 -> y - p1 -> y;

  return sqrt(pow(dx,2)+pow(dy,2));
}
```

Listing 12 – Distance entre deux points

3.3 Ressort

Pour résoudre cette contraintes, on va utiliser les équations fournies pour calculer les forces produites par le ressort sur chacun de ses deux extrémités :

$$\vec{f_{12}} = \frac{d(P_1, P_2) - l}{2} \times k \times \frac{\vec{P_1 P_2}}{d(P_1, P_2)} \tag{5}$$

$$\vec{f_{21}} = \frac{d(P_1, P_2) - l}{2} \times k \times \frac{\vec{P_2 P_1}}{d(P_1, P_2)} \tag{6}$$

Le code suivant effectue le calcul des forces exercées par le ressort, et rajoute ces forces dans la `T_LIST` de forces de chacune des particules :

```
void calculer_force_ressort(Ressort* ressort) {
  /* Principe: On va calculer les forces appliquées sur chacune des
   * extrémités du ressort, et les rajouter dans la liste de forces
       de
   * chacune des particules
   */

  /* Pour alleger les notations */
  Particule* p1 = ressort -> e1;
  Particule* p2 = ressort -> e2;

  /* Calcul de la distance entre les deux points */
  double dist = distance(p1 -> position, p2 -> position);

  /* Coefficient multiplicateur utilisée dans la norme des forces */
  double coef =
    (dist - ressort -> longueur)*(ressort -> coefficient)/(2*dist*
        MULT);

  /* Composante en X */
  double c_x = coef*(p2 -> position -> x - p1 -> position -> x);
  /* Composante en Y */
  double c_y = coef*(p2 -> position -> y - p1 -> position -> y);
```

```
  /* Calcul de f12 */
  Vecteur* f12 = creer_vecteur(c_x,c_y);
  /* Calcul de f21 */
  Vecteur* f21 = creer_vecteur(-c_x,-c_y);

  /* Ajout des forces aux particules */
  cons(f12, &(p1 -> forces));
  cons(f21, &(p2 -> forces));

}
```

Listing 13 – Résolution de la contrainte ressort

Remarque On utilise une macro `MULT`, pour les mêmes raisons que celles évoquées au listing 7 : dans certains cas, le coefficient de raideur du ressort est trop élevé, et crée des forces de composante trop élevée.

4 Description du système par un fichier

Nous avons déjà décrit la structure du fichier utilisé pour décrire un système. Voyons comment en effectuer la lecture.

4.1 Principe

L'algorithme 1 décrit en langage naturel le principe général de la lecture du fichier de description du système. Attachons nous aux fonctions auxiliaires. Comme elles sont similaires, nous ne les décrirons pas toutes.

4.1.1 Commentaires

Le fichier peut contenir des commentaires, qu'il faut pouvoir ignorer. Le code suivant déroule le fichier ouvert jusqu'à une ligne ne commençant pas par le symbole '#' :

```
void parse_commentaire(FILE* file, char* ligne){

  while((fgets(ligne, NB_CHAR_LG, file) != NULL)&&(ligne[0]=='#'));

}
```

Listing 14 – Procédure ignorant les commentaires

Cette procédure devra être appelée entre chaque fonction de lecture d'un élément du système.

Algorithm 1 Principe de lecture d'un fichier

```
1: Systeme Fonction CHARGER_SYSTEME ( Tableau de chractères fichier )
2:     Ouverture du fichier
3:     gx, gy ←Lecture de la gravité
4:     Lecture du coefficient de viscosité
5:     nb ←Lecture du nombre de particules
6:     Pour i ← 0 à nb Faire
7:         Lire particule numéro nb
8:     Fin Pour
9:     nb ←Lecture du nombre d'ancres
10:    Pour i ← 0 à nb Faire
11:        Lire ancre numéro nb
12:    Fin Pour
13:    nb ←Lecture du nombre de tiges
14:    Pour i ← 0 à nb Faire
15:        Lire tige numéro nb
16:    Fin Pour
17:    nb ←Lecture du nombre de chaines
18:    Pour i ← 0 à nb Faire
19:        Lire chaine numéro nb
20:    Fin Pour
21:    nb ←Lecture du nombre de espaces
22:    Pour i ← 0 à nb Faire
23:        Lire espace numéro nb
24:    Fin Pour
25:    nb ←Lecture du nombre de ressort
26:    Pour i ← 0 à nb Faire
27:        Lire particule ressort nb
28:    Fin Pour
       Retourne Système créée
29: Fin function
```

4.1.2 Lecture d'une particule

Il nous faut maintenant lire les particules une à une. C'est le code suivant qui effectue cette opération :

```
void lire_particule(Systeme* systeme, FILE* file, char* ligne, int
    id){
  double x;
  double y;
  double m;

  int retour; /* Contient le nombre d'élément lus */

  /* On saute les commentaires */
  parse_commentaire(file,ligne);

  /* Lecture d'une particule, au format
   *  x y masse
   */
  retour = sscanf(ligne,"%lf %lf %lf", &x, &y, &m);
  if(retour != 3) {
  /* Un problème dans le format de la particule */
    fprintf(stderr, "Erreur: Particule %d au format incorrect ! \n",
        id);
  } else if(m < EPS) {
    fprintf(stderr, "Erreur: Particule %d de masse nulle !\n",id);
  } else {
    printf("Particule n %d de position (%g,%g), de masse %g\n",id,x,y
        ,m);

    /* Ajout de la particule au système */
    ajouter_particule(systeme,x,y,m,id);
  }
}
```

Listing 15 – Procédure lisant une particule

Le principe de cette procédure est simple : on ignore les commentaires, puis on effectue la lecture à l'aide de la fonction `sscanf` des trois données décrivant une particule. On teste ensuite le nombre d'éléments lus, et en cas d'erreur, on affiche un message.

S'il n'y a pas d'erreur, on fait appel à la procédure `ajouter_particule` qui va créer la particule, et l'ajouter au système.

Voici le code de cette dernière procédure :

```
void ajouter_particule(Systeme* systeme, double x, double y, double
    m, int id) {
  /* Création de la particule */
  Particule *part = creer_particule(x,y,m,id);

```

```
  /* Ajout de la particule à la liste */
  cons(part,&(systeme -> particules));
}
```

Listing 16 – Procédure d'ajout d'une particule au système

4.2 Lecture d'une ancre

Suivant le même principe que précédemment, on va lire les paramètres d'une ancre. Le code suivant effectue cette opération :

```
void lire_ancre(Systeme* systeme, FILE* file, char* ligne){
  double x;
  double y;
  int id;

  int retour; /* Contiendra le nombre de données lues */

  /* On saute les commentaires */
  parse_commentaire(file,ligne);

  /* Lecture d'une ancre au format
   *  x y n
   */
  retour = sscanf(ligne,"%lf %lf %d", &x, &y, &id);

  if(retour != 3) {
    fprintf(stderr, "Erreur dans le format de l'ancre %d ! \n",id);
  } else {
    printf("Ancre de position (%g,%g), associée à la particule %d\n",
        x,y,id);

    /* Ajout de l'ancre au système */
    ajouter_ancre(systeme,x,y,id);
  }
}
```

Listing 17 – Lecture d'une ancre

Comme précédemment, voyons le code de la procédure `ajouter_ancre` :

```
void ajouter_ancre(Systeme* systeme, double x, double y, int id){
  /* Création du particule temporaire servant à la recherche */
  Particule* tmp = creer_particule(0,0,0,id);

  /* On récupère la particule d'identifiant id */
  Particule *part = member(tmp,systeme -> particules,
      egalite_particule_id);

  /* On vérifie que la particule existe */
```

```
  if(part == NULL) {
    fprintf(stderr, "Pas de particule n%d\n",id);
  } else {
    /* Création de l'ancre */
    Ancre* ancre = creer_ancre(part,x,y);

    /* Ajout de l'ancre au système */
    cons(ancre,&(systeme->ancres));
  }


  /* Suppression de la particule */
  detruire_particule(&tmp);
}
```

Listing 18 – Procédure d'ajout d'une ancre

Cette procédure est légèrement plus complexe, détaillons là.

4.2.1 Principe de l'ajout d'une ancre

- On va chercher à l'aide de la fonction `member`, la particule d'identifiant `id`
- On teste si cette particule existe.
  - Si oui, on crée une ancre, puis on l'ajoute à la liste des ancres du système
  - Si non, on affiche un message d'erreur

Le code de la fonction de création d'ancre est simple :

```
Ancre* creer_ancre(Particule* particule, double x, double y) {
  /* Allocation de mémoire */
  Ancre* ancre = (Ancre*)malloc(sizeof(Ancre));

  /* Affectation des valeurs */
  ancre -> particule = particule;
  ancre -> position = creer_point(x,y);

  return ancre;
}
```

Listing 19 – Création d'une ancre

4.2.2 Recherche d'une particule

Voyons l'utilisation de la fonction `member`.

Comme nous l'indique le cartouche de cette fonction, elle prend en paramètre une fonction de comparaison, qui dans notre cas, doit rechercher une particule à partir de son identifiant. Voici le code de cette fonction de recherche :

```c
int egalite_particule_id(Particule *p1, Particule* p2){
  return p1 -> id == p2 -> id;
}
```

Listing 20 – Égalité particule par rapport à l'identifiant

Pour utiliser la fonction `member`, il faut créer une particule temporaire qui servira de « masque »lors de la recherche : c'est la particule `tmp`.

5 Interaction

La principale « difficulté »de ce sujet réside au fait que le programme fournit une interface graphique, qu'il nous faudra maitriser pour obtenir un système qui puisse réagir aux interactions à la souris.

Faisons une présentation des codes utilisés pour réaliser ces interactions.

5.1 `visualiseur.c`

Ce fichier, qui contient la fonction `main` du programme, va se charger d'intialiser l'affichage graphique, et le système.

Les points importants dans ce fichier sont :

1. La création du système dans la fonction `main`, effectuée par un appel à la fonction `charger_systeme` ;
2. La fonction `callback`, où l'on effectue les divers calculs ;
3. La desctruction du système, par appel à la procédure `detruire_systeme`

5.1.1 Fonction de callback

Cette fonction sera appellée toutes les 50ms au cours de l'execution du programme. C'est dans cette fonction que l'on doit :

- Effacer l'écran
- Calcul de la nouvelle position des particules et résolution des contraintes par appel à la procédure `resoudre_contraintes`
- Dessiner les éléments à leur nouvelle place

Procédure `resoudre_contraintes` Cette procédure va effectuer l'intégration de Verlet, puis la résolution des différents types de contraintes.

Procédure `detruire_systeme` Cette procédure va effectuer la destruction du système, par application des fonctions de destruction de chacun des types définis à l'aide de l'itérateur des listes polymorphes `do_list`.

Pour exemple, voici le code de la fonction de destruction d'une tige, ainsi que celui utilisé pour l'appel dans l'itérateur :

```
void detruire_tige(Tige** tige) {
  /* Les particules seront supprimées dans une autre procédure */
  free(*tige); (*tige) = NULL;
}
```

Listing 21 – Procédure de destruction d'une tige

```
void detruire_tige_it(Tige* tige) {
  detruire_tige(&tige);
}
```

Listing 22 – Enveloppe pour l'appel à l'aide de l'itérateur

5.2 `interaction.c`

Ce fichier va contenir les procédures appelés lors des diverses interaction avec la souris, à savoir : appui sur le bouton, lors du relâchement du bouton, et lors du déplacement de la souris avec le bouton appuyé.

5.2.1 Appui sur le bouton

Lorsque l'utilisateur appuie sur le bouton, il faut rechercher si une particule se trouve à proximité du curseur de la souris.

Pour cela, on va utiliser la fonction `member`, avec une fonction de recherche qui vérifie que deux particules ont même position avec une certaine précision (il est quasiment impossible de cliquer du premier coup exactement sur la particule). Cette particule sera ensuite ancrée, et ajouté dans le champ spécial du système (`deplace`), et le flag `destructible` est mis à 1, indiquant que cette ancre pourra être détruite lorsque l'utilisateur relâchera le bouton de la souris.

Remarque On va en fait rechercher en premier lieu s'il existe une ancre à proximité du pointeur de la souris. Si une ancre existe, elle est stockée dans le champ `deplace` du système, et le flag `destructible` est mis à 0, indiquant qu'il ne faudra pas la détruire.

Cette recherche d'ancre permettra à l'utilisateur de déplacer une ancre qui existait déjà dans le système.

5.2.2 Relâchement du bouton

Très simplement, on supprime l'ancre `deplace` du système si le flag `destructible` est à 1.

5.2.3 Déplacement de la souris

Cette étape est la plus compliquée à réaliser. La procédure appelée nous permet de récupérer les coordonnées de la destination du déplacement.

Initialement, je m'étais contenté de mettre à jour avec les coordonnées fournies par l'appel à cette procédure la position de l'ancre, la particule étant déplacée lors de la résolution des contraintes.

Cette solution, satisfaisante dans un premier lieu, donne cependant un résultat décevant : en effet, les objets peuvent alors être déplacé sans respecter les contraintes qui s'appliquent lorsque l'utilisateur maintient le bouton enfoncé. Bien évidement, lorsque l'utilisateur relâche la souris, les contraintes sont appliquées.

J'ai alors utilisé une autre méthode. En premier lieu, j'ai enlevé du système l'ancre `deplace`. Puis j'ai effectué un appel à la procédure `satisfaire_contraintes`. Ainsi, la particule pointée par la souris n'étant plus ancrée à sa position, elle doit donc respecter les contraintes. Il ne reste plus qu'à replacer l'ancre `deplace`, et à mettre à jour les coordonnée de la position de l'ancre avec celle de la particule.

Remarque Ces deux modes de fonctionnement sont disponibles. Pour disposer du mode avec l'effet « d'étirement », il suffit que la macro `CONTRAINTE` ne soit pas définie. Pour le second mode, il faut la définir.

6 Tests

Pour tester le comportement du programme, j'ai utilisé divers fichiers de test disponible dans le dossier `Systeme`. Le programme peut prendre comme argument le chemin vers un fichier de test. Si aucun argument n'est fourni, c'est le fichier `corde.sp`, qui doit se trouver dans le même dossier que l'exécutable qui est utilisé.

Description des fichiers de tests

air.sp Ce fichier permet de tester le système avec un coefficient de viscosité égal à celui de l'air

chaine.sp Fichier de test du type chaine

corde_mult.sp Fichier de test faisant appel à tous les types

corde.sp Fichier de description fourni avec le sujet

eau.sp Fichier pour tester le système avec un coefficient de viscosité égal à celui de l'eau

espace.sp Fichier de test du type espace

ressort.sp Fichier de test du type ressort

tige.sp Fichier de test du type tige

roue.sp Fichier décrivant un carré (le nom est bien choisi !)

Résultat des tests

Voici donc les diverses remarques que j'ai pu relever lors des tests.

Tout d'abord, il faut noter que de faibles valeurs du coefficient de viscosité ne se distinguent pas. Ainsi, je n'ai vu aucune différence entre l'utilisation du fichier `air.sp` et `eau.sp`. Par contre, des valeurs supérieures à 1 modifient bien l'évolution du système. Il est à noté que pour des valeurs plus importantes (supérieures à 20) le système passe dans un état instable. Il faut donc augmenter le coefficient utilisé dans le calcul de la vitesse si l'on souhaite pouvoir continuer à utiliser le programme.

Le même problème d'état instable se pose suivant la valeur du coefficient de raideur du ressort. Un coefficient supérieur à 10 nécessiterait une augmentation du coefficient utilisé dans le calcul de la force. D'autre part, il faut remarquer que le fichier de test `ressort.sp` présente un système assez étrange : en effet, bien que disposant de forces de frottement, le mouvement des particules ne semble pas s'arrêter ! Encore une fois, il faut que la valeur du coefficient de frottement soit plus importante (de l'ordre de l'unité au moins) pour que le ressort s'arrête.

Ce comportement disparaît néanmoins lors de l'utilisation du ressort dans un système plus important (exemple du fichier `corde_mult.sp`

La dernière remarque concerne le fichier `roue.sp`. Avec les paramètres par défaut du programme, rien d'anormal n'est à constater. Cependant, il suffit d'augmenter le nombre d'itération nécessaire à la résolution des contraintes pour voir apparaître un comportement pour le moins étrange : plus le nombre d'itérations est important, plus le carré va pivoter dans le sens inverse des aiguilles d'une montre, jusqu'à arriver au point où il se mettra à tourner.

Ce comportement s'explique par l'étape de résolution des contraintes. Comme l'on applique itérativement diverses procédures de résolution aux divers éléments du système, certains sont déplacés avant d'autres, et donc cela entraîne de légers déplacements du système qui peuvent, selon les cas, amener à des mouvements de plus grande amplitude.

7 Conclusion

8 Listings

```c
/*
 * Projet Système de particules
 *
 * geometrie.h
 *
 *  Fonctions et types géométriques utiles.
 *
 *      --- A compléter si besoin ---
 *
 */

#ifndef __GEOMETRIE_H__
#define __GEOMETRIE_H__

struct _Coordonnees {
  double x;
  double y;
};

typedef struct _Coordonnees Point;

/** Structure de donnée vecteur */
typedef struct Vect {
        double norme_x; /* Norme en abscisse */
        double norme_y; /* Norme en ordonnée */
} Vecteur;

/** Fonction créant un point en coordonnées cartésiennes
 * Arguments:
 *      x: Abscisse du point
 *      y: Ordonnée du point
 * Retourne: Un pointeur sur le point créé
 */
Point* creer_point (double x, double y);

/** Fonction calculant la distance entre deux points
 * Arguments:
 *      *p1: Premier point
 *      *p2: Deuxieme point
 * Retourne: La distance entre p1 et p2
 */
double distance (Point* p1, Point* p2);

/** Fonction de création d'un vecteur
 * Arguments:
 *      nx: Norme en x
 *      ny: Norme en y
 * Retourne: Un pointeur sur un vecteur
 */
Vecteur* creer_vecteur (double nx, double ny);

/** Procédure détruisant un vecteur
 * Argument: vect: Le vecteur à détruire
*/
void detruire_vecteur(Vecteur** vecteur);

/** Procédure utilisée lors de l'appel à detruire_vecteur dans l'itérateur
 * Argument: vect: Vecteur à détruire
*/
void detruire_vecteur_it(Vecteur* vect);

/** Fonction retournant la norme d'un vecteur
```



```c
 * Argument: Un vecteur
 * Retourne: La norme du vecteur
 */
double norme(Vecteur* vect);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * geometrie.c
 *
 *  Fonctions et types géométriques utiles.
 *
 *
 */
#include <stdlib.h>
#include <math.h>
#include "geometrie.h"

/** Fonction de création d'un point
 * Arguments:
 *      double x :  L'abscisse du point
 *      double y :  L'ordonnée du point
 * Retourne: Un pointeur vers un Point
 */
Point* creer_point (double x, double y){
        /* On alloue la mémoire */
        Point* point = (Point*)malloc(sizeof(Point));

        /* Affectation des valeurs */
        point -> x = x;
        point -> y = y;

        return point;
}

/** Fonction calculant la distance entre deux points
 * Arguments:
 *      p1: Premier point
 *      p2: Deuxieme point
 * Retourne: La distance entre p1 et p2
 */
double distance (Point* p1, Point* p2) {
        double dx = p2 -> x - p1 -> x;
        double dy = p2 -> y - p1 -> y;

        return sqrt(pow(dx,2)+pow(dy,2));
}

/** Fonction de création d'un vecteur
 * Arguments:
 *      nx: Norme en x
 *      ny: Norme en y
 * Retourne: Un pointeur sur un vecteur
 */
Vecteur* creer_vecteur (double nx, double ny) {
        /* Allocation de mémoire */
        Vecteur* vect = (Vecteur*)malloc(sizeof(Vecteur));

        /* Affectation des valeurs */
        vect -> norme_x = nx;
        vect -> norme_y = ny;

        return vect;
}

/** Fonction retournant la norme d'un vecteur
 * Argument: Un vecteur
```



```c
 * Retourne: La norme du vecteur
 */
double norme(Vecteur* vect) {
        double x = vect -> norme_x;
        double y = vect -> norme_y;

        return sqrt(x*x+y*y);
}

/** Procédure détruisant un vecteur
 * Argument: vect: Le vecteur à détruire
*/
void detruire_vecteur(Vecteur** vecteur) {
        free(*vecteur); vecteur = NULL;
}

/** Procédure utilisée lors de l'appel à detruire_vecteur dans l'itérateur
 * Argument: vect: Vecteur à détruire
*/
void detruire_vecteur_it(Vecteur* vect) {
        detruire_vecteur(&vect);
}
```

```c
/*
 * Projet Système de particules
 *
 * particule.h
 *
 *  Fonctions et types particule.
 *
 *
 */

#ifndef _H_PARTICULE
#define _H_PARTICULE

/* Définition du type particule */
typedef struct {
        int id; /* Numéro de création de la particule */
        Point* position; /* La position de la particule en T */
        Point* position_p; /* La position de la particule en T-1 */
        double masse; /* Masse de la particule */
        T_LIST forces; /* Les forces appliquées sur la particule */
        } Particule;

/** Fonction de création d'une particule
 * Arguments:
 *      x Abscisse de la particule
 *      y Ordonnée de la particule
 *      m Masse de la particule
 * Retourne: Un pointeur sur une particule
 */
Particule* creer_particule(double x, double y, double m, double id);

/** Fonction d'égalité de particules au sens de l'identifiant
 * Teste si deux particules ont même identifiant
 * Arguments:
 *      p1: Première particule
 *      p2: Deuxième particule
 */
int egalite_particule_id(Particule *p1, Particule* p2);

/** Fonction d'égalité de particules au sens des coordonnées
 * Teste si deux particules ont même coordonnées, à +- delta
 * Arguments:
 *      p1: Première particule
 *      p2: Deuxieme particule
 */
int egalite_particule_pos(Particule* p1, Particule* p2);

/** Fonction de calcul de la vitesse de la particule
 * Arguments:
 *      particule: Particule dont on souhaite calculer la vitesse
 *      dt: intervalle de temps
 * Retourne: Le vecteur vitesse
 */
Vecteur* calculer_vitesse (Particule* particule, double dt);

/** Fonction calculant l'accelération
 * Argument:
 *      particule: Particule dont on souhaite sommer les forces
 *      dt: Intervalle de temps entre deux calculs
 *      gx: Norme X de la force de gravité
 *      gy: Norme Y de la force de gravité
 *      k: Coefficient de friction
```



```c
 * Retourne: Le vecteur accélération (somme des forces / masse)
 */
Vecteur* calculer_acceleration (Particule* particule, double dt, double gx, doub
le gy, double k);

/** Procédure intégrant l'équation du mouvement par l'équation de Verlet
 * Arguments:
 *      particule: Particule dont on souhaite intégrer le mouvement
 *      dt: Intervalle de temps
 *      g: Vecteur gravité
 *      k: Coefficient de friction
 */
void integrer(Particule* particule, double dt, Vecteur* g, double k);

/** Procédure ré-initialisant les forces appliquées à une particule
 * Argument: particule: La particule que l'on traite
*/
void raz_forces(Particule* part);

/** Procédure détruisant la particule
 * Argument: particule: Particule à détruire
 */
void detruire_particule(Particule** particule);

/** Procédure utilisée pour convertir l'appel à detruire_particule dans
 * l'itérateur
 * Argument: part: La particule à détruire
 */
void detruire_part_it(Particule* part);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * particule.c
 *
 *  Fonctions et types particule.
 *
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"

/* Précision de la recherche d'une particule par rapport à sa position */
#define DELTA 4

/* Coefficient utilisé lors du calcul de la vitesse */
#define MULT 50

/** Fonction de création d'une particule
 * Arguments:
 *      x Abscisse de la particule
 *      y Ordonnée de la particule
 *      m Masse de la particule
 * Retourne: Un pointeur sur une particule
 */
Particule* creer_particule(double x, double y, double m, double id) {
        /* Allocation de la mémoire */
        Particule* particule = (Particule*)malloc(sizeof(Particule));

        /* Affectation des valeurs */
        particule -> position = creer_point(x,y);

        /* La vitesse est donc nulle au premier calcul */
        particule -> position_p = creer_point(x,y);

        /* Aucune force n'est appliquée sur la particule */
        particule -> forces = nil();

        particule -> masse = m;

        particule -> id = id;

        return particule;
}

/** Fonction d'égalité de particules au sens de l'identifiant
 * Teste si deux particules ont même identifiant
 * Arguments:
 *      p1: Première particule
 *      p2: Deuxième particule
 */
int egalite_particule_id(Particule *p1, Particule* p2){
        return p1 -> id == p2 -> id;
}

/** Fonction d'égalité de particules au sens des coordonnées
 * Teste si deux particules ont même coordonnées, à +- DELTA
 * Arguments:
```



```c
 *      p1: Première particule
 *      p2: Deuxieme particule
 */
int egalite_particule_pos(Particule* p1, Particule* p2) {
        return (abs(p1 -> position -> x - p2 -> position -> x) <= DELTA) &&
                (abs(p1 -> position -> y - p2 -> position -> y) <= DELTA);
}

/** Fonction de calcul de la vitesse de la particule
 * Arguments:
 *      particule: Particule dont on souhaite calculer la vitesse
 *      dt: intervalle de temps
 */
Vecteur* calculer_vitesse (Particule* particule, double dt){

        /* On utilise la formule DV = DP/DT */
        double norme_x = particule -> position -> x - particule -> position_p ->
 x;
        double norme_y = particule -> position -> y - particule -> position_p ->
 y;


        /* Remarque: On utilise MULT car sinon la vitesse tend vers l'infini */
        norme_x /= dt*MULT;
        norme_y /= dt*MULT;

        return creer_vecteur(norme_x, norme_y);
}

/** Fonction calculant l'accélération
 * Argument:
 *      particule: Particule dont on souhaite sommer les forces
 *      dt: Intervalle de temps entre deux calculs
 *      gx: Norme X de la force de gravité
 *      gy: Norme Y de la force de gravité
 *      k: Coefficient de friction
 * Retourne: Le vecteur accélération (somme des forces / masse)
 */
Vecteur* calculer_acceleration (Particule* particule, double dt, double gx, doub
le gy, double k) {

        double nx = 0; /* Composante en X de la somme des forces */
        double ny = 0; /* Composante en Y de la somme des forces */

        Vecteur* vitesse = calculer_vitesse(particule,dt);

        int ret; /* Valeur de retour des fonctions du module liste */

        /* T_LIST qui contiendra la liste des forces */
        T_LIST forces = particule -> forces;

        /* Vecteur contenant la tête de la liste des forces */
        Vecteur* vect;

        /* On calcule la somme des forces dues aux ressorts */
        while(!is_empty(forces)){
                /* On récupère la tete de la liste */
                vect = (Vecteur*)head(forces,&ret);

                /* Ajout des valeurs des composantes */
                nx += vect -> norme_x;
                ny += vect -> norme_y;
```

```
                /* On passe à la suite de la liste */
                forces = tail(forces, &ret);
        }

        /* Calcul de la force de friction */
        nx += -k*norme(vitesse)*(vitesse->norme_x);
        ny += -k*norme(vitesse)*(vitesse->norme_y);

        /* On divise par la masse pour obtenir une accélération */
        nx /= particule->masse;
        ny /= particule->masse;

        /* On rajoute la gravité */
        nx += gx;
        ny += gy;

        return creer_vecteur(nx,ny);
}


/** Procédure intégrant l'équation du mouvement par l'équation de Verlet
 * Arguments:
 *      particule: Particule dont on souhaite intégrer le mouvement
 *      dt: Intervalle de temps
 *      g: Vecteur gravité gravité
 *      k: Coefficient de friction
 */
void integrer(Particule* particule, double dt, Vecteur* g, double k){

        double gx = g -> norme_x;
        double gy = g -> norme_y;

        /* On calcule la somme des forces appliquées à cette particule */
        Vecteur* a = calculer_acceleration(particule,dt,gx,gy,k);

        /* On récupère la position courante */
        Point* tmp = particule -> position;

        /* Équation de Verlet */
        double n_x = 2*(tmp -> x) - particule -> position_p -> x + dt*dt*(a -> n
orme_x);
        double n_y = 2*(tmp -> y) - particule -> position_p -> y + dt*dt*(a -> n
orme_y);

        /* On crée le point associé à la nouvelle position */
        particule -> position = creer_point(n_x,n_y);

        /* On libère la mémoire de la position précédente */
        free(particule -> position_p);

        /* On met à jour la position précédente */
        particule -> position_p = tmp;

        /* On détruit le vecteur accélération*/
        free(a); a = NULL;

}

/** Procédure ré-initialisant les forces appliquées à une particule
 * Argument: particule: La particule que l'on traite
*/
```



```
void raz_forces(Particule* part) {
        /* On applique l'itérateur pour détruire les vecteurs forces stockés
         * dans la liste des forces
         */
        do_list(detruire_vecteur_it,part -> forces);

        /* Puis, on réinitialise la liste */
        part -> forces = nil();
}

/** Procédure détruisant la particule
 * Argument: particule: Particule à détruire
 */
void detruire_particule(Particule** particule) {
        /* On détruit chacun de ses champs */
        free((*particule) -> position); (*particule) -> position = NULL;
        free((*particule) -> position_p); (*particule) -> position_p = NULL;
        (*particule) = NULL;
}

/** Procédure utilisée pour convertir l'appel à detruire_particule dans
 * l'itérateur
 * Argument: part: La particule à détruire
 */
void detruire_part_it(Particule* part) {
        detruire_particule(&part);
}
```

```c
/**
 * Projet Système de particules
 *
 * ancre.h
 *
 *  Fonctions et type ancre.
 *
 *
 */

#ifndef _H_ANCRE
#define _H_ANCRE

/* Définition du type particule */
typedef struct {
        Particule* particule; /* La particule à ancrer */
        Point* position; /* Sa position */
        } Ancre;

/* Fonction de création d'une ancre
 * Arguments:
 *      particule: Pointeur sur la particule qui sera ancrée
 *      x: Abscisse de l'ancre
 *      y: Ordonnée de l'ancre
 * Retourne: Un pointeur sur une ancre
 */
Ancre* creer_ancre(Particule* particule, double x, double y);

/** Procédure de résolution de contrainte de type "ancre"
 * Argument: L'ancre dont on veut résoudre la contrainte
 */
void resoudre_contrainte_ancre(Ancre* ancre);

/** Procédure détruisant l'ancre
 * Argument: ancre: Ancre à détruire
 */
void detruire_ancre(Ancre** ancre);

/** Procédure utilisée pour l'appel à detruire_ancre dans l'itérateur
 * Argument: ancre: Ancre à détruire
 */
void detruire_ancre_it(Ancre* ancre);

/** Fonction d'égalité de deux ancres par leur position
 * Arguments:
 *      a1: Première ancre
 *      a2: Seconde ancre
 * Retourne: position a1 == position a2
 */
int egalite_ancre_pos(Ancre* a1, Ancre* a2);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * ancre.c
 *
 *  Fonctions et type ancre.
 *
 *
 */

#include <stdlib.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "ancre.h"

/** Précision de la recherche */
#define DELTA 4


/* Fonction de création d'une ancre
 * Arguments:
 *      particule: La particule qui sera ancrée
 *      x: Abscisse de l'ancre
 *      y: Ordonnée de l'ancre
 * Retourne: Un pointeur sur une ancre
 */
Ancre* creer_ancre(Particule* particule, double x, double y) {
        /* Allocation de mémoire */
        Ancre* ancre = (Ancre*)malloc(sizeof(Ancre));

        /* Affectation des valeurs */
        ancre -> particule = particule;
        ancre -> position = creer_point(x,y);

        return ancre;
}

/** Procédure de résolution de contrainte de type "ancre"
 * Argument: L'ancre dont on veut résoudre la contrainte
 */
void resoudre_contrainte_ancre(Ancre* ancre) {
        Particule* particule = ancre -> particule;
        Point* position = ancre -> position;

        /* Pour une ancre: On replace la particule à sa position originelle */
        particule -> position -> x = position -> x;
        particule -> position -> y = position -> y;
}


/** Procédure détruisant l'ancre
 * Argument: ancre: Ancre à détruire
 */
void detruire_ancre(Ancre** ancre) {
        /* On doit juste détruire le point, et non la particule */
        free((*ancre) -> position); (*ancre) -> position = NULL;

        /* Destruction de l'ancre */
        free(*ancre); (*ancre) = NULL;
}
```



```c
/** Procédure utilisée pour l'appel à detruire_ancre dans l'itérateur
 * Argument: ancre: Ancre à détruire
 */
void detruire_ancre_it(Ancre* ancre) {
        detruire_ancre(&ancre);
}

/** Fonction d'égalité de deux ancres par leur position
 * Arguments:
 *      a1: Première ancre
 *      a2: Seconde ancre
 * Retourne: position a1 == position a2
 */
int egalite_ancre_pos(Ancre* a1, Ancre* a2) {
        return (abs(a1 -> position -> x - a2 -> position -> x) <= DELTA) &&
                (abs(a1 -> position -> y -  a2 -> position -> y) <= DELTA);
}
```

```c
/*
 * Projet Système de particules
 *
 * tige.h
 *
 *  Fonctions et type tige.
 *
 *
 */

#ifndef _H_TIGE
#define _H_TIGE

/* Définition du type Tige */
typedef struct {
        Particule* e1; /* La première extrémité */
        Particule* e2; /* La seconde extrémité */
        double longueur; /* La longueur de la tige */
        } Tige;


/** Fonction créant une tige de longueur l et d'extrémité p1 et p2
 * Arguments:
 *      p1: Première extrémité de la tige
 *      p2: Seconde extrémité de la tige
 *      l: Longueur de la tige
 * Retourne: Un pointeur sur la tige créée
 */
Tige* creer_tige(Particule* p1, Particule* p2, double l);

/** Procédure de résolution de contrainte de type "tige"
 * Argument: La tige dont on veut résoudre la contrainte
 */
void resoudre_contrainte_tige(Tige* tige);

/** Procédure détruisant la tige
 * Argument: tige: Tige à détruire
 */
void detruire_tige(Tige** tige);

/** Procédure utilisée pour convertir l'appel à detruire_tigee dans
 * l'itérateur
 * Argument: tige: La tige à détruire
 */
void detruire_tige_it(Tige* tige);

/** Fonction d'égalité de deux tiges par la position d'une de leur extrémité
 * Arguments:
 *      t1: Première tige
 *      t2: Seconde tige
 * Retourne: (extremite 1 de t1 == extremite 1 de t2) || (extremite 2 t1 ==
 * extremite 2 de t2)
 */
int egalite_tige_ext(Tige* t1, Tige* t2);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * tige.c
 *
 *  Fonctions et type tige.
 *
 *
 */
#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "tige.h"

/** Précision utilisée lors de la recherche d'une particule par rapport à sa
 * position
*/
#define DELTA 4

/** Fonction créant une tige de longueur l et d'extrémité p1 et p2
 * Arguments:
 *      p1: Première extrémité de la tige
 *      p2: Seconde extrémité de la tige
 *      l: Longueur de la tige
 * Retourne: Un pointeur sur la tige créée
 */
Tige* creer_tige(Particule* p1, Particule* p2, double l) {
        /* Allocation de mémoire */
        Tige* tige = (Tige*)malloc(sizeof(Tige));

        /* Affectation des valeurs */
        tige -> e1 = p1;
        tige -> e2 = p2;
        tige -> longueur = l;

        return tige;
}

/** Procédure de résolution de contrainte de type "tige"
 * Argument: La tige dont on veut résoudre la contrainte
 * Algorithme basé sur celui décruit sur le site suivant :
 *      www.teknikus.dk/tj/gdc2001.htm
 */
void resoudre_contrainte_tige(Tige* tige) {
        Point* pt1 = tige -> e1 -> position;
        Point* pt2 = tige -> e2 -> position;

        /* On récupère l'inverse des masses des particules */
        double invm1 = tige -> e1 -> masse;
        double invm2 = tige -> e2 -> masse;

        /* On calcule la distance entre les deux points */
        double delta_x = pt2 -> x - pt1 -> x;
        double delta_y = pt2 -> y - pt1 -> y;

        /* delta est la norme du vecteur pt2 - pt1 */
        double delta = sqrt(pow(delta_x,2)+pow(delta_y,2));

        /* La correction à appliquer aux points pour donner longueur fixée */
        double diff = (delta - tige -> longueur)/(delta*(invm1 + invm2));
```



```c
        /* On corrige le premier point */
        pt1 -> x += invm1*delta_x*diff;
        pt1 -> y += invm1*delta_y*diff;

        /* On corrige le second point */
        pt2 -> x -= invm2*delta_x*diff;
        pt2 -> y -= invm2*delta_y*diff;
}

/** Procédure détruisant la tige
 * Argument: tige: Tige à détruire
 */
void detruire_tige(Tige** tige) {
        /* Les particules seront supprimées dans une autre procédure */
        free(*tige); (*tige) = NULL;
}

/** Procédure utilisée pour convertir l'appel à detruire_tigee dans
 * l'itérateur
 * Argument: tige: La tige à détruire
 */
void detruire_tige_it(Tige* tige) {
        detruire_tige(&tige);
}

/** Fonction d'égalité de deux tiges par la position d'une de leur extrémité
 * Arguments:
 *      t1: Première tige
 *      t2: Seconde tige
 * Retourne: (extremite 1 de t1 == extremite 1 de t2) || (extremite 2 t1 ==
 *                                                        extremite 2 de t2)
 */
int egalite_tige_ext(Tige* t1, Tige* t2) {
        return (egalite_particule_pos(t1 -> e1, t2 -> e1) || egalite_particule_p
os(t1 -> e2, t2 -> e2));
}
```

```c
/**
 * Projet Système de particules
 *
 * chaine.h
 *
 * Définition du type chaine.
 *
 */

#ifndef _H_CHAINE
#define _H_CHAINE

/* Définition de la structure type Chaine */
typedef struct {
        Particule* e1; /* Première extrémité de la chaine */
        Particule* e2; /* Seconde extrémité de la chaîne */
        double longueur; /* Longueur (maximale) de la chaine */
        } Chaine;


/** Fonction d'ajout d'une chaine
 * Arguments:
 *      p1: Première particule
 *      p2: Seconde particule
 *      l: Longueur de la chaine
 * Retourne: La chaine qui vient d'être créée
 */
Chaine* creer_chaine(Particule* p1, Particule* p2, double l);

/** Procédure de résolution de contrainte de type "chaine"
 * Argument: La chaine dont on veut résoudre la contrainte
 */
void resoudre_contrainte_chaine(Chaine* chaine);

/** Procédure détruisant la chaine
 * Argument: chaine: Chaine à détruire
 */
void detruire_chaine(Chaine** chaine);

/** Procédure utilisée pour convertir l'appel à detruire_chainee dans
 * l'itérateur
 * Argument: chaine: La chaine à détruire
 */
void detruire_chaine_it(Chaine* chaine);

#endif
```

```c
/**
 * Projet Système de particules
 *
 * chaine.c
 *
 * Implémentation du type chaine.
 *
 */

#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "chaine.h"

/** Fonction d'ajout d'une chaine
 * Arguments:
 *      p1: Première particule
 *      p2: Seconde particule
 *      l: Longueur de la chaine
 * Retourne: La chaine qui vient d'être créée
 */
Chaine* creer_chaine(Particule* p1, Particule* p2, double l) {
        /* Allocation de la mémoire */
        Chaine* chaine = (Chaine*)malloc(sizeof(Chaine));

        /* Affectation des membres */
        chaine -> e1 = p1;
        chaine -> e2 = p2;

        chaine -> longueur = l;

        return chaine;
}

/** Procédure de résolution de contrainte de type "chaine"
 * Argument: La chaine dont on veut résoudre la contrainte
 */
void resoudre_contrainte_chaine(Chaine* chaine) {
        Point* pt1 = chaine -> e1 -> position;
        Point* pt2 = chaine -> e2 -> position;

        /* On récupère l'inverse des masses des particules */
        double invm1 = chaine -> e1 -> masse;
        double invm2 = chaine -> e2 -> masse;

        /* Variables servant pour les calculs */
        double delta_x, delta_y;
        double delta;
        double diff;

        /* On ne modifie la distance que si les deux particules sont éloignées
         * de plus de la longueur de la chaine.
         */
        if(distance(pt1,pt2) > chaine -> longueur) {
                /* On calcule la distance entre les deux points */
                delta_x = pt2 -> x - pt1 -> x;
                delta_y = pt2 -> y - pt1 -> y;

                /* delta est la norme du vecteur pt2 - pt1 */
                delta = sqrt(pow(delta_x,2)+pow(delta_y,2));
```



```c
                /* La correction à appliquer aux points pour donner longueur fix
ée */
                diff = (delta - chaine -> longueur)/(delta*(invm1 + invm2));

                /* On corrige le premier point */
                pt1 -> x += invm1*delta_x*diff;
                pt1 -> y += invm1*delta_y*diff;

                /* On corrige le second point */
                pt2 -> x -= invm2*delta_x*diff;
                pt2 -> y -= invm2*delta_y*diff;
        }
}

/** Procédure détruisant la chaine
 * Argument: chaine: Chaine à détruire
 */
void detruire_chaine(Chaine** chaine) {
        free(*chaine); (*chaine) = NULL;
}

/** Procédure utilisée pour convertir l'appel à detruire_chainee dans
 * l'itérateur
 * Argument: chaine: La chaine à détruire
 */
void detruire_chaine_it(Chaine* chaine) {
        detruire_chaine(&chaine);
}
```

```c
/** Définition de la contrainte espace
 */

#ifndef _H_espace
#define _H_espace
/* Définition de la structure Espace */
typedef struct {
        Particule* e1; /* Première extrémité de l'espace */
        Particule* e2; /* Seconde extrémité de l'espace */
        double longueur; /* Longueur (minimale) de l'espace */
        } Espace;

/** Fonction d'ajout d'un espace
 * Arguments:
 *      p1: Première particule
 *      p2: Seconde particule
 *      l: Longueur de l'espace
 * Retourne: La espace qui vient d'être créée
 */
Espace* creer_espace(Particule* p1, Particule* p2, double l);

/** Procédure de résolution de contrainte de type "espace"
 * Argument: L'espace dont on veut résoudre la contrainte
 */
void resoudre_contrainte_espace(Espace* espace);

/** Procédure détruisant l'espace
 * Argument: espace: Espace à détruire
 */
void detruire_espace(Espace** espace);

/** Procédure utilisée pour convertir l'appel à detruire_espace dans
 * l'itérateur
 * Argument: espace: L'espace à détruire
 */
void detruire_espace_it(Espace* espace);

#endif
```

```c
/**
 * Projet Système de particules
 *
 * espace.c
 *
 * Implémentation du type espace
 *
 */
#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "espace.h"

/** Fonction d'ajout d'un espace
 * Arguments:
 *      p1: Première particule
 *      p2: Seconde particule
 *      l: Longueur de l'espace
 * Retourne: L'espace qui vient d'être créé
 */
Espace* creer_espace(Particule* p1, Particule* p2, double l) {
        /* Allocation de la mémoire */
        Espace* espace = (Espace*)malloc(sizeof(Espace));

        /* Affectation des membres */
        espace -> e1 = p1;
        espace -> e2 = p2;

        espace -> longueur = l;

        return espace;
}

/** Procédure de résolution de contrainte de type "espace"
 * Argument: L'espace dont on veut résoudre la contrainte
 */
void resoudre_contrainte_espace(Espace* espace) {
        Point* pt1 = espace -> e1 -> position;
        Point* pt2 = espace -> e2 -> position;

        /* On récupère l'inverse des masses des particules */
        double invm1 = espace -> e1 -> masse;
        double invm2 = espace -> e2 -> masse;

        /* Variables servant pour les calculs */
        double delta_x, delta_y;
        double delta;
        double diff;

        /* On ne modifie la distance que si les deux particules sont éloignées
         * de moins de la longueur de l'espace.
         */
        if(distance(pt1,pt2) < espace -> longueur) {


                /* On calcule la distance entre les deux points */
                delta_x = pt2 -> x - pt1 -> x;
                delta_y = pt2 -> y - pt1 -> y;

                /* delta est la norme du vecteur pt2 - pt1 */
```



```c
                delta = sqrt(pow(delta_x,2)+pow(delta_y,2));

                /* La correction à appliquer aux points pour donner longueur fix
ée */
                diff = (delta - espace -> longueur)/(delta*(invm1 + invm2));

                /* On corrige le premier point */
                pt1 -> x += invm1*delta_x*diff;
                pt1 -> y += invm1*delta_y*diff;

                /* On corrige le second point */
                pt2 -> x -= invm2*delta_x*diff;
                pt2 -> y -= invm2*delta_y*diff;
        }
}

/** Procédure détruisant l'espace
 * Argument: espace: Espace à détruire
 */
void detruire_espace(Espace** espace) {
        free(*espace); (*espace) = NULL;
}

/** Procédure utilisée pour convertir l'appel à detruire_espace dans
 * l'itérateur
 * Argument: espace: L'espace à détruire
 */
void detruire_espace_it(Espace* espace) {
        detruire_espace(&espace);
}
```

```c
/*
 * Projet Système de particules
 *
 * ressort.h
 *
 *  Fonctions et type ressort.
 *
 *
 */

#ifndef _H_RESSORT
#define _H_RESSORT

/* Définition du type Ressort */
typedef struct {
        Particule* e1; /* Première extrémité du ressort */
        Particule* e2; /* Seconde extrémité */
        double longueur; /* Longueur au repos du ressort */
        double coefficient; /* Coefficient de raideur du ressort */
        } Ressort;


/** Fonction créant un ressort de longueur l, de coefficient k
 *et d'extrémité p1 et p2
 * Arguments:
 *      p1: Première extrémité
 *      p2: Seconde extrémité
 *      l: Longueur de la tige
 *      k: Coefficient de raideur du ressort
 * Retourne: Un pointeur sur le ressort créé
 */
Ressort* creer_ressort(Particule* p1, Particule* p2, double l, double k);

/** Procédure de calcul ds forces d'appliquant sur chacune des extrémités du
 * ressort
 */
void calculer_force_ressort(Ressort* ressort);

/** Procédure détruisant le ressort
 * Argument: ressort: Ressort à détruire
 */
void detruire_ressort(Ressort** ressort);

/** Procédure utilisée pour convertir l'appel à detruire_ressort dans
 * l'itérateur
 * Argument: ressort: Le ressort à détruire
 */
void detruire_ressort_it(Ressort* ressort);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * ressort.c
 *
 *  Fonctions et type ressort.
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "geometrie.h"
#include "list.h"
#include "particule.h"
#include "ressort.h"

/** Coefficient multiplicateur utilisé lors du calcul de la force exercée par
 * le ressort sur les particules
 */
#define MULT 1

/** Fonction créant un ressort de longueur l, de coefficient k
 *et d'extrémité p1 et p2
 * Arguments:
 *      p1: Première extrémité
 *      p2: Seconde extrémité
 *      l: Longueur de la tige
 *      k: Coefficient de raideur du ressort
 * Retourne: Un pointeur sur le ressort créé
 */
Ressort* creer_ressort(Particule* p1, Particule* p2, double l, double k) {
        /* Allocation de mémoire */
        Ressort* ressort = (Ressort*)malloc(sizeof(Ressort));

        /* Affectation des champs */
        ressort -> e1 = p1;
        ressort -> e2 = p2;
        ressort -> longueur = l;
        ressort -> coefficient = k;

        return ressort;
}

/** Procédure de calcul ds forces d'appliquant sur chacune des extrémités du
 * ressort
 */
void calculer_force_ressort(Ressort* ressort) {
        /* Principe: On va calculer les forces appliquées sur chacune des
         * extrémités du ressort, et les rajouter dans la liste de forces de
         * chacune des particules
         */

        /* Pour alleger les notations */
        Particule* p1 = ressort -> e1;
        Particule* p2 = ressort -> e2;

        /* Calcul de la distance entre les deux points */
        double dist = distance(p1 -> position, p2 -> position);

        /* Coefficient multiplicateur utilisée dans la norme des forces */
        double coef =
```



```c
                (dist - ressort -> longueur)*(ressort -> coefficient)/(2*dist*MU
LT);

        /* Composante en X */
        double c_x = coef*(p2 -> position -> x - p1 -> position -> x);
        /* Composante en Y */
        double c_y = coef*(p2 -> position -> y - p1 -> position -> y);

        /* Calcul de f12 */
        Vecteur* f12 = creer_vecteur(c_x,c_y);
        /* Calcul de f21 */
        Vecteur* f21 = creer_vecteur(-c_x,-c_y);

        /* Ajout des forces aux particules */
        cons(f12, &(p1 -> forces));
        cons(f21, &(p2 -> forces));

}

/** Procédure détruisant le ressort
 * Argument: ressort: Ressort à détruire
 */
void detruire_ressort(Ressort** ressort) {
        free(*ressort); (*ressort)=NULL;
}

/** Procédure utilisée pour convertir l'appel à detruire_ressort dans
 * l'itérateur
 * Argument: ressort: Le ressort à détruire
 */
void detruire_ressort_it(Ressort* ressort) {
        detruire_ressort(&ressort);
}
```

```c
/*
 * Projet Système de particules
 *
 * systeme.h
 *
 *  Fonctions et types système de particules.
 *
 *
 */

#ifndef _H_SYSTEME
#define _H_SYSTEME

/* Définition du type Systeme */
typedef struct {

        /* Les paramètres physiques du systeme */
        Vecteur* gravite; /* Le vecteur gravité */

        double coef; /* Coefficient de viscosité */

        /* La liste des éléments du système */
        T_LIST particules; /* Liste de particules */

        T_LIST tiges; /* Liste de tiges */

        T_LIST ancres; /* Liste d'ancres */

        T_LIST chaines; /* Liste de chaines */

        T_LIST espaces; /* Liste d'espaces */

        T_LIST ressorts; /* Liste de ressorts */

        /* Les données nécessaires pour l'interaction avec la souris */
        Ancre* deplace; /* Ancre qui sera déplacée par la souris */

        int destructible; /* Indique si l'ancre précédente peut être détruite */

        } Systeme;


/** Fonction qui créant un systeme vide
 * Arguments:
 *      gx: Norme en X de la gravité
 *      gy: Norme en Y de la gravité
 *      k: Coefficient de viscosité
 * Retourne: Pointeur sur un systeme
 */
Systeme* creer_systeme(double gx, double gy, double k);

/** Procédure créant et ajoutant une particule à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      x: Abscisse de la particule
 *      y: Ordonnée de la particule
 *      m: masse de la particule
 *      id: identifiant de la particule
 */
void ajouter_particule(Systeme* systeme, double x, double y, double m, int id);

/** Procédure créant et ajoutant une ancre à un système
```



```c
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      x: Abscisse de l'ancre
 *      y: Ordonnée
 *      id: Identifiant de la particule associée
 */
void ajouter_ancre(Systeme* systeme, double x, double y, int id);

/** Procédure créant et ajoutant une tige à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de la tige
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_tige(Systeme* systeme, double l, int id1, int id2);

/** Procédure créant et ajoutant une chaine à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de la chaine
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_chaine(Systeme* systeme, double l, int id1, int id2);

/** Procédure créant et ajoutant un espace à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de l'espace
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_espace(Systeme* systeme, double l, int id1, int id2);

/** Procédure créant et ajoutant un ressort à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur du ressort
 *      k: coefficient du ressort
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_ressort(Systeme* systeme, double l, double k, int id1, int id2);

/** Procédure intégrant le systeme
 * Argument: Le système à intégrer
 */
void verlet(Systeme* sys);

/** Procédure résolvant les contraintes
 * Argument: Le système dont on souhaite satisfaire les contraintes
 */
void satisfaire_contraintes(Systeme* sys);

/** Procédure détruisant le système
 * Argument: Système à détruire
 */
void detruire_systeme(Systeme* sys);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * systeme.c
 *
 *  Fonctions et types système de particules.
 *
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "ancre.h"
#include "tige.h"
#include "chaine.h"
#include "espace.h"
#include "ressort.h"
#include "systeme.h"

/* Le nombre d'itérations pour résoudre les contraintes */
#define NB_ITER 100

/* L'intervalle entre chaque appel à la fonction "verlet" */
#define DT 0.5

/** Fonction qui créant un systeme vide
 * Arguments:
 *      gx: Norme en X de la gravité
 *      gy: Norme en Y de la gravité
 *      k: Coefficient de viscosité
 * Retourne: Pointeur sur un systeme
 */
Systeme* creer_systeme(double gx, double gy, double k) {
        /* On alloue de la mémoire pour le système... */
        Systeme* sys = (Systeme*)malloc(sizeof(Systeme));

        /* Affectation des paramètres du système */
        sys -> coef = k;
        sys -> gravite = creer_vecteur(gx,gy);

        /* Aucune particule n'est déplacée */
        sys -> deplace = NULL;

        /* Les listes sont vides */
        sys -> particules = nil();
        sys -> tiges = nil();
        sys -> ancres = nil();
        sys -> chaines = nil();
        sys -> espaces = nil();
        sys -> ressorts = nil();
        sys -> destructible = 0;

        return sys;
}


/** Procédure créant et ajoutant une particule à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
```



```c
 *      x: Abscisse de la particule
 *      y: Ordonnée de la particule
 *      m: masse de la particule
 *      id: identifiant de la particule
 */
void ajouter_particule(Systeme* systeme, double x, double y, double m, int id) {
        /* Création de la particule */
        Particule *part = creer_particule(x,y,m,id);

        /* Ajout de la particule à la liste */
        cons(part,&(systeme -> particules));
}

/** Procédure créant et ajoutant une ancre à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      x: Abscisse de l'ancre
 *      y: Ordonnée
 *      id: Identifiant de la particule associée
 */
void ajouter_ancre(Systeme* systeme, double x, double y, int id){
        /* Création du particule temporaire servant à la recherche */
        Particule* tmp = creer_particule(0,0,0,id);

        /* On récupère la particule d'identifiant id */
        Particule *part = member(tmp,systeme -> particules, egalite_particule_id
);

        /* On vérifie que la particule existe */
        if(part == NULL) {
                fprintf(stderr, "Pas de particule n°%d\n",id);
        } else {
                /* Création de l'ancre */
                Ancre* ancre = creer_ancre(part,x,y);

                /* Ajout de l'ancre au système */
                cons(ancre,&(systeme->ancres));
        }


        /* Suppression de la particule */
        detruire_particule(&tmp);
}

/** Procédure créant et ajoutant une tige à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de la tige
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_tige(Systeme* systeme, double l, int id1, int id2){
        /* Création des particules "motif" */
        Particule* tmp1 = creer_particule(0,0,0,id1);
        Particule* tmp2 = creer_particule(0,0,0,id2);

        /* Recherche des particules par identifiant */
        Particule* part1 = member(tmp1,systeme -> particules,egalite_particule_i
d);
        Particule* part2 = member(tmp2,systeme -> particules,egalite_particule_i
d);
```

```
        /* On vérifie que les particules existent */
        if((part1 == NULL) || (part2 == NULL)){
                fprintf(stderr,"L'une des particules n°%d ou %d n'existe pas\n",
                        id1,id2);
        } else {
                /* Création de la tige */
                Tige* tige = creer_tige(part1, part2, l);

                /* Ajout de la tige au système */
                cons(tige,&(systeme->tiges));
        }
        /* Suppression des particules temporaires */
        detruire_particule(&tmp1);
        detruire_particule(&tmp2);
}


/** Procédure créant et ajoutant une chaine à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de la chaine
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_chaine(Systeme* systeme, double l, int id1, int id2){
        /* Création des particules "motif" */
        Particule* tmp1 = creer_particule(0,0,0,id1);
        Particule* tmp2 = creer_particule(0,0,0,id2);

        /* Recherche des particules par identifiant */
        Particule* part1 = member(tmp1,systeme -> particules,egalite_particule_i
d);
        Particule* part2 = member(tmp2,systeme -> particules,egalite_particule_i
d);

        /* Test du retour de la fonction member */
        if((part1 == NULL) || (part2 == NULL)){
                fprintf(stderr,"L'une des deux particules n°%d ou %d n'existe pas\n",
                        id1, id2);
        } else {
                /* Création de la chaine */
                Chaine* chaine = creer_chaine(part1, part2, l);

                /* Ajout de la chaine au système */
                cons(chaine,&(systeme->chaines));
        }

        /* Suppression des particules temporaires */
        detruire_particule(&tmp1);
        detruire_particule(&tmp2);
}


/** Procédure créant et ajoutant un espace à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur de l'espace
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_espace(Systeme* systeme, double l, int id1, int id2){
        /* Création des particules "motif" */
```



```
        Particule* tmp1 = creer_particule(0,0,0,id1);
        Particule* tmp2 = creer_particule(0,0,0,id2);

        /* Recherche des particules par identifiant */
        Particule* part1 = member(tmp1,systeme -> particules,egalite_particule_i
d);
        Particule* part2 = member(tmp2,systeme -> particules,egalite_particule_i
d);

        /* Test du retour de la fonction member */
        if((part1 == NULL) || (part2 == NULL)){
                fprintf(stderr,"L'une des deux particules n°%d ou %d n'existe pas\n",
                        id1, id2);
        } else {
                /* Création de l'espace */
                Espace* Espace = creer_espace(part1, part2, l);

                /* Ajout de l'espace au système */
                cons(Espace,&(systeme->espaces));
        }

        /* Suppression des particules temporaires */
        detruire_particule(&tmp1);
        detruire_particule(&tmp2);
}



/** Procédure créant et ajoutant un ressort à un système
 * Arguments:
 *      systeme: Le systeme que l'on modifie
 *      l: longueur du ressort
 *      k: coefficient du ressort
 *      id1: Identifiant de la première particule
 *      id2: Identifiant de la seconde particule
 */
void ajouter_ressort(Systeme* systeme, double l, double k, int id1, int id2){
        /* Création des particules "motif" */
        Particule* tmp1 = creer_particule(0,0,0,id1);
        Particule* tmp2 = creer_particule(0,0,0,id2);

        /* Recherche des particules par identifiant */
        Particule* part1 = member(tmp1,systeme -> particules,egalite_particule_i
d);
        Particule* part2 = member(tmp2,systeme -> particules,egalite_particule_i
d);

        /* On teste si les deux particules existent */
        if((part1 == NULL) || (part2 == NULL)){
                fprintf(stderr,"L'une des particules n°%d ou %d n'existe pas\n",
                        id1,id2);
        } else {
                /* Création du ressort */
                Ressort* ressort = creer_ressort(part1, part2, l, k);

                /* Ajout du ressort au système */
                cons(ressort,&(systeme->ressorts));
        }

        /* Suppression des particules temporaires */
        detruire_particule(&tmp1);
        detruire_particule(&tmp2);
```

```
}


/** Procédure intégrant le systeme
 * Argument: Le système à intégrer
 */
void verlet(Systeme* sys){

        int ret; /* Variable utilisée pour la fonction head */

        Particule* part;

        /* La liste des particules du systeme */
        T_LIST l_part = sys -> particules;

        /* On parcourt la liste des particules */
        while(!is_empty(l_part)){
                /* Récupération de la tête de la liste */
                part = (Particule*) head(l_part,&ret);
                /* On calcule la nouvelle position de la particule */
                integrer(part, DT, sys -> gravite, sys -> coef);
                /* On récupère le reste de la liste */
                l_part = tail(l_part,&ret);
        }

}

/** Procédure résolvant les contraintes
 * Argument: Le système dont on souhaite satisfaire les contraintes
 */
void satisfaire_contraintes(Systeme* sys) {

        int i;

        /* RAZ des forces sur les particules */
        do_list(raz_forces, sys -> particules);

        /* Calcul des forces exercées par les ressorts */
        do_list(calculer_force_ressort, sys -> ressorts);

        /* Calcul de la nouvelle position des particules */
        verlet(sys);

        /* Résolution des contraintes */
        for(i=0;i<NB_ITER;i++){

                /* On résout d'abord les contraintes sur les tiges */
                do_list(resoudre_contrainte_tige, sys -> tiges);
                /* Puis sur les chaines */
                do_list(resoudre_contrainte_chaine, sys -> chaines);
                /* Puis les espaces */
                do_list(resoudre_contrainte_espace, sys -> espaces);
                /* Puis sur les ancres */
                do_list(resoudre_contrainte_ancre, sys -> ancres);

                /* Si le système contient une ancre en train d'être déplacée, il
 faut
                 * aussi la traiter dans les contraintes */
                if(sys -> deplace != NULL){
                        resoudre_contrainte_ancre(sys -> deplace);
                }
        }
```



```
}

/** Procédure détruisant le système
 * Argument: Système à détruire
 */
void detruire_systeme(Systeme* sys) {

        /* On commence par détruire chacun des objets du systeme */
        do_list(detruire_ancre_it,sys -> ancres);
        do_list(detruire_tige_it,sys -> tiges);
        do_list(detruire_chaine_it, sys -> chaines);
        do_list(detruire_espace_it, sys -> espaces);
        do_list(detruire_part_it, sys -> particules);
        do_list(detruire_ressort_it, sys -> ressorts);

        /* On libère la mémoire du système */
        free(sys);

        printf("Système détruit !\n");
}
```

```
/*
 * Projet Système de particules
 *
 * parseur.h
 *
 *  Fonctions pour parser le fichier de description du système de particules
 *
 *
 */

/** Fonction de création du système à partir d'un fichier, passé en paramètre
 * Argument:
 *      fichier: Chemin du fichier passé en paramètre.
 * Retourne: Pointeur sur un objet systeme
 */
Systeme* charger(char* fichier);
```

```c
/*
 * Projet Système de particules
 *
 * parseur.c
 *
 *  Fonctions pour parser le fichier de description du système de particules
 *
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include "geometrie.h"
#include "list.h"
#include "particule.h"
#include "ancre.h"
#include "systeme.h"
#include "parseur.h"

#define NB_CHAR_LG 255 /* Nombre de caractères par ligne */
#define EPS 1e-8 /* Masse minimale d'une particule */

/** Procédure qui déroule un fichier jusqu'a une ligne sans commentaire
 * Arguments:
 *      file: fichier à lire
 *      ligne: variable ou l'ont stocke la ligne
 */
void parse_commentaire(FILE* file, char* ligne){

        while((fgets(ligne, NB_CHAR_LG, file) != NULL)&&(ligne[0]=='#'));

}

/** Procédure lisant la configuration d'un particule avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter la particule
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 *      id: identifiant de la particule
 */
void lire_particule(Systeme* systeme, FILE* file, char* ligne, int id){
        double x;
        double y;
        double m;

        int retour; /* Contient le nombre d'élément lus */

        /* On saute les commentaires */
        parse_commentaire(file,ligne);

        /* Lecture d'une particule, au format
         *      x y masse
         */
        retour = sscanf(ligne,"%lf %lf %lf", &x, &y, &m);
        if(retour != 3) {
        /* Un problème dans le format de la particule */
                fprintf(stderr, "Erreur: Particule %d au format incorrect ! \n",id);
        } else if(m < EPS){
                fprintf(stderr, "Erreur: Particule %d de masse nulle !\n",id);
        } else {
                printf("Particule n° %d de position (%g,%g), de masse %g\n",id,x,y,m);

                /* Ajout de la particule au système */
```



```c
                ajouter_particule(systeme,x,y,m,id);
        }
}

/** Procédure lisant la configuration d'une ancre avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter la particule
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 */
void lire_ancre(Systeme* systeme, FILE* file, char* ligne){
        double x;
        double y;
        int id;

        int retour; /* Contiendra le nombre de données lues */

        /* On saute les commentaires */
        parse_commentaire(file,ligne);

        /* Lecture d'une ancre au format
         *      x y n
         */
        retour = sscanf(ligne,"%lf %lf %d", &x, &y, &id);

        if(retour != 3) {
                fprintf(stderr, "Erreur dans le format de l'ancre %d ! \n",id);
        } else {
                printf("Ancre de position (%g,%g), associée à la particule %d\n",x,y,id);

                /* Ajout de l'ancre au système */
                ajouter_ancre(systeme,x,y,id);
        }
}

/** Procédure lisant la configuration d'une tige avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter la particule
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 */
void lire_tige(Systeme* systeme, FILE* file, char* ligne){
        int id1;
        int id2;
        double l;

        int retour; /* Contiendra le nombre de données lues */

        /* On saute les commentaires */
        parse_commentaire(file,ligne);

        /* Lecture d'une tige au format
         *      longueur n1 n2
         */
        retour = sscanf(ligne,"%lf %d %d", &l,&id1,&id2);
        if(retour != 3) {
                fprintf(stderr, "Erreur: Tige au format incorrect ! \n");
        } else {
                printf("Tige de longueur %g, associée aux particules %d - %d\n",l,id1,id2);

                /* Ajout de la tige au systeme */
                ajouter_tige(systeme,l,id1,id2);
```

```
        }
}

/** Procédure lisant la configuration d'une chaine avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter la particule
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 */
void lire_chaine(Systeme* systeme, FILE* file, char* ligne){
        int id1;
        int id2;
        double l;

        int retour; /* Variable contenant le nombre de données lues */

        /* On saute les commentaires */
        parse_commentaire(file,ligne);

        /* Lecture d'une chaine au format
         *      longueur n1 n2
         */
        retour = sscanf(ligne,"%lf %d %d", &l,&id1,&id2);
        if(retour != 3){
                fprintf(stderr, "Erreur: Chaine de format incorrect !\n");
        } else {
                printf("Chaine de longueur %g, associée aux particules %d - %d\n",l,id1,id2);

                /* Ajout de la chaine au systeme */
                ajouter_chaine(systeme,l,id1,id2);
        }
}

/** Procédure lisant la configuration d'un espace avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter l'espace
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 */
void lire_espace(Systeme* systeme, FILE* file, char* ligne){
        int id1;
        int id2;
        double l;

        int retour; /* Variable contenant le nombre de données lues */

        /* On saute les commenatires */
        parse_commentaire(file,ligne);

        /* Lecture d'un espace au format
         *      longueur n1 n2
         */
        retour = sscanf(ligne,"%lf %d %d", &l,&id1,&id2);
        if(retour != 3){
                fprintf(stderr, "Erreur: Espace de format incorrect !\n");
        } else {
                printf("Espace de longueur %g, associé aux particules %d - %d\n",l,id1,id2);

                /* Ajout de l'espace au systeme */
                ajouter_espace(systeme,l,id1,id2);
        }
}
```



```
/** Procédure lisant la configuration d'un ressort avec ajout au système
 * Arguments:
 *      systeme: Le systeme auquel ajouter l'espace
 *      file: le fichier de configuration
 *      ligne: la ligne à lire
 */
void lire_ressort(Systeme* systeme, FILE* file, char* ligne){
        int id1;
        int id2;
        double l;
        double k;

        int retour; /* Variable contenant le nombre de données lues */

        /* On saute les commenatires */
        parse_commentaire(file,ligne);

        /* Lecture d'un espace au format
         *      longueur coefficient n1 n2
         */
        retour = sscanf(ligne,"%lf %lf %d %d", &l,&k,&id1,&id2);
        if(retour != 4){
                fprintf(stderr, "Erreur: Ressort de format incorrect !\n");
        } else {
                printf("Ressort de longueur %g, de coefficient %g, associé aux particules %d - %d\n",l,
k,id1,id2);

                /* Ajout de l'espace au systeme */
                ajouter_ressort(systeme,l,k,id1,id2);
        }
}
/** Fonction de création du système à partir d'un fichier, passé en paramètre
 * Argument:
 *      fichier: Chemin du fichier passé en paramètre.
 * Retourne: Pointeur sur un objet systeme
 */
Systeme* charger(char* fichier) {
        /* Variable stockant la ligne lue */
        char ligne[NB_CHAR_LG];

        /* Le système créé */
        Systeme* sys;

        /* Le FILE servant à la lecture du fichier */
        FILE* file;

        /* Les variables servant à parser le fichier */
        double g1,g2,k;
        int nb;
        int i;

        int retour; /* Variable contenant le nombre de données lues */

        /* Ouverture du fichier */
        printf("Ouverture du fichier... %s\n", fichier);
        file = fopen(fichier,"r");

        /* On teste si l'ouverture a réussie */
        if(file != NULL){
                /* Si l'ouverture a réussie */
                puts("Fichier ouvert correctement");
```

```
                /* On ignore les commentaires */
                parse_commentaire(file, ligne);

                /* Lecture du vecteur gravité */
                retour = sscanf(ligne, "%lf %lf",&g1,&g2);
                if(retour != 2) {
                        fprintf(stderr,"Erreur dans la définition de la gravité !\n On prendra les val
eurs \"usuelles\"\n");
                        g1 = 0.0;
                        g2 = 9.8;
                } else {
                        printf("Gravité : (%g, %g)\n",g1,g2);
                }

                parse_commentaire(file,ligne);

                /* Lecture du coef de viscosité */
                retour = sscanf(ligne,"%lf",&k);
                if(retour != 1) {
                        fprintf(stderr,"Erreur dans le coefficient de friction !\n On prendra une val
eur par défaut\"\n");
                        k=0.05;
                } else {
                        printf("Coef de viscosite: %g\n",k);
                }

                /* Création du système */
                puts("Création du système...");
                sys = creer_systeme(g1,g2,k);
                if(sys == NULL){
                        fprintf(stderr, "Erreur d'allocation mémoire !\n");
                        return NULL;
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre de particules */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier !\n");
                        return NULL;
                }
                printf("%d particules\n",nb);

                /* On lit les particules une à une*/
                for(i=0;i<nb;i++){
                        lire_particule(sys,file,ligne,i);
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre d'ancres */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier\n");
                        return NULL;
                }
                printf("%d ancres\n",nb);

                /* On lit les ancres une à une */
                for(i=0;i<nb;i++){
```



```
                        lire_ancre(sys,file,ligne);
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre de tiges */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier !\n");
                        return NULL;
                }
                printf("%d tiges\n",nb);

                /* On lit les tiges une à une */
                for(i=0;i<nb;i++){
                        lire_tige(sys,file,ligne);
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre de chaine */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier !\n");
                        return NULL;
                }
                printf("%d chaines\n",nb);

                /* On lit les chaines une à une */
                for(i=0;i<nb;i++){
                        lire_chaine(sys,file,ligne);
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre d'espaces */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier !\n");
                        return NULL;
                }
                printf("%d espaces\n",nb);

                /* On lit les espaces un à un */
                for(i=0;i<nb;i++){
                        lire_espace(sys,file,ligne);
                }

                parse_commentaire(file,ligne);

                /* Lecture du nombre de ressorts */
                retour = sscanf(ligne,"%d",&nb);
                if(retour != 1){
                        fprintf(stderr, "Erreur dans le format du fichier !\n");
                        return NULL;
                }
                printf("%d ressorts\n",nb);

                /* On lit les ressorts un à un */
                for(i=0;i<nb;i++){
                        lire_ressort(sys,file,ligne);
                }
```

```
                /* À finir : Lecture des contraintes supplémentaires */

                /* Fermeture du fichier */
                fclose(file);

        }else{
                /* En cas d'erreur */
                fprintf(stderr, "Erreur d'Entrée/Sortie à l'ouverture du fichier !\n");
                return NULL;
        }

        return sys;
}
```

```c
/*
 * Projet Système de particules
 *
 * interaction.h
 *
 * Fonctions de manipulation de l'IHM
 *
 */

#ifndef __INTERACTION_H__
#define __INTERACTION_H__

#include "geometrie.h"

/*
 * L'utilisateur a cliqué (bouton enfoncé) dans la fenêtre au point (x,y)
 */
extern void on_viewport_mouse_press(double x, double y);

/*
 * Le bouton de la souris a été relaché.
 */
extern void on_viewport_mouse_release();

/*
 * Autorise/Désactive les événements cliquer-glisser.
 * Appelé lorsque un object a été sélectionné
 */
extern void viewport_activate_drag(int activate);

/*
 * L'utilisateur a fait un cliquer-glisser dans la fenêtre.
 *
 * Le cliqué est en (orig_x, orig_y) et la destination
 * (position courante de la souris) est (dest_x, dest_y).
 *
 */
extern void on_viewport_mouse_drag(double orig_x, double orig_y,
                                   double dest_x, double dest_y);

/*
 * Efface la fenêtre d'affichage (blanc)
 */
extern void viewport_clear();

/*
 * Dessine un point aux coordonnées (x,y)
 * dans la fenêtre.
 */
extern void viewport_draw_point(double x, double y, int size);

/*
 * Dessine un segment entre les points (x1,y1) et (x2,y2)
 * width est l'épaisseur de la ligne.
 */
extern void viewport_draw_line(double x1, double y1, double x2, double y2, int w
idth);

/*
 * Dessine un polygone fermé constitué des points données.
 */
extern void viewport_draw_polygon(Point *points, int nb_points, int plein, int l
```



```c
ine_width);

/*
 * Change la couleur des dessins en cours
 * red:   proportion de rouge (entre 0.0 et 1.0)
 * blue:  proportion de bleu (entre 0.0 et 1.0)
 * green: proportion de vert (entre 0.0 et 1.0)
 */
extern void viewport_change_color(double red, double green, double blue);

#endif
```

```c
/*
 * Projet Système de particules
 *
 * interaction.c
 *
 * Fonctions de manipulation de l'IHM
 *
 */

#include <stdio.h>
#include <math.h>
#include "interaction.h"
#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "ancre.h"
#include "tige.h"
#include "systeme.h"
#include "parseur.h"

/** Si la macro CONTRAINTE est définie, alors on compile en mode "respect des
 * contraintes lors du déplacement. (voir plus bas, fonction
 * on_viewport_mouse_drag
 */
#define CONTRAINTE


/** Pointeur global vers le systeme déclaré dans le fichier
 * "visualiseur.c". On partage ainsi le même objet
*/
extern Systeme *systeme;

void on_viewport_mouse_press(double x, double y){
  /* Le pointeur sur l'ancre la plus proche */
  Ancre* ancre;

  /* Le pointeur sur la particule la plus proche */
  Particule* part;

  /* On crée des éléments temporaires servant à la recherche */
  Ancre* a_tmp = creer_ancre(NULL,x,y);
  Particule* p_tmp = creer_particule(x,y,0,-1);

  printf ("Bouton enfoncé: (%f, %f)\n", x, y);

  /** Recherche de l'ancre ou de la particule la plus proche de la souris **/

  /* ancre pointe sur l'ancre la plus proche (si elle existe) */
  ancre = member(a_tmp,systeme -> ancres, egalite_ancre_pos);


  /* On regarde si une ancre a été trouvée */
  if(ancre != NULL){
          /* Si une ancre se trouve à proximité */
          printf("Saisie de l'ancre associée à la particule n°%d\n",ancre -> particule -> id);
          /* On l'ajoute comme ancre déplaçable */
          systeme -> deplace = ancre;
          /* On indique qu'il ne faudra pas la détruire */
          systeme -> destructible = 0;
  } else {

          /* part pointe sur la particule la plus proche (si elle existe)*/
```



```c
          part = member(p_tmp,systeme -> particules, egalite_particule_pos);

          /* On regarde si une particule a été trouvée */
          if(part != NULL){
                  /* Si une particule se trouve à proximité */
                  printf("Saisie de la particule n°%d\n", part -> id);
                  /* Elle est ajoutée comme ancre dans le système */
                  systeme -> deplace = creer_ancre(part, x, y);
                  /* On indique qu'elle peut être détruite */
                  systeme -> destructible = 1;
          }
  }

  /* Destruction des éléments temporaires */
  detruire_particule(&p_tmp);
  detruire_ancre(&a_tmp);


}

void on_viewport_mouse_release(){

  printf ("Bouton relaché\n");

  /* Il faut détruire l'ancre */
  if((systeme -> destructible) && (systeme -> deplace != NULL)){
        detruire_ancre(&(systeme -> deplace));
  }

  /* On ne déplace plus l'ancre */
  systeme -> deplace = NULL;

}

/** Remarque :
 *      Il y a deux "modes", choisies par une compilation conditionnelle
 *       -> Le premier (constante CONTRAINTE non-définie) n'impose aucun respect
 * des contraintes lors du déplacement. Le système est donc plus fluide.
 * Cependant, on subit un effet "elastique".
 *       -> Le second (constante CONTRAINTE définie) empêche l'utilisateur de
 * tirer une particule sans respecter les contraintes qui lui sont appliquées.
 * Cependant, la fluidité de la simulation est altérée.
 */
void on_viewport_mouse_drag(double orig_x, double orig_y,
                            double dest_x, double dest_y){
  #ifdef CONTRAINTE
  Ancre* tmp;
  #endif

  printf ("Cliqué glisser de (%f,%f) - (%f,%f)\n",
                    orig_x, orig_y, dest_x, dest_y);

  /* Si une particule a été trouvée */
  if(systeme -> deplace != NULL){

        #ifdef CONTRAINTE
        /** "Mode respect des contraintes pendant le déplacement" **/

        /* On MAJ la position de la particule directement */
        systeme -> deplace -> particule -> position -> x = dest_x;
        systeme -> deplace -> particule -> position -> y = dest_y;
```

```
        /* Si la particule cliquée était déjà une ancre,
         * on déplace l'ancre aussi
         */
        if(!(systeme -> destructible)) {
                systeme -> deplace -> position -> x = dest_x;
                systeme -> deplace -> position -> y = dest_y;
        }

        /* On garde un pointeur sur l'ancre pointée */
        tmp = systeme -> deplace;

        /* On "annule" temporairement la contrainte ancre */
        systeme -> deplace = NULL;

        /* Résolution des contraintes -> la particule pointée est donc ramenée à
         * sa position correcte, dans le respect des contraintes */
        satisfaire_contraintes(systeme);

        /* On réinscrit l'ancre dans le système */
        systeme -> deplace = tmp;

        /* Puis mise à jour des coordonnées de l'ancre à l'aide des coordonées
         * de la particule
         *      -> Pas très très beau
         */
        systeme -> deplace -> position -> x = systeme -> deplace -> particule ->
 position -> x;
        systeme -> deplace -> position -> y = systeme -> deplace -> particule ->
 position -> y;

        #else
        /** Mode sans respect des contraintes pendant le déplacement */
        /* On ne déplace que l'ancre, la particule sera déplacée lors de la
         * résolution des contraintes
         */
                systeme -> deplace -> position -> x = dest_x;
                systeme -> deplace -> position -> y = dest_y;
         #endif
  }

}
```

```c
/*
 * Programme principal.
 * Lance une fenetre de visualisation et d'interaction pour la
 * manipulation du système de particules
 *
 *
 */

#include <gtk/gtk.h>
#include <locale.h>
#include "interaction.h"

#include "list.h"
#include "geometrie.h"
#include "particule.h"
#include "ancre.h"
#include "tige.h"
#include "chaine.h"
#include "espace.h"
#include "ressort.h"
#include "systeme.h"
#include "parseur.h"


/* Taille des divers éléments dessinés */
#define TAILLE_PARTICULE 4
#define TAILLE_ANCRE 6
#define TAILLE_TIGE 3
#define TAILLE_RESSORT 5

#define VIEWPORT_WIDTH   400
#define VIEWPORT_HEIGHT  300

typedef struct {
  GtkWidget *viewport;
  GdkPixmap *image;
  GdkGC *gc;
  GdkColor color;
  gboolean receive_drag;
  gdouble drag_orig_x;
  gdouble drag_orig_y;
} Visualiseur;

static Visualiseur visualiseur;

/* Pointeur global sur le type Systeme */
Systeme* systeme;

/**********************************
 *
 * Interaction avec la souris
 *
 **********************************/

void viewport_activate_drag(int activate) {
  visualiseur.receive_drag = activate;
}

static gboolean on_viewport_motion(GtkWidget      *widget,
                                   GdkEventMotion  *event,
                                   Visualiseur     *visualiseur) {
  gdouble x, y;
```



```c
  x = event->x;
  y = event->y;

  if (visualiseur->receive_drag) {
    on_viewport_mouse_drag (visualiseur->drag_orig_x, visualiseur->drag_orig_y,
        x, y);
  }

  return TRUE;
}

static gboolean on_viewport_button_press(GtkWidget      *widget,
                                         GdkEventButton *event,
                                         Visualiseur    *visualiseur) {
  gdouble x, y;

  x = event->x;
  y = event->y;

  visualiseur->drag_orig_x = x;
  visualiseur->drag_orig_y = y;

  on_viewport_mouse_press (x, y);

  return TRUE;
}

static gboolean on_viewport_button_release(GtkWidget      *widget,
                                           GdkEventButton *event,
                                           gpointer        user_data) {
  gdouble x, y;

  x = event->x;
  y = event->y;

  on_viewport_mouse_release (x, y);

  return TRUE;
}

/*****************************************
 *
 * Fonctions de dessin
 *
 *****************************************/

void viewport_draw_point(double x, double y, gint size) {

  GdkDrawable *drawable = visualiseur.image;
  GdkGC *gc = visualiseur.gc;

  gdk_draw_arc (drawable, gc,
      TRUE, (gint)x-size/2, (gint)y-size/2,
      size, size,
      0, 64*360);

  gtk_widget_queue_draw (visualiseur.viewport);
}

void viewport_draw_line(double x1, double y1, double x2, double y2, int width) {
```

```
  GdkDrawable *drawable = visualiseur.image;
  GdkGC *gc = visualiseur.gc;

  gdk_gc_set_line_attributes (gc, width,
      GDK_LINE_SOLID, GDK_CAP_BUTT, GDK_JOIN_MITER);
  gdk_draw_line (drawable, gc,
      (gint)x1, (gint)y1, (gint)x2, (gint)y2);

  gtk_widget_queue_draw (visualiseur.viewport);
}

void viewport_draw_polygon(Point *points, int nb_points, int plein, int line_wid
th) {

  GdkDrawable *drawable = visualiseur.image;
  GdkGC *gc = visualiseur.gc;
  GdkPoint *gdk_points;
  gint i;

  gdk_gc_set_line_attributes (gc, line_width,
      GDK_LINE_SOLID, GDK_CAP_BUTT, GDK_JOIN_MITER);

  gdk_points = g_new (GdkPoint, nb_points);
  for (i=0; i<nb_points; i++) {
    gdk_points[i].x = (gint)points[i].x;
    gdk_points[i].y = (gint)points[i].y;
  }

  gdk_draw_polygon (drawable, gc, plein, gdk_points, nb_points);

  g_free (gdk_points);

  gtk_widget_queue_draw (visualiseur.viewport);

}

void viewport_change_color(double red, double green, double blue) {

  GdkGC *gc = visualiseur.gc;

  visualiseur.color.red   = CLAMP (red * 0xFFFF, 0, 0xFFFF);
  visualiseur.color.green = CLAMP (green * 0xFFFF, 0, 0xFFFF);
  visualiseur.color.blue  = CLAMP (blue * 0xFFFF, 0, 0xFFFF);
  gdk_gc_set_rgb_fg_color (gc, &visualiseur.color);
}

void viewport_clear (){
  GdkDrawable *drawable = visualiseur.image;
  GdkGC *gc = visualiseur.gc;
  GdkColor white;
  gint width, height;

  /* change color to white */
  white.red = white.green = white.blue = 0xFFFF;
  gdk_gc_set_rgb_fg_color (gc, &white);

  gdk_drawable_get_size (GDK_DRAWABLE (visualiseur.image),
      &width, &height);
  gdk_draw_rectangle  (drawable, gc,
      TRUE,
      0, 0, width, height);
```



```
  /* change back to user color */
  gdk_gc_set_rgb_fg_color (gc, &visualiseur.color);

  gtk_widget_queue_draw (visualiseur.viewport);
}

/** Procédure affichant à l'écran une particule
 * Argument:
 *      p: La particule à afficher
 */
void dessiner_particule(Particule* p){
        viewport_change_color(0,0,0);
        viewport_draw_point(p->position->x,p->position->y,TAILLE_PARTICULE);
}

/** Procédure affichant à l'écran une ancre
 * Argument:
 *      a: Ancre à afficher
 */
void dessiner_ancre(Ancre* a){
        Particule* p = a->particule;
        viewport_change_color(1,0.2,0.2);
        viewport_draw_point(p->position->x,p->position->y,TAILLE_ANCRE);
}

/** Procédure affichant à l'écran une tige
 * Argument:
 *      t: La tige à afficher
 */
void dessiner_tige(Tige* t){
        Particule* p1 = t->e1;
        Particule* p2 = t->e2;
        viewport_change_color(0.4,1,0.4);
        viewport_draw_line(p1->position->x,p1->position->y,p2->position->x,p2->p
osition->y,TAILLE_TIGE);
}

/** Procédure affichant à l'écran une chaine
 * Argument:
 *      c: La chaine à afficher
 */
void dessiner_chaine(Chaine* c){
        Particule* p1 = c->e1;
        Particule* p2 = c->e2;
        viewport_change_color(1,0.8,0.5);
        viewport_draw_line(p1->position->x,p1->position->y,p2->position->x,p2->p
osition->y,TAILLE_TIGE);
}

/** Procédure affichant à l'écran un espace
 * Argument:
 *      e: L'espace à afficher
 */
void dessiner_espace(Espace* e){
        Particule* p1 = e->e1;
        Particule* p2 = e->e2;
        viewport_change_color(0.2,0.5,1);
        viewport_draw_line(p1->position->x,p1->position->y,p2->position->x,p2->p
osition->y,TAILLE_TIGE);
}

/** Procédure affichant à l'écran un ressort
```

```
 * Argument:
 *      r: le ressort
 */
void dessiner_ressort(Ressort* r){
        Particule* p1 = r->e1;
        Particule* p2 = r->e2;
        viewport_change_color(0.2,0,0.5);
        viewport_draw_line(p1->position->x,p1->position->y,p2->position->x,p2->p
osition->y,TAILLE_RESSORT);
}

/*****************************************************************/

gboolean animation_callback(gpointer unused) {

  /* On efface tout */
  viewport_clear();

  /* Calcul des nouvelles positions */
  satisfaire_contraintes(systeme);


  /* Affichage du système */
  do_list(dessiner_tige,systeme->tiges);
  do_list(dessiner_chaine,systeme->chaines);
  do_list(dessiner_espace,systeme->espaces);
  do_list(dessiner_ressort,systeme->ressorts);
  do_list(dessiner_particule,systeme->particules);
  do_list(dessiner_ancre,systeme->ancres);

  return TRUE;
}

/*****************************************************************/

gint main(gint argc, gchar **argv) {

  GtkWidget *window;
  GtkWidget *viewport;
  GtkWidget *event_box;
  GdkPixmap *image;

  gtk_init (&argc, &argv);

  /* Réglage des locales */
  setlocale(LC_ALL, "C");

  g_print ("GTK version :%d.%d.%d\n",
      GTK_MAJOR_VERSION, GTK_MINOR_VERSION, GTK_MICRO_VERSION);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Système de particules");
  gtk_window_set_resizable (GTK_WINDOW (window), FALSE);
  g_signal_connect (G_OBJECT (window), "delete-event",
      G_CALLBACK (gtk_main_quit), NULL);

  image = gdk_pixmap_new (NULL, VIEWPORT_WIDTH, VIEWPORT_HEIGHT, 24);
  viewport = gtk_image_new_from_pixmap (image, NULL);
  gtk_misc_set_alignment (GTK_MISC (viewport), 0.0, 0.0);
  gtk_widget_set_size_request (viewport, VIEWPORT_WIDTH, VIEWPORT_HEIGHT);

  event_box = gtk_event_box_new ();
```



```
  g_signal_connect (G_OBJECT (event_box), "motion-notify-event",
      G_CALLBACK (on_viewport_motion), &visualiseur);
  g_signal_connect (G_OBJECT (event_box), "button-press-event",
      G_CALLBACK (on_viewport_button_press), &visualiseur);
  g_signal_connect (G_OBJECT (event_box), "button-release-event",
      G_CALLBACK (on_viewport_button_release), &visualiseur);

  gtk_container_add (GTK_CONTAINER (event_box), viewport);
  gtk_container_add (GTK_CONTAINER (window), event_box);

  gtk_widget_show_all (window);

  viewport_activate_drag (TRUE);
  visualiseur.viewport = viewport;
  visualiseur.image = image;
  visualiseur.gc = gdk_gc_new (image);

  viewport_clear ();

  /* --------------------*/

  /* Création du système */
  if(argc == 2){
        systeme = charger(argv[1]);
  } else {
      systeme = charger("corde.sp");
  }

  /* On vérifie qu'il n'y a pas eu de problème à la création du système */
  if(systeme == NULL) {
        return -1;
  }

  /* la fonction animation_callback est appelée toutes les 50ms */
  g_timeout_add (50, (GSourceFunc)animation_callback, NULL);

  /* --------------------*/

  gtk_main ();

  /* --------------------*/
  /* Destruction du système */
  detruire_systeme(systeme);

  return 0;
}
```