Ce document montre de manière très progressive comment étendre NanoÉdit. Une méthode plus directe consiste en modifier des objets existants. Il ne peut être compris que si NanoÉdit a été utilisé intéractivement et que le cours d'aprentissage à son utilisation a été suivie.

Par abus de langage, les noeuds des formules sont appelé objets.

1 Architecture

Le répertoire de base est `/home/exco/NE/1.11` dans lequel on trouve :

- `UTILITAIRES` des utilitaires de bases indépendant de NanoÉdit.
- `COEUR` le coeur de NanoEdit.
- `OBJETS` un répertoire d'objets.
- `TYPES` un répertoire de types.
- `TESTS` un répertoire contenant des tests de fonctionnement.
- `ARTICLES` documentations diverses.
- `EXEMPLES` qui contient les fichiers présentés dans ce document.
- `Nanoedit.ad` des ressources X11 de base vouées à disparaître.
- `AFAIRE`, une belle liste de choses.

Chaque répertoire (notemment `/home/exco/NE/1.11/OBJETS`) permettant d'étendre NanoÉdit a la structure suivante :

- `Makefile` une copie de `/home/exco/NE/1.11/Makefile.proto`
- Des fichiers commençant par `o_` pour définir des classes.
- Des fichiers commençant par `t_` pour définir des types.
- `MODELES` qui contient des modèles.
- `Nanoedit.ad` des ressources X11 pour les objets locaux.

2 Initialisations

Pour créer un nouveau répertoire de travail et lancer NanoEdit : `mkdir NE`

```
cd NE
mkdir 1.11
cd 1.11
cp /home/exco/NE/1.11/Makefile.proto Makefile
make # Compile/Edition des liens/Lancement
# make lance # Lance
# make gdb # Lance sous le debuggueur
# make depend # Calcule automatiquement les dépendances dans Makefile
```

Si la commande `make` ne fonctionne pas, lancez `/usr/local/bin/make`.

Si vous désirez utilisez des objets créés par d'autre personnes éditer le `Makefile` pour modifier la liste des répertoires où il faut chercher des objets (pour piquer ceux de vos copains). Vous pouvez par exemple ajouter le repertoire `/home/exco/NE/1.11/EXEMPLES` de NanoÉdit afin de profiter de tous les exemples qui suivent sans avoir à les recopier. Dans le menu «Charge/EXEMPLES» il y a des exemples d'utilisation de tous les objets qui suivent.

3 Ajout progressif de l'objet segment de droite

On va présenter pas à pas l'ajout d'un objet segment de droite défini par deux points. Seul la version finale est complètement fonctionnelle, les versions intermédiaires sont incomplètes mais néanmoins fonctionnelle.

3.1 Ajouter un objet minimal

Le programme le plus simple consiste à ajouter à votre répertoire un fichier commençant par “o_” qui contient la définition de la classe (“o_s1.c”).

```
#include "o_objet.h"        /* Car il faut définir un objet */

struct segment              /* Votre structure de donnée */
{
};

CLASSE(segment1,            /* Nom de la classe */
       struct segment,      /* La structure de donnée liée à l'objet */
      )
```

Une fois ce fichier entré vous pouvez taper `make` pour lancer la compilation et l'exécution.

Dans le menu «création» vous trouvez «NON DÉFINI/segment1» qui vous permet de créer une instance de l'objet, celle-ci apparait dans la fenêtre contenant le graphe du modèle.

3.2 Chemin dans le menu, raccourci clavier et aide

Ces différentes choses ont été ajoutées dans l'objet suivant (“o_s2.c”) :

```
#include "o_objet.h"
struct segment
{
};
CLASSE(segment2,
       struct segment,
       /*
        * ATTENTION: La suite ne contient pas de séparateur , ou ;
        */
       EVENEMENT("S2")  /* Raccourci clavier */
       MENU("Exemples/Segment2") /* Chemin dans le menu */
       BULLE_D_AIDE         /* Aide sur l'objet */
       ("Objet avec raccourci et menu\n")
      )
```

