

Simulation physique d'un système de particules

Axel de Sablet

Groupe D

2 avril 2008

Résumé

Rapport de projet, sur l'implémentation en langage C d'une méthode de simulation du comportement d'un système de particules.

Table des matières

1	Présentation	2
1.1	Introduction	2
1.2	Code fourni	2
1.3	Principe	2
2	Conception	3
2.1	Organisation	3
2.2	Détails	4
2.2.1	Module Géométrie	4
2.2.2	Module Particule	8
2.2.3	Module Ancre	10
2.2.4	Module Tige	11
2.2.5	Modules Chaîne et Espacement	13
2.2.6	Module Ressort	13
2.2.7	Module Obstacle	14
2.2.8	Module Système	16
2.2.9	Module Lecture	18
2.2.10	Module Interaction	18
2.2.11	Résumé	19
2.2.12	Macros	19
2.2.13	Exécution	19
3	Tests	20
4	Conclusion	22

1 Présentation

1.1 Introduction

Le but de ce projet était la réalisation d'une simulation de modèle physique, comprenant un ensemble de points matériels liés les uns aux autres par plusieurs types de contraintes, et soumis à des forces. Les particules doivent évoluer dans un plan.

Le code joint au présent rapport propose une solution prenant en compte les forces et contraintes suivantes :

- force de pesanteur
- force de frottement
- contrainte ancre
- contrainte tige
- contrainte espacement
- contrainte chaîne
- force de rappel d'un ressort
- rencontre entre une particule et un solide

Comme demandé, le programme et l'utilisateur peuvent interagir. Premièrement, la définition du système à modéliser se fait par le biais d'un fichier texte, que l'utilisateur peut modifier comme bon lui semble. Par ailleurs, pendant l'exécution de la simulation, il lui est possible d'utiliser la souris pour modifier la position d'une particule, grâce à un cliquer-déposer.

1.2 Code fourni

Pour nous aider dans la réalisation de ce projet, quelques fichiers nous étaient fournis. Il s'agissait principalement de morceaux de codes permettant d'ouvrir une fenêtre en utilisant les bibliothèques GTK+, ainsi que de fonctions qui permettent d'effectuer des dessins simples dans la fenêtre. Le code fournissait également un moyen de détecter les actions de clic et drag-and-drop effectuées par l'utilisateur.

Par ailleurs, il nous était demandé d'utiliser correctement le module *“Liste Générique”*, déjà manipulé en TP, pour implémenter notre programme.

1.3 Principe

Détaillons la manière dont le système de particules est animé. Une tentative brutale de calcul du mouvement par la résolution d'un système d'équations différentielles, sur un nombre quelconque de points, dépasse nos compétences et probablement celles de nos machines. C'est pourquoi on se tourne plutôt vers une méthode itérative, moins précise mais sans doute bien plus efficace.

On effectuera bien les calculs de certaines forces, mais on ne déterminera pas celles dues aux liaisons tige, espacement, ou chaîne. En effet ceci tiendrait d'un problème de physique extrêmement complexe. De même, on ne considèrera pas les forces de réaction exercées sur les particules par les obstacles.

Le problème principal lors de la réalisation de ce projet est : comment obtenir une simulation respectant les contraintes, sans calculer les forces exercées par ces contraintes ? Thomas Jakobsen propose dans un article appelé *Advanced Character Physics*¹ un moyen itératif de résolution de problèmes de modélisation physique comme le nôtre.

Cette méthode repose sur deux algorithmes fondamentaux :

– **L'intégration de Verlet.**

Elle nous servira à déterminer l'évolution d'une particule en connaissant sa position précédente, sa position courante, et son accélération. En notant X_i la position d'une particule à l'instant numéro i , Δt l'intervalle de temps entre deux itérations, et \vec{a} l'accélération de la particule, l'équation de Verlet s'écrit :

$$X_{i+1} = 2X_i - X_{i-1} + \vec{a}(\Delta t)^2$$

L'accélération de la particule de masse m , \vec{a} , sera calculée d'après la somme \vec{F} des forces s'exerçant sur la particule :

$$\vec{F} = m \vec{a}$$

On a vu que toutes les forces ne pouvaient pas être calculées simplement. C'est pourquoi dans la somme des forces \vec{F} ne figureront que les termes dus à la pesanteur, aux frottements, et aux ressorts. Nous avons alors besoin de corriger l'erreur faite en laissant de côté les forces dues aux contraintes.

– **La résolution de plusieurs contraintes concurrentes par relaxation.**

Cette technique permettra de corriger les erreurs précédemment citées. Elle consiste à effectuer la résolution de chaque contrainte séparément, et à itérer ce procédé un nombre important de fois. Les itérations successives sont importantes, car sans elles, en résolvant une contrainte, on peut déplacer une particule qui était auparavant bien placée.

Le nombre d'itérations à effectuer peut varier selon la complexité du système, mais les particules finissent toujours par converger vers une position stable.

2 Conception

2.1 Organisation

La solution proposée présente une organisation en douze modules. Chaque module comporte un fichier `.c` et un en-tête `.h`. Le fichier principal, `visualiseur.c`, est le seul à ne pas faire partie d'un module, et utilise les fonctions et procédures écrites par ailleurs.

Les modules sont les suivants :

- Le module `Geometrie` comporte une structure et des fonctions de manipulation de points, comme l'obtention ou la mise à jour des coordonnées. On s'en servira également pour effectuer des opérations simples sur des vecteurs du plan, principalement des sommes de vecteurs, ou le produit d'un vecteur par un scalaire ;

¹www.teknikus.dk/tj/gdc2001.htm

- Le module `Particule` permettra la manipulation d'une particule, représentée par un point matériel. Il permettra d'effectuer, entre autres, l'application des forces sur une particule, ou l'affichage à l'écran ;
- Le module `Ancre` applique la contrainte du même nom à une particule, autrement dit s'assure que sa position ne varie pas ;
- Le module `Tige` servira à représenter une contrainte bâton entre deux particules, c'est-à-dire à maintenir constante la distance entre les deux ;
- `Chaîne` et `Espacement` sont similaires à `Tige`, excepté qu'une contrainte chaîne (resp. Espacement) maintient la distance entre deux particules inférieure (resp. supérieure) à une constante ;
- Le module `Ressort` modélise une force de rappel entre deux particules. Il permet le calcul de la force qu'exercent l'une sur l'autre deux particules liées par un ressort ;
- Le module `Système` représente le système de particule, à savoir un ensemble de particules reliées les unes autres par des contraintes, et pouvant rencontrer des obstacles. Il contient des fonctions et procédures essentielles d'affichage des éléments, ainsi que de mise à jour ;
- Le module `Lecture` effectue la création d'un système de particules à partir d'un fichier texte, comme par exemple le fichier `corde_new.sp` fourni ;
- Le module `List` est celui fourni avec le TP 9. On se servira de ses fonctions basiques sur les listes ;
- Enfin, le module `Interaction` incorpore les fonctionnalités d'interaction, grâce à la souris, entre l'utilisateur et les particules animées. Son en-tête contient accessoirement les signatures de quelques fonctions dont le code figure dans le fichier principal `visualiseur.c`.

Ces modules ont été écrits de manière à respecter autant que possible une organisation de type programmation-objet.

Remarque : Un `Makefile` permet d'automatiser sous des systèmes UNIX la compilation de tous les modules et l'édition de liens.

2.2 Détails

Rentrons dans les détails des modules. On présente les structures choisies, ainsi que les fonctions et procédures compliquées.

2.2.1 Module Géométrie

Structure On a utilisé, dans l'interface du module Géométrie, la définition suivante :

```

struct _Coordonnees {
    double x;
    double y;
};

typedef struct _Coordonnees Point;

```

Cette définition permet de manipuler un point dont les coordonnées sont de présentées par des `double`. La définition de la structure est placée dans l'en-tête du module, donc les composantes d'un point seront accessibles depuis tous les autres modules. Cette structure est la seule du programme à avoir été laissée publique.

Le module Géométrie est également utilisé pour manipuler des vecteurs du plan. Comme un vecteur est constitué de deux composantes, on utilisera la même définition. Il arrivera qu'un élément de type `Point` représente en fait un vecteur.

Fonctions Ce module contient les fonctions basiques suivantes :

```
/* Fonction de création d'un Point à partir de coordonnées */
extern Point* point_creer (double x, double y);

/* Fait la somme de deux points, considérés comme des vecteurs */
/* Le premier argument, p1, est un Point* dans lequel sera écrit
le résultat ;
le second argument est le Point à sommer à p1 */
extern void somme_points (Point* p1, Point p2) ;

/* Fait la soustraction de deux points, considérés comme
des vecteurs */
/* Le premier argument, p1, est un Point* dans lequel
sera écrit le résultat ; le second argument est le Point
à soustraire à p1 */
extern void soust_points (Point* p1, Point p2) ;

/* Fonction d'ajout des coordonnées x_ et y_ à un Point */
/* Le résultat sera écrit dans *p */
extern void ajouter_coord (double x, double y, Point* p) ;

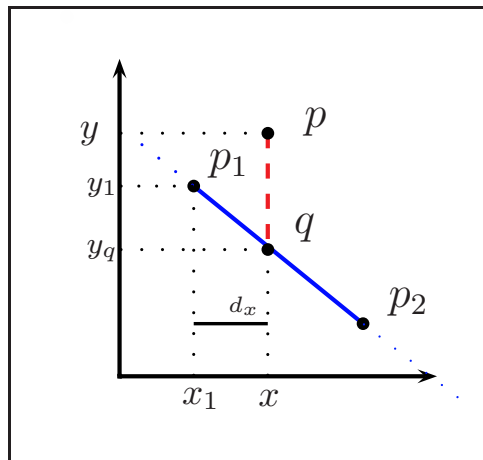
/* Multiplication d'un vecteur par un scalaire */
/* Le résultat sera écrit dans *p */
extern void mult_scalaire (double a, Point* p) ;

/* Fonction de calcul de la norme d'un vecteur */
/* Le résultat est renvoyé */
extern double norme (Point p) ;
```

Une fonction interne au module `Géométrie`, très utile lors du traitement des obstacles, est la suivante :

```
int est_au_dessus(Point p, Point p1, Point p2) ;
```

Elle détermine si un point p de coordonnées (x, y) est situé au dessus de la droite formée par deux autres points p_1 et p_2 de coordonnées respectives (x_1, y_1) et (x_2, y_2) . On peut ainsi comparer un point et une droite.



Comme le montre le dessin ci-dessus, on détermine l'ordonnée de q , le projeté de p sur la droite (p_1p_2) parallèlement à l'axe des ordonnées. Ceci se fait en posant et calculant le coefficient directeur a de la droite :

$$a = \frac{y_1 - y_2}{x_1 - x_2}$$

On a alors immédiatement $a \cdot dx = y_q - y_1$, ce qui nous donne le y_q recherché.

$$y_q = y_1 + a(x - x_1)$$

Le résultat renvoyé est vrai lorsque $y_q \leq y$.

Cas particulier : Lorsque la droite est verticale, ce raisonnement ne fonctionne plus, et on risquerait d'effectuer une division par zéro. C'est pourquoi, dans le cas où p_1 et p_2 auraient la même abscisse, on renverra simplement $(x_1 \geq x)$. Et on a encore une fonction efficace de comparaison entre un point et une droite.

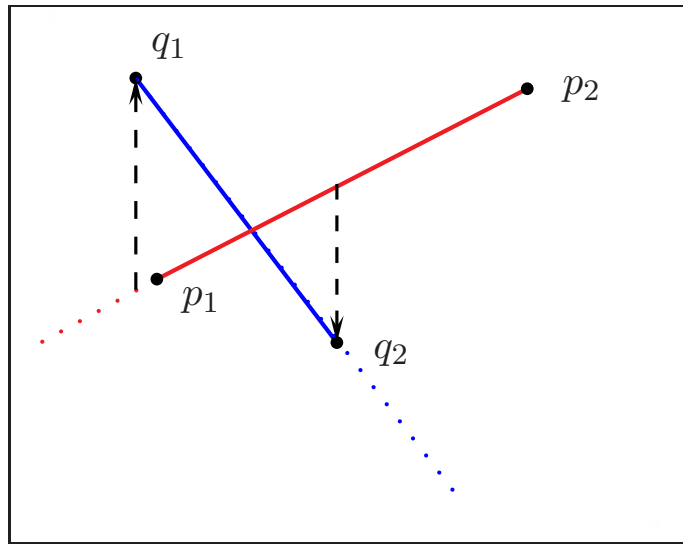
Cette fonction interne est utilisée par une autre fonction du module **Géométrie**.

```
int se_croisent (Point p1, Point p2, Point q1, Point q2) ;
```

Celle-ci détermine si les segments $[p_1p_2]$ et $[q_1q_2]$ se croisent.

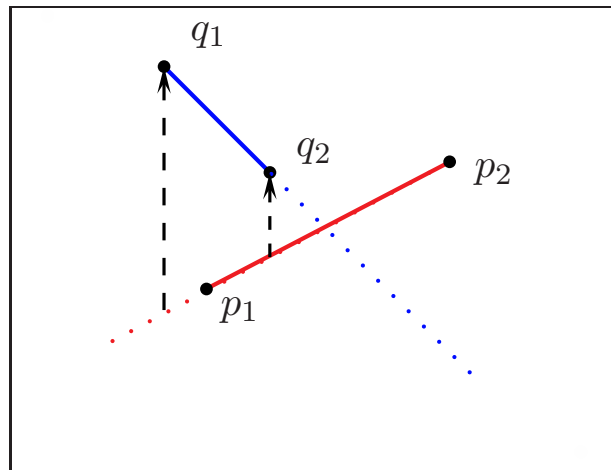
La propriété utilisée est la suivante : $[p_1p_2]$ et $[q_1q_2]$ se coupent si, et seulement si, q_1 et q_2 sont placés de part et d'autre de $[p_1p_2]$ et si p_1 et p_2 sont placés de part et d'autre de $[q_1q_2]$.

Vérifions cette propriété sur des exemples. Le dessin ci-dessous montre le cas où deux segments se coupent.



Les flèches montrent que q_1 et q_2 sont bien placés de part et d'autre de $[p_1p_2]$ (q_1 est au-dessus, et q_2 en dessous); de même, p_1 et p_2 sont placés de part et d'autre de $[q_1q_2]$. On a bien le résultat : les segments se coupent.

Voyons maintenant un exemple dans le cas contraire : les segments ne se coupent pas.



Dans ce cas-ci, p_1 et p_2 sont bien de part et d'autre de (q_1q_2) (p_1 en dessous et p_2 au-dessus) : la première condition est vérifiée. Mais q_1 et q_2 sont tous les deux au-dessus de (p_1p_2) , donc la deuxième condition n'est pas vérifiée. On a bien le bon résultat : les segments ne se coupent pas.

Le code correspondant est le suivant :

```
/* Détermine si les segments [p1 p2] et [q1 q2] se croisent */
int se_croisent (Point p1, Point p2, Point q1, Point q2) {
    return (((est_au_dessus(q1, p1, p2))
        && (! est_au_dessus(q2, p1, p2)))
        || ((est_au_dessus(q2, p1, p2))
        && (! est_au_dessus(q1, p1, p2))))

        && (((est_au_dessus(p1, q1, q2))
        && (! est_au_dessus(p2, q1, q2)))
        || ((est_au_dessus(p2, q1, q2))
        && (! est_au_dessus(p1, q1, q2)))));
}
```

2.2.2 Module Particule

Structure Dans `particule.c` est définie la structure suivante :

```
/* Définition d'une structure particule, contenant des pointeurs
   vers deux points qui représentent respectivement la position
   à l'instant courant, et la position à l'instant précédent.
   On a aussi la valeur de la masse de la particule. */
struct particule {
    Point* courant ;
    Point* prec ;
    double masse ;
} ;
```

Comme on l'avait détaillé dans la présentation, on ne stocke pas la vitesse de la particule, mais sa position à l'instant précédent, ainsi que sa position courante. Le calcul de la vitesse, une simple soustraction de vecteurs, s'effectuera à l'aide des fonctions écrites dans le module `Géométrie`. Par ailleurs, chaque particule a sa propre masse, c'est pourquoi celle-ci figure dans la structure.

Cette structure est utilisable uniquement depuis le fichier `particule.c`, c'est pourquoi on ajoute à l'en-tête du module la définition suivante :

```
typedef struct particule *Particule ;
```

De cette façon, les autres modules pourront manipuler un objet `Particule`, qui sera un pointeur vers le `struct particule` défini plus haut.

Fonctions et procédures Le module `Particule` contient les fonctions et procédures basiques suivantes :

```
/* Renvoie le point représentant la position de la Particule  
part à l'instant précédent. */  
extern Point get_prec(Particule part) ;  
  
/* Renvoie le point représentant la position courante  
de la Particule part */  
/* Utile à l'extérieur du fichier particule.c */  
extern Point get_courant(Particule part) ;  
  
/* Fonction de création d'une nouvelle particule  
à partir de ses coordonnées, et de la valeur  
de sa masse. */  
extern Particule new_particule(double x, double y,  
                                double masse) ;  
  
/* Fonction qui dessine une particule à l'écran */  
extern void afficher_particule (Particule p) ;
```

En plus des ces fonctions simples, le module `Particule` contient plusieurs fonctionnalités nécessaires pour l'animation de la simulation. En particulier :

```
/* Fonction de mise à jour d'une particule, qui prend en  
argument un pointeur vers cette Particule, et les  
nouvelles coordonnées. Cette fonction remplace également  
la position précédente par la position courante. Elle  
sera utilisée lors de l'application des forces  
sur les particules */  
extern void update_particule (Particule p, double x, double y) ;  
  
/* Fonction de changement de la position courante d'une  
particule, qui prend en argument un pointeur vers la  
Particule considérée, ainsi que le Point de destination.  
La position à l'instant précédent n'est pas modifiée.  
Cette fonction sera appelée lors  
de la résolution des contraintes. */  
extern void nouvelle_position_particule (Particule part,  
                                          Point point) ;  
  
/* Application de la formule de Verlet sur une  
Particule dont on passe un  
pointeur en argument */  
extern void verlet (Particule p) ;
```

Les deux premières sont des procédures de mise à jour des coordonnées de la particule. La première, `update_particule` utilisée après un calcul de forces, effectuera une mise à jour de la position de la particule à l’instant précédent, et remplacera la position courante par les coordonnées fournies. Pour cela, on permutera des pointeurs, ce qui est moins coûteux que si on avait dû échanger des structures entières.

En revanche, `nouvelle_position_particule` est utilisée au moment de la correction de l’erreur, c’est-à-dire la résolution des contraintes. Elle n’effectuera aucun changement sur la position précédente de la particule. En effet, une telle modification fausserait le calcul de la vitesse, et on observerait typiquement un mouvement dans lequel l’effet de l’inertie serait quasi-absent.

Une fonction `forces` effectue le calcul de la somme des forces de pesanteur et de frottement exercées sur une particule. Ces forces, respectivement appelées \vec{P} et \vec{f} , sont prises telles que :

$$\vec{P} = m \vec{g} \quad \text{et} \quad \vec{f} = -k \vec{v}$$

La procédure `verlet` modifie la particule selon le résultat donné par ce calcul de forces. L’équation du mouvement utilisée est celle citée dans la Présentation.

Une procédure `revenir` permettra, lors d’un contact avec un solide, de ramener une particule à sa position précédente.

Enfin, la fonction `est_proche` détermine si une particule est proche de certaines coordonnées. Ceci sera utile dans le module `Interaction`, pour trouver quelle particule a été cliquée. Le résultat calculé par cette fonction dépend d’un paramètre `D`, défini sous la forme d’une macro dans le fichier `constantes.h`, et qui représente la distance maximale qui pourra séparer deux particules dites proches.

2.2.3 Module Ancre

Structure Dans `ancre.c` est définie la structure suivante :

```

/* Définition d'une structure ancre, contenant un pointeur
   vers une Particule représentant la particule ancrée,
   et un Point représentant la position d'ancrage. */
struct ancre {
    Particule particule ;
    Point fixation ;
} ;

```

A l’instar de ce qui avait été fait dans le module `Particule`, cette structure est utilisable seulement depuis le fichier `ancre.c`, c’est pourquoi on ajoute dans `ancre.h` la définition :

```

typedef struct ancre *Ancre ;

```

Fonctions et procédures Ce module comprend trois fonctions basiques, ainsi qu'une procédure de résolution de contraintes.

```
/* Fonction de création d'une nouvelle Ancre, à partir d'une
   Particule déjà existante. Les coordonnées d'ancrage seront
   prises égales à la position courante de la particule. */
extern Ancre new_ancre (Particule part) ;

/* Fonction d'affichage d'une ancre. Se sert de la fonction
   afficher_particule_ancre du module particule */
extern void afficher_ancre (Ancre a) ;

/* Fonction de mise à jour de la position du
   point d'ancrage d'une Ancre. */
extern void update_ancre (Ancre a, double x, double y) ;

/* Fonction AUXiliaire de Résolution des Contraintes Ancre */
/* La résolution consiste à placer la particule
   ancrée à sa position d'ancrage. */
extern void aux_rca (Ancre a) ;
```

Comme précisé dans le commentaire, la procédure `aux_rca` est très simple. Elle remplace la position de la particule associée à l'ancre passée en argument par la position voulue, à savoir le point de fixation de l'ancre.

2.2.4 Module Tige

La composition de ce module est très similaire à celle du module `Ancre`.

Structure Dans `tige.c` est définie la structure suivante :

```
/* Définition d'un struct tige, contenant deux pointeurs
   (les deux Particules liées), ainsi que la longueur
   de cette liaison */
struct tige {
    Particule p1 ;
    Particule p2 ;
    double longueur ;
} ;
```

Dans l'en-tête, on trouve la définition suivante :

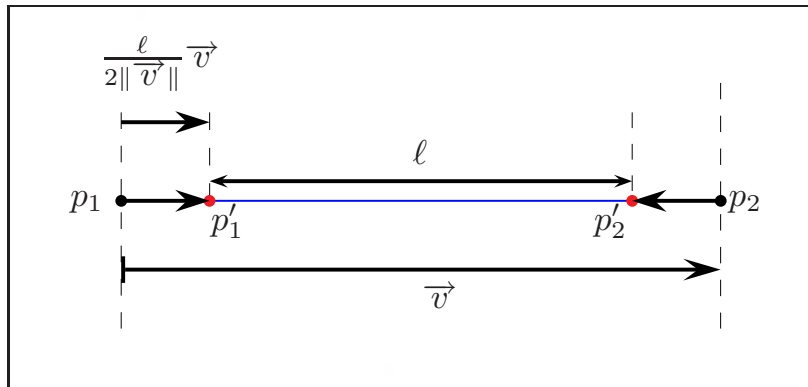
```
typedef struct tige *Tige ;
```

Fonctions et procédures Les fonctions basiques sont encore les mêmes :

```
/* Fonction de création d'une nouvelle Tige,  
à partir de deux Particules à lier. */  
extern Tige new_tige (Particule p1, Particule p2) ;  
  
/* Procédure d'affichage d'une Tige à l'écran */  
extern void afficher_tige(Tige t) ;  
  
/* Procédure réglant la longueur d'une Tige */  
/* Utile lors de la lecture du fichier */  
extern void set_longueur (Tige t, double l) ;
```

La procédure de résolution des contraintes est un peu plus complexe que dans le cas précédent. Il s'agit, dans le cas qui nous occupe maintenant, de replacer deux particules liées par une tige, de sorte que la distance entre les deux égale la longueur de la tige. On choisit de faire parcourir à chaque particule la moitié du chemin.

Remarque : on aurait également pu décider de faire bouger l'une des deux particules et de laisser l'autre à sa position actuelle. Cependant, dans de nombreux cas, le mouvement se passe mal et certaines des contraintes tige ne sont pas respectées. C'est pourquoi on ne traite pas le problème de cette façon.



Le dessin ci-dessus représente les calculs qui seront effectués pour déplacer les particules dans le but de résoudre une contrainte tige :

- Calcul de $\vec{v} = \overrightarrow{p_1 p_2}$.
- Remplacement de \vec{v} par $\frac{\ell}{2\|\vec{v}\|}\vec{v}$.
- Ajout de \vec{v} à p_1 .
- Ajout de $-\vec{v}$ à p_2 .

2.2.5 Modules Chaîne et Espacement

Ces deux modules sont en tout point similaires au module `Tige`. Les seules différences résident dans la résolution des contraintes :

- Dans le cas d’une contrainte de type `Chaîne`, on ne déplacera les particules que lorsque la contrainte n’est pas déjà respectée, c’est-à-dire lorsque la distance entre les deux particules est supérieure à la longueur caractéristique de la chaîne.
- Dans le cas d’une contrainte de type `Espacement`, on n’effectuera cette opération que lorsque la distance entre les deux particules est inférieure à la longueur de l’espacement.

On a par ailleurs veillé à utiliser des couleurs différentes pour l’affichage, dans le but de pouvoir, lors des tests, différencier immédiatement les tiges, les chaînes et les espacements.

2.2.6 Module Ressort

Structure Voici la structure utilisée pour représenter un objet ressort.

```
struct ressort {
    Particule p1 ;
    Particule p2 ;
    double raideur ;
    double longueur ;
} ;
```

Une telle structure permet de pointer vers les deux particules liées, et d’avoir en mémoire les éléments caractéristiques du ressort : sa raideur et sa longueur à vide.

Fonctions et procédures Deux fonctions ne nécessitent pas d’explication approfondie :

```
/* Fonction de création d'un nouveau Ressort ,
   à partir de deux Particules , de la longueur à vide et de la
   raideur du ressort. */
Ressort new_ressort(Particule p1, Particule p2,
                   double raideur, double longueur) ;

/* Procédure d'affichage à l'écran d'un Ressort */
void afficher_ressort(Ressort r) ;
```

Ce module contient en outre une fonction qui viendra se greffer au calcul de l’équation de Verlet. Cette fonction `aux_forces_ressort` a pour but d’effectuer le calcul de la force exercée par un ressort sur une particule. Elle sera utilisée par la fonction `forces_ressorts` du module `Système`.

Cette fonction détermine d'abord si la particule est l'une des extrémités du ressort. Dans le cas contraire, le vecteur force renvoyé est nul. Sinon, on effectue le calcul du vecteur $\vec{v} = \overrightarrow{p_1 p_2}$ (où p_1 est la particule considérée, et p_2 l'autre extrémité du ressort), que l'on multiplie par $k(l - l_0)$ pour obtenir la force recherchée.

D'où le code suivant :

```

/* Fonction qui renvoie le vecteur nul si p n'est pas une
extrémité de r, et sinon, la force exercée par le ressort
r sur la particule p */
Point* aux_forces_ressort (Particule p, Ressort r) {
    Point* v = point_creer(0., 0.) ;
    double l ;
    double l0 = r->longueur ;
    double k = r->raideur ;
    if (p == r->p1) {
        somme_points(v, get_courant(r->p2)) ;
        soust_points(v, get_courant(r->p1)) ;
        l = norme(*v) ;
        mult_scalaire(k * (l - l0) / l, v) ;
    } else if (p == r->p2) {
        somme_points(v, get_courant(r->p1)) ;
        soust_points(v, get_courant(r->p2)) ;
        l = norme(*v) ;
        mult_scalaire(k * (l - l0) / l, v) ;
    }
}

```

2.2.7 Module Obstacle

Ce module servira à représenter les solides que peuvent heurter des particules. Ces solides sont modélisés par des polygones. On choisit de dans notre implémentation de représenter un polygone par un tableau de points. Ceci permettra d'appeler facilement la fonction (fournie au préalable) d'affichage d'un polygone, qui prend elle aussi un tableau de points en argument.

A cause de la façon dont sont traitées les collisions, la variable `est_englobant` est ignorée.

Structure On choisit donc la structure suivante pour représenter un obstacle :

```

struct obstacle {
    int nb_points ;
    Point* points ;
} ;

```

Et la définition associée, dans l'en-tête du module :

```

typedef struct obstacle *Obstacle ;

```

Fonctions et procédures Ce module contient, en plus des procédures d'initialisation et d'affichage habituelles, deux fonctions utilisées lors du traitement des contraintes.

La première de ces fonctions détermine si une particule vient de traverser une arête donnée d'un polygone. Ceci revient à voir si le segment entre la position précédente et la position courante coupe l'arête en question. Ceci utilise les fonctions du module **Géométrie** et se code comme suit :

```
/* Détermine si une particule est passée d'un côté à
   l'autre d'un segment entre l'instant précédent et
   l'instant courant. */
int a_traverse_arete (Particule p, Point q1, Point q2) {
    return (se_croisent(get_courant(p), get_prec(p), q1, q2)) ;
}
```

Cette dernière fonction est utilisée pour savoir si une particule vient d'entrer dans un obstacle donné. Il suffit en effet de parcourir toutes les arêtes de l'obstacle considéré. Voici une façon de l'implémenter :

```
/* Détermine si une particule est rentrée dans un obstacle entre
   l'instant précédent et l'instant courant */
int est_rentre_obstacle (Particule p, Obstacle o) {
    int i=0 ;
    int n = o->nb_points ;
    int resultat ;

    resultat = a_traverse_arete(p, o->points[n-1], o->points[0]) ;
    while(!resultat && i < n-1) {
        resultat = a_traverse_arete(p, o->points[i], o->points[i+1]) ;
        i++ ;
    }
    return resultat ;
}
```

Remarque : Cette fonction `est_rentre_obstacle` ne sait pas faire la différence entre un passage de l'intérieur vers l'extérieur d'un obstacle, ou de l'extérieur vers l'intérieur. En effet à aucun moment on ne sait si une particule se situe dans un polygone.

Si jamais une particule est initialisée à l'intérieur d'un solide, alors elle ne pourra pas franchir les bords, et restera donc bloquée à l'intérieur.

C'est la raison pour laquelle la variable `est_englobant` a été ignorée.

2.2.8 Module Système

Abordons maintenant le module central de notre programme : le module `Système`.

Structure Un système est constitué de particules, d'ancres, d'obstacles, et de liaisons chaîne, tige, ressort, et espacement. Nous allons utiliser des listes pour représenter ces ensembles. D'où la structure suivante :

```
struct systeme {
    T_LIST particules ;
    T_LIST ancrs ;
    T_LIST tiges ;
    T_LIST chaines ;
    T_LIST espacements ;
    T_LIST obstacles ;
    T_LIST ressorts ;
} ;
```

Encore une fois, on définit dans `systeme.h` un pointeur vers un `struct systeme` :

```
typedef struct systeme *Systeme ;
```

Fonctions et procédures Le module *Système* comporte d'abord des fonctions simples, par exemple d'initialisation des listes, et d'ajout d'éléments :

```
/* Fonction de création d'un nouveau Systeme vide. */
extern Systeme new_systeme () ;

/* Procédure d'ajout d'une Particule à un Systeme */
extern void ajout_particule (Particule p, Systeme s) ;

/* Procédure d'ajout d'une Ancre à un Systeme */
extern void ajout_ancre (Ancre a, Systeme s) ;

/* Procédure d'ajout d'une Tige à un Systeme */
extern void ajout_tige (Tige t, Systeme s) ;

/* Procédure d'ajout d'une Chaine à un Systeme */
extern void ajout_chaine (Chaine c, Systeme s) ;

/* Procédure d'ajout d'un Espacement à un Systeme */
extern void ajout_espacement (Espacement e, Systeme s) ;

/* Procédure d'ajout d'un Ressort à un Systeme */
extern void ajout_ressort (Ressort r, Systeme s) ;

/* Procédure d'ajout d'un Obstacle à un Systeme */
extern void ajout_obstacle (Obstacle o, Systeme s) ;

/* Fonction d'affichage à l'écran d'un système. */
extern void afficher_systeme (Systeme s) ;

/* Procédure de mise à jour d'un système */
/* Application des forces (pesanteur,
frottement) puis appel plusieurs fois de resoudre_contraintes */
extern void update_systeme (Systeme s) ;

/* Procédure de retrait de la dernière ancre ajoutée au système.
Utile dans la partie d'interaction avec la souris. */
extern void enlever_derniere_ancre(Systeme s) ;
```

On trouve également des procédures regroupant les fonctions auxiliaires écrites dans les modules précédents, comme `resoudre_contraintes`, qui met à jour les positions de toutes les particules pour qu'elles respectent toutes les contraintes de type Tige, Ancre, Chaîne et Espacement.

On notera aussi la présence de la procédure :

```
/* Procédure de Résolution des Contraintes
   dues aux Obstacles sur un Systeme s */
void rco (Systeme s) ;
```

Cette fonction se trouve dans le module `Système`, et non dans `Obstacle`, car elle agit sur le système tout entier et non sur un objet `Obstacle` en particulier.

Elle est appelée à chaque itération de la mise à jour du système, et son but est d'empêcher les particules de rentrer dans les obstacles. Elle consiste donc en deux parcours imbriqués : pour chaque obstacle `o`, et pour chaque particule `p`, si `p` est rentrée dans le solide `o`, on la ramène à sa position précédente, qui était en dehors de l'obstacle. Les contraintes dues aux obstacles sont donc de nouveau respectées.

2.2.9 Module Lecture

Ce module consiste en une procédure de lecture d'un fichier texte, dont les spécifications sont précisées en détail dans l'énoncé, et qui définit le système à modéliser.

La fonction de lecture utilise les modules précédents, ainsi que plusieurs fonctions utiles du langage C, en particulier `fgets`, `sscanf`, et `strtok`. La lecture consiste en plusieurs étapes, commentées dans le code.

Pour résumer, après avoir lu les constantes du problème (valeurs de la gravité et des frottements), pour chaque type d'objet à ajouter au système, on crée un tableau de la bonne taille, que l'on remplit des objets considérés, avant d'ajouter tous ces éléments au système.

On n'oublie pas, une fois la lecture terminée, de libérer l'espace utilisé par les tableaux.

2.2.10 Module Interaction

Ce module permet à l'utilisateur de cliquer sur une particule non ancrée, et de la déplacer en effectuant un cliquer-glisser.

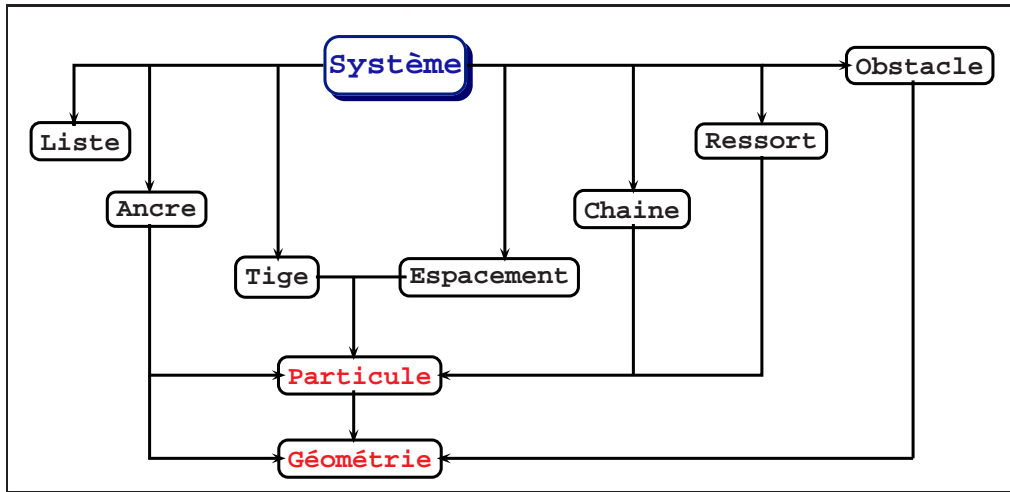
On utilisera une `Particule p` en variable globale. Elle vaut initialement `NULL`.

Le principe est, lorsqu'un clic de souris est effectué :

- Recherche d'une particule proche. Si la recherche donne un résultat, on fait pointer `p` vers la particule trouvée. Sinon, `p` reste à `NULL`.
- Création et ajout au système d'une ancre, placée sur la particule pointée par `p` et fixée à la position du clic.
- Pendant le cliquer-glisser, on vérifie qu'une particule avant été trouvée à la première étape. Si `p` est `NULL`, c'est un cliquer-glisser dans le vide, on ne fait rien. Sinon, déplacement de la dernière ancre créée (c'est-à-dire celle générée lors du clic de souris), vers la nouvelle position de la souris.
- Lorsque la souris est relâchée, suppression de l'ancre précédemment créée.

2.2.11 Résumé

Le schéma ci-dessous résume l'organisation des modules utilisés pour constituer un système de particules.



2.2.12 Macros

Certaines constantes du programme sont, comme on l'a évoqué plus haut, définies par des macros. Ces macros sont définies dans le fichier `constantes.h`.

Il s'agit du nombre d'itérations à effectuer lors de la résolution des contraintes, mais aussi de la distance entre deux positions considérées comme proches, la longueur maximale d'une ligne du fichier à lire, et également un coefficient multiplicatif sur la vitesse d'une particule qui peut être utilisé pour obtenir lors du rendu des mouvements de particules plus fluides.

2.2.13 Exécution

L'exécution du programme se déroulera comme suit :

- Ouverture d'une fenêtre grâce aux bibliothèques GTK
- Lecture d'un fichier, définition du système. Le fichier utilisé par défaut est `corde2.sp`, mais un autre peut être spécifié, en passant son nom en argument de la ligne de commande.
- Tant que la fenêtre est ouverte, exécution toutes les 50 millisecondes des instructions : nettoyage de l'écran, affichage du système puis mise à jour du système.

Remarque 1 : Le programme comporte quelques variables globales. Il s'agit des éléments utilisés par tous les fichiers, à savoir le système modélisé, ainsi que les constantes de gravité et de frottement choisies. C'est pourquoi on trouve au début de plusieurs modules une ou plusieurs des déclarations suivantes :

```
/* Déclaration des variables globales ,  
utilisées dans tout le programme. Ces variables sont  
définies lors de la lecture du fichier. Il s'agit du  
Système, des valeurs de l'accélération de la pesanteur  
(selon les x et selon les y), ainsi que du  
coefficient de frottement. */  
extern Systeme s ;  
extern double gx ;  
extern double gy ;  
extern double k ;
```

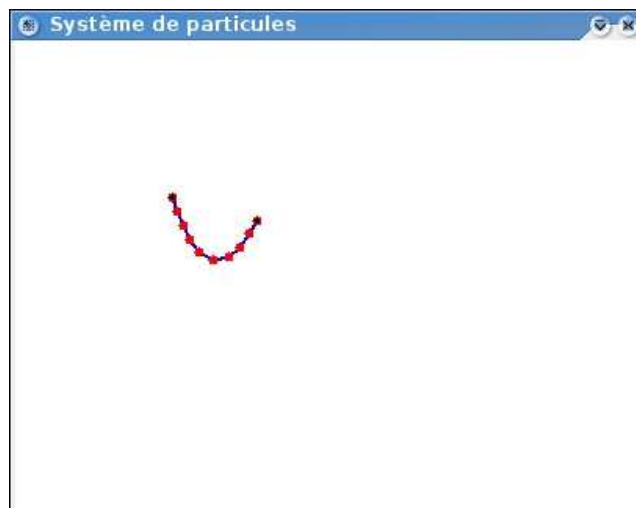
Remarque 2 : On a pris soin, lors de la rédaction du code de ce programme, de ne pas conserver en mémoire des éléments qui ne seront plus utilisés par la suite. En effet, de nouveaux objets `Point` sont créés à chaque itération, et pourraient, s'ils n'étaient pas désalloués après utilisation, entraîner une grande consommation de mémoire vive.

3 Tests

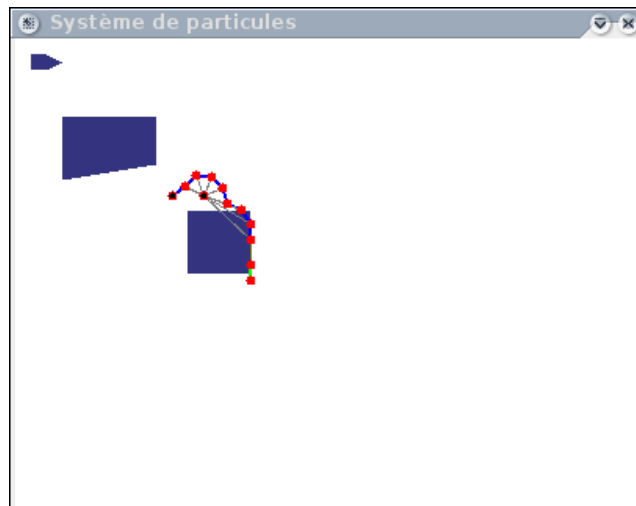
On a utilisé, pour les tests, plusieurs fichiers de configurations du système. Retenon-en un pour exemple, le plus complet. Ce fichier a été rendu avec les sources du programme, il est nommé `corde2.sp`. Il permet de représenter une corde ancrée constituées de tiges, d'un ressort et d'une chaîne; certaines des particules de la corde sont liées à une seconde ancre par des espacements; enfin, trois solides sont placés dans le plan.

Voici une capture d'écran faite en utilisant le fichier de configuration fourni, `corde_new.sp`. On voit deux ancres, celle définie par le fichier, et une seconde placée à la souris.

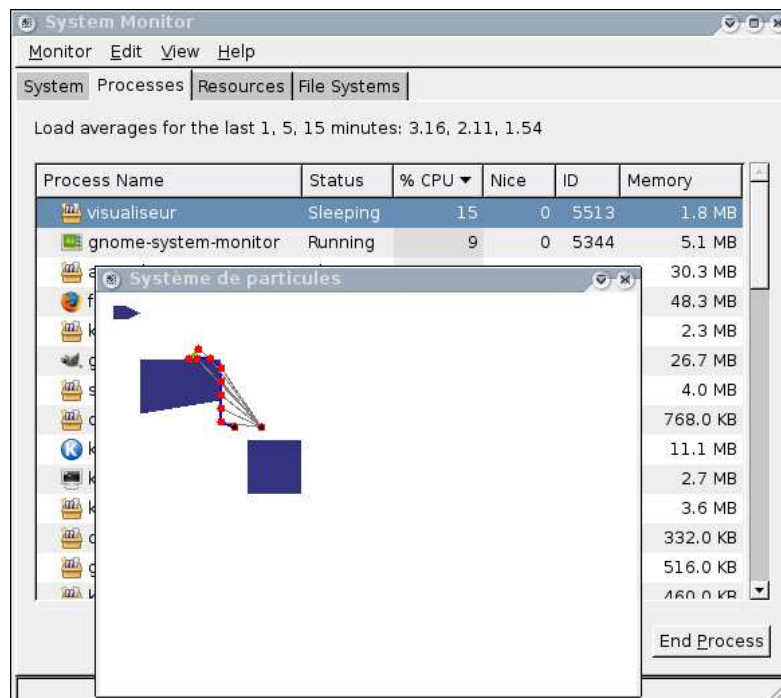
Remarque : la souris n'est pas visible, elle a été retirée par le logiciel de capture d'écran.



Voici maintenant ce que donne le fichier corde2.sp.



On peut examiner la consommation de ressources du processeur et de la mémoire vive du programme, après l'avoir laissé tourner quelques minutes.



Nous utilisons 15% d'un processeur AMD Athlon, et environ 2 Mo de mémoire vive, ce qui n'est pas excessif compte tenu du nombre relativement important de calculs à effectuer.

4 Conclusion

Ce projet est globalement concluant. Quelques points pourraient avec un peu de travail être améliorés. En particulier, il peut être difficile de trouver l'équilibre entre les frottements et les forces de rappel des ressorts, pour obtenir un système à la fois stable et suffisamment mobile...

On aurait par ailleurs pu tenter d'ajouter du rebond lors d'un contact entre une particule et un obstacle. Ceci n'a pas été fait, par manque de temps.

Néanmoins, le programme rendu est fonctionnel et, mis à part les quelques points de détail évoqués au-dessus, respecte la quasi-totalité des contraintes imposées par l'énoncé.

On aura pu, lors de l'élaboration et de l'écriture de ce projet, découvrir des bibliothèques utiles, bien que moyennement portables, du langage C : les bibliothèques graphiques GTK.

Nous avons également eu l'occasion de mettre en pratique ce qui nous a été enseigné en TD et TP, notamment sur la programmation modulaire.