- On indique le raccourci dans cet exemple par :
  `EVENEMENT("S2")`
  ceci permet de créer l'objet simplement en tapant “s2”. Le raccourci n'a pas de longueur limite. Si l'on veut indiquer des “modificateurs”, il suffit de précéder le raccourci par l'une des chaines suivantes : ”Shft+”, ”Ctrl+”, ”ShftCtrl+”, ”Meta+”, ”ShftMeta+”, ”CtrlMeta+”, ”ShftCtrlMeta+”.
- Le chemin dans le menu est explicite par lui-même :
  `MENU("Exemples/Segment2")`
- L'aide aussi :
  `BULLE_D_AIDE("Plus petit objet que l'on peut faire avec NanoÉdit")`
  Pour tester l'aide, vous éditer l'objet et cliquez avec le bouton de droite dans la fenêtre.

3.3 Le segment de droite possède deux champs

Voici comment ajouter des champs à l'objet segment de droite :

- On utilise la macro “CHAMP” pour ajouter des champs à une classe, le premier argument est le nom du champ dans la structure C, le deuxième est un programme interprété définissant le champ.

- “t_geometrie.h” : NanoÉdit contient des types géométriques prédéfinis dans “TYPES/t_geometrie.h”, “Triplet” est un tableau de 3 “Flottant” et “Flottant” est en fait un “double”. Mais en fait cela n’a aucune importance, vous pouvez utilisez les types que vous voulez, il est facile d’indiquer à NanoÉdit comment les manipuler.
- “L_point” indique le type logique du champ. Le champs représente un point de R3, ce point est stocké dans un “Triplet” (type “P_triplet”). Si vous voulez stocker ce point dans une chaine de caractère (`char*`) par exemple, vous pouvez mettre “P_chaine” au lieu de “P_triplet”. La liste actuelle des types logiques et physiques est à la fin de ce document.
- “Edite” indique que l’utilisateur peut modifier la valeur du champs intéractivement.
- “Sauve” indique que la valeur du champ est sauvegardée dans le fichier modèle et sera récupérée lors d’un chargement.

L’objet suivant possède donc deux champs éditables textuellement (“o_s3.c”).

```
#include "o_objet.h"
#include "t_geometrie.h"  /* Pour Triplet et L_point3 */

struct segment            /* Les deux points */
{                         /* On trouve les types existants dans */
  Triplet depart;         /* TYPES/t_base.h TYPES/t_geometrie.h */
  Triplet arrivee;
};

CLASSE(segment3,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve) /* Départ */
       CHAMP(arrivee, L_point P_triplet Edite Sauve) /* Arrivée */
      )
```

3.4 Améliorations au contenu des champs

Elles sont nombreuses et triviales.

- Si l’on indique une bulle d’aide dans la déclaration du champ, elle s’applique au champ lui-même.
- Si l’on indique “Init3d” dans les champs, on pourra alors modifier intéractivement la valeur du champs en
  - Activant l’objet segment
  - Positionnant la souris dans une fenêtre 2D.
  - Tapant 1 pour modifier la valeur du premier champ et 2 pour modifier celle du deuxième.

  Du plus, lors de la création du segment, la valeur du premier champ sera initialisée à la position du curseur.
- Si l’on indique “DEFAUT”, le champ sera initialisé à la valeur indiquée par la chaîne de caractère.
- Dans la fenêtre d’édition, la valeur du champ est précédée d’un *label* (texte explicatif), par défaut, le texte du label est le nom du champ dans la structure C. Avec “LABEL” on peut modifier ceci.

“o_s4.c” :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment
{
  Triplet depart;
  Triplet arrivee;
```

```
};

CLASSE(segment4,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d
             LABEL("Point de départ")
             BULLE_D_AIDE("Départ du segment"))
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d
             LABEL("Point d'arrivée")
             DEFAUT("1 2 3")
             BULLE_D_AIDE("Arrivée du segment"))
      )
```

3.5 Extraction du point de départ et d'arrivé

Pour le moment, le segment de droite n'est pas paramétré. On ne peut pas définir un segment de droite à partir de deux points existants. Pour le rendre paramétré, il suffit d'ajouter "Extrait" dans les champs que l'on souhaite extraire des arguments du noeuds. Le premier champs extrait le sera du premier paramètre, le deuxième du deuxième, ...

Le fichier contient la définition de deux classes, la première veut deux paramètres de type "L_point3" (Le type "L_point3" indique le type "P_triplet" par défaut) et ne pourra donc pas être paramétrée par un "L_point4". La deuxième veut des paramètres de type "L_point" (donc 3 ou 4) et les stocke sous forme de triplet dans la structure C ("o_s5.c").

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment
{
  Triplet depart;
  Triplet arrivee;
};

CLASSE(segment5,
       struct segment,
       CHAMP(depart , L_point3 Edite Sauve Init3d /**/ Extrait /**/)
       CHAMP(arrivee, L_point3 Edite Sauve Init3d /**/ Extrait /**/)
      )
CLASSE(segment5_,
       struct segment,
       CHAMP(depart , /**/ L_point P_triplet /**/ Edite Sauve Init3d Extrait)
       CHAMP(arrivee, /**/ L_point P_triplet /**/ Edite Sauve Init3d Extrait)
      )
```

Que le champ soit extrait ou non, il est stocké dans les enregistrements de la structure C. Si les paramètres changent, le changement est reporté de manière transparante dans la structure C.

Si l'on désire que le segment de droite soit **obligatoirement** paramétré par deux points, on ajoute "Obligatoire" en plus de "Extrait"

3.6 Affichage du segment sur l'écran

Il faut ajouter un champ particulier qui indique la méthode d'affichage, l'affichage écran doit être écrit en utilisant la bibliothèque Open GL. Il est aussi possible de définir la méthode d'affichage en POV. "o_s6.c" :

```
#include "o_objet.h"
#include "t_geometrie.h"
#include <GL/gl.h>                                /**/
struct segment { Triplet depart, arrivee; };
static void affiche_segment(struct segment *s,     /**/
```

```
                        const Champ *ch,              /**/
                        const char *qualite)          /**/
{                                                     /**/
glBegin(GL_LINES) ;                                   /**/
glVertex3f(s-> depart.x, s-> depart.y, s-> depart.z) ;    /**/
glVertex3f(s->arrivee.x, s->arrivee.y, s->arrivee.z) ;    /**/
glEnd();                                              /**/
}                                                     /**/
CLASSE(segment6,
     struct segment,
     CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
     CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
     CHAMP_VIRTUEL(L_affiche_gl(affiche_segment))        /**/
    )
```

Par défaut, l'objet «vue» de NanoÉdit change automatiquement l'épaisseur des traits, la taille des points et le type d'ombrage suivant que l'objet est actif ou non. Si vous voulez afifcher l'objet différemment suivant qu'il est actif ou non, vous pouvez tester si l'objet est actif en testant la valeur de `ACTIF(votre_objet)`.

Si l'on veut faire un tracé dépendant de la qualité du rendu demandé dans la vue, il faut tester la valeur de `qualite[0]` qui est un caractère entre '0' et '9', '5' indiquant la qualité "normale".

Pour accélérer l'affichage, l'objet «vue» de NanoÉdit stocke systématique les ordres d'affichage dans une liste OpenGL. Pour définir plus précisemment si l'on doit ou non stocker dans une liste OpenGL on peut ajouter des options d'affichage dans le champ contenant `L_affiche_gl`. On peut faire un ou binaire entre les options d'affichage :

- `AFFICHAGE(Toujours_different)`
  Chaque fois que la fonction d'affichage est appelée, elle affiche quelque chose de différent, il ne faut donc pas utiliser de liste OpenGL. C'est aussi utile si l'objet à afficher est très gros et qu'il faut trop de mémoire pour faire sa liste.
- `AFFICHAGE(Toujours_identique)`
  Quelque soit l'instance de la classe, l'objet à afficher est identique. Par exemple la sphère unité.
- `AFFICHAGE(Different_si_actif)` L'objet est affiché différemment suivant qu'il est actif ou non, on prévois donc deux listes OpenGL différentes pour les deux cas.
- `AFFICHAGE(Different_suivant_qualite)` L'objet est affiché différemment suivant qu'il est actif ou non, on prévois donc 10 listes différentes suivant la qualité.
- `AFFICHAGE(Different_suivant_type_affichage)` L'objet est affiché différemment suivant que l'état courant est ponctuel/filaire ou facette. Cela multiplie donc par 3 le nombre de listes OpenGL.

3.7 Affichage d'un "résumé" de l'objet

Dans la fenêtre contenant le graphe du modèle, le texte entre parenthèse est par défaut le contenu du premier champ de l'objet avec l'attribut «Init3d». Ce texte apparait aussi dans le menu listant les objets actifs.

On peut calculer ce texte soit même en fonction de l'objet. Dans le cas du segment ("o_s7.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee ; } ;
static const char *informations_segment(const struct segment *s)/**/
{                                                         /**/
  static char info[1000] ;                                /**/
  sprintf(info, "%.3g,%.3g,%.3g ---> %.3g,%.3g,%.3g",         /**/
          s->depart.x, s->depart.y, s->depart.z,          /**/
          s->arrivee.x, s->arrivee.y, s->arrivee.z) ;     /**/
  return(info) ;                                          /**/
```

```
}                                                                /**/
CLASSE(segment7,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       INFORMATIONS(informations_segment)                        /**/
      )
```

3.8 Le segment doit calculer son vecteur directeur

Si certain champs d'un objet sont calculés (en fonction d'autres champs ou de paramètres), il faut que l'objet soit prévenu lors des changements afin qu'il se mette à jour.

Le programme utilise des fonctions utilitaires sur les "Triplet" qui sont définies dans "UTILITAIRES/u_geometrie.h" ("o_s8.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee, vecteur ; } ; /* Ajoute vecteur */
static void changement_segment(struct segment *s)                /**/
{                                                                /**/
  if ( CHAMP_CHANGE(s, depart) || CHAMP_CHANGE(s,arrivee) ) /**/
    s->vecteur = u_soustrait_triplet(&s->arrivee, &s->depart) ; /**/
}                                                                /**/
CLASSE(segment8,
       struct segment,
       CHAMP(depart , L_point   P_triplet Edite   Sauve Init3d Extrait)
       CHAMP(arrivee, L_point   P_triplet Edite   Sauve Init3d Extrait)
       CHAMP(vecteur, L_vecteur P_triplet Affiche)               /**/
       CHANGEMENT(changement_segment)                            /**/
      )
```

Vous remarquerez que le vecteur n'est ni sauvegardé ni éditable car cela n'a pas de sens car on le calcule, par contre, il est affichable et d'autres objets peuvent utiliser sa valeur en l'extrayant.

Si la valeur est impossible à calculer, alors il faut noter que le champ résultat est invalide. Dans la fonction changement, on indique : "NOTE_CHAMP_INVALIDE(objet, resultat)"

3.9 Ajouter un champ milieu virtuel

On peut vouloir afficher la valeur du milieu du segment sans toutefois l'ajouter dans la structure C, de plus on aimerais qu'une modification du milieu entraine un translation des deux points du segment.

Pour cela il faut écrire une méthode permettant de calculer le milieu et une autre permettant de la modifier. Ceci est facile ("o_s9.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee ; } ;

static void ecrit_milieu(struct segment *s, const Champ *c, const char *v)
{
  Triplet ancien_milieu, nouveau_milieu, vecteur ;

  if ( CHAMP_CHANGE(s,depart) || CHAMP_CHANGE(s, arrivee) )
    return ; /* On change le milieu seulement s'il est le seul à changer */

  ancien_milieu  = u_milieu_2_triplet(&s->depart, &s->arrivee) ;
  nouveau_milieu = char_vers_triplet(v) ;
  vecteur        = u_soustrait_triplet(&nouveau_milieu, &ancien_milieu) ;
  s->depart      = u_ajoute_triplet(&s->depart, &vecteur) ;
```

```
  s->arrivee     = u_ajoute_triplet(&s->arrivee, &vecteur) ;
}

static const char* lit_milieu(struct segment *s, const Champ *c)
{
  Triplet milieu ;

  milieu = u_milieu_2_triplet(&s->depart, &s->arrivee) ;

  return( triplet_vers_char(&milieu) ) ;
}

CLASSE(segment9,
       struct segment,
       CHAMP(depart , L_point P_triplet    Edite   Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet    Edite   Sauve Init3d Extrait)
       CHAMP_VIRTUEL(L_fonction2(ecrit_milieu, lit_milieu) Init3d) /**/
      )
CLASSE(segment9_,
       struct segment,
       CHAMP(depart , L_point P_triplet    Edite   Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet    Edite   Sauve Init3d Extrait)
       CHAMP_VIRTUEL(LABEL("Milieu") Edite Init3d
                    L_point P_virtuel(ecrit_milieu, lit_milieu))
      )
```

Pour la première classe le champ milieu n'apparait pas dans la fenêtre d'édition mais on peut modifier le milieu intéractivement en activant le segment et en tapant '3' car c'est le troisième champ initialisable.

Dans la deuxième classe, on peut l'utiliser comme un champ normal car il est typé comme un point et il est éditable.

3.10 Les "tags" de base

Un "tag" est un "menu" graphique contextuel. Dans le cas du segment de droite, on a envie de voir apparaître des carrés aux extrémités du segment lorsque celui est actif afin que l'on puisse "tirer" sur ces carrés pour déplacer l'un ou l'autre des points extrémités. L'utilisation basique des "tags" est facile, il suffit de modifier la fin de la fonction d'affichage.

Le premier paramètre de la fonction affichant le tag indique le numéro du champ associé au tag, on peut tout à fait associer un tag à un champ virtuel (par exemple le milieu du segment de droite).

Voici "o_s10.c" qui affiche un tag pour le point de départ et d'arrivé :

```
#include "o_objet.h"
#include "t_geometrie.h"
#include "u_graphique.h" /* Pour t_tag_origine */
#include <GL/gl.h>
struct segment {  Triplet depart, arrivee ; } ;
static void affiche_segment(struct segment *s, const Champ *ch, const char *q)
{
glBegin(GL_LINES) ;
glVertex3f(s-> depart.x, s-> depart.y, s-> depart.z) ;
glVertex3f(s->arrivee.x, s->arrivee.y, s->arrivee.z) ;
glEnd();
if ( ACTIF(s) )                       /**/
    {                                 /**/
      t_tag_origine(0, &s->depart  ) ; /**/
      t_tag_origine(1, &s->arrivee) ; /**/
    }                                 /**/
}
CLASSE(segment10,
```

```
    struct segment,
    CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
    CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
    CHAMP_VIRTUEL(L_affiche_gl(affiche_segment))
   )
```

Remarque : lorsque l'on clique avec le bouton de droite sur un objet, on a la liste de tous les champs de l'objet, si de plus on clique avec le bouton de droite sur un "tag" on a la valeur du champ associé au "tag" (ça marche aussi pour le milieu du segment de droite si le champ virtuel a été ajouté).

Pour varier l'affichage des "tags", il y a "t_tag", "t_tag_dilation", "t_tag_rotation", "t_tag_destruction", "t_tag_poids" pour avoir des couleurs différentes suivant la sémantique du "tag".

3.11 Les sous-objets activables

Dans certain cas, on veut pouvoir désigner et travailler sur des sous-objets, par exemple, l'objet "nuage de point" permet d'activer individuellement des points du nuage. Pour faire cela, il suffit de prévoir dans la structure de donnée la place pour stocker les numéros des sous-objets actifs et l'indiquer par un champ spécial. Lors de l'affichage, les sous-objets actifs seront visualisés de manière particulière afin de donner à chacun un numéro différent (`glPushName/glLoadName`).

Voici l'exemple du segment de droite, mais cette fois il est divisé en 10 sous-segments. Une fois le segment actif, on peut sous-(in)activer des dixièmes de segment avec le bouton du milieu ("o_s21.c").

```
#include "o_objet.h"
#include "t_geometrie.h"
#include "u_graphique.h"
#include <GL/gl.h>
#define NB 10                /* Découpage du segment en NB parties */
#define NUMERO_TAG 1000
struct segment
  {
    Triplet depart, arrivee;
    Liste *sous_actifs;      /* L'objet doit prévoir le stockage */
  };
/* Calcul du point milieu de la partie "i" */
static Triplet point(Triplet *a, Triplet *b, float i) /* Utilitaire */
{
  i /= NB;
  return( u_ponderation_2_triplet(a, i,   b, 1-i) );
}
static void affiche_segment(struct segment *s)
{
  int i;
  Triplet pt;
  glPushName(0);             /* Il faut nommer les sous-objets */
  for(i=0;i<NB;i++)
    {
      glLoadName(NUMERO_TAG+i); /* Donne un nom aux graphismes suivants */
      glBegin(GL_LINES);    /* On doit mettre cette ligne après LoadName */
      pt = point( &s->depart, &s->arrivee, i+.1);
      t_triplet(&pt);
      pt = point( &s->depart, &s->arrivee, i+1-.1);
      t_triplet(&pt);
      glEnd();
    }
  glPopName();               /* Enlève ce que l'on a empilé */

  /* La boucle POUR permet de parcourir la liste des numéros de sous-actif */
  if ( ACTIF(s) ) /* Pour ne pas surcharger l'affichage */
    POUR(int, sous_objet_actif, s->sous_actifs,
```

```
        {
          /* On affiche un tag au milieu de chaque sous-segment actif */
          pt = point(&s->depart,&s->arrivee,sous_objet_actif-NUMERO_TAG+.5);
          t_tag(sous_objet_actif, &pt);
        });
}
CLASSE(segment21,
       struct segment,
       Edite Sauve
       CHAMP(depart     , L_point P_triplet        Init3d Extrait)
       CHAMP(arrivee    , L_point P_triplet        Init3d Extrait)
       CHAMP(sous_actifs, L_sous_actif                           ) /**/
       CHAMP_VIRTUEL(     L_affiche_gl(affiche_segment)          )
      )
```

3.12 Ajout du concept de bipoint

Les types physiques et logiques pour Bipoint existent déjà de base, c'est pour cela qu'ils sont postfixés par '2' dans l'exemple pour ne pas faire de conflits.

L'ajout de type logique est simple, néanmoins il faut être certain que vous avez réellement besoin d'un nouveau type logique. C'est-à-dire que cela ne recouvre pas un type existant.

L'ajout du type physique peut être fait de plusieur manières, il y en a déjà beaucoup de défini et il sont très simple à ajouter.

On stocke généralement les types (logique, physique ou d'éditeur) dans un fichier préfixé par "t_".

"t_segment.c" :

```
#include "o_objet.h"
#include "t_segment.h"

DEFINITION_TYPE_LOGIQUE(L_bipoint2, /* Le nom du type logique */
                        L_valeur, /* La surclasse du type logique */
                        "Bipoint", /* Aide sur le type logique */
                        /* On indique la hauteur du Widget (Facultatif) */
                        c->te_hauteur = 40; /* car 2 lignes de texte */
                        )
TYPE_PHYSIQUE(codec_bitriplet2, /* Le nom du type physique */
              Bitriplet2,        /* La structure C associée au type physique */
              /* Ce qui suit est une fonction C */
              {
                codec_triplet((t_l3_double*)&a->t1); /* Contient P */
                codec_triplet((t_l3_double*)&a->t2); /* et Q */
              }
              )
```

"t_segment.h" :

```
#ifndef T_SEGMENT_H
#define T_SEGMENT_H

#include "t_geometrie.h"

PROTOTYPE_TYPE_LOGIQUE(L_bipoint2) /* Création de prototypes nécessaires */
#define L_bipoint2 TL(L_bipoint2)  /* Pour simplifier les définitions */

typedef struct
{
  Triplet t1, t2;
} Bitriplet2;                /* Le stockage Bi-Triplet  */

PROTOTYPE_TYPE_PHYSIQUE(codec_bitriplet2, Bitriplet2) /* Prototypes */
```

```
#define P_bitriplet2 TP(codec_bitriplet2) /* Pour simplifier les définitions */

#endif
```

"o_s11.c" :

```
#include "o_objet.h"
#include "t_segment.h"

struct segment { Bitriplet2 pq ; } ; /* Le segment contient un bipoint */
CLASSE(segment11,
       struct segment,
       /*
        * Ce champ indique à NanoÉdit que l'objet contient un bipoint
        */
       CHAMP(pq   , L_bipoint2 P_bitriplet2)
       /*
        * Les champs habituels qui cette fois sont les deux
        * points contenus dans le bipoint
        */
       CHAMP(pq.t1 , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(pq.t2 , L_point P_triplet Edite Sauve Init3d Extrait)
      )
```

Avec ce nouvelle objet, on peut paramétrer d'autre objets directement par une valeur de type "bipoint". Pour vérifier immédiatement si cela fonctionne, vous pouvez utiliser un navigateur WWW pour voir le contenu de tous les champs de l'objet segment.

Par la suite, l'objet longueur de bipoint extraira la valeur du champ bipoint de l'objet.

3.13 Les "tags" virtuels

Les tags virtuels sont ici, car l'exemple donné nécessite la connaissance préalable des types physiques.

On peut vouloir changer intéractivement une valeur associée à un sous-objet, pour cela on peut utiliser des tags virtuels (par exemple les points d'une "table de point" sont déplaçable individuellement).

Les tags virtuel ne sont associé à aucun champ.

Dans le cas de notre segment, l'utilisation des tags virtuels permet de changer interactivement la couleur de chaque dixième de segment ("o_s22.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
#include "u_graphique.h"
#include <GL/gl.h>
#define NB 10
#define NUMERO_TAG 1000 /* Pour éviter les conflits avec les # de champs */
typedef float CouleurNB[NB][3] ; /* Le type de ma table de couleur */
struct segment
   {
     Triplet depart, arrivee ;
     CouleurNB couleur ;        /* Couleur des sous-objets */
   } ;
/* Un type physique est créé car c'est une table
 * de longueur très particulière.
 * Le type physique est fait pour que
 * le tableau soit initialisé à 1 pour donner du blanc. */
TYPE_PHYSIQUE(codec_couleurNB, /* Le nom du type physique */
              CouleurNB,       /* La structure C associée au type physique */
              codec_l(a,       /* Table de taille fixe */
                      NB*3,    /* La taille */
```

```
                    codec_float1 /* Flottant avec valeur par défaut 1 */
                    );
          )
#define P_couleurNB TP(codec_couleurNB)

static Triplet point(Triplet *a, Triplet *b, float i) /* Utilitaire */
{
  i /= NB;
  return( u_ponderation_2_triplet(a, i,    b, 1-i) );
}
static void affiche_segment(struct segment *s)
{
  int i;
  Triplet pt;
  glBegin(GL_LINES);
  for(i=0;i<NB;i++)
    {
      glColor3f(s->couleur[i][0], s->couleur[i][1], s->couleur[i][2]);
      pt = point( &s->depart, &s->arrivee, i+.1);
      t_triplet(&pt);
      pt = point( &s->depart, &s->arrivee, i+1-.1);
      t_triplet(&pt);
    }
  glEnd();
  if ( ACTIF(s) ) /* Pour ne pas surcharger l'affichage */
    for(i=0;i<NB;i++)
      {
        pt = point( &s->depart, &s->arrivee, i+.5);
        t_tag(NUMERO_TAG+i, &pt);
      }
}
/* La fonction suivante permet de changer la valeur du tag,
 * dans notre cas on change la couleur. */
static void ecrit_tag(struct segment *s, const Champ *c, const char *v)
{
  int i;
  Triplet couleur;

  i = lit_numero_tag(c) - NUMERO_TAG;
  if ( i >= 0 && i < NB ) /* En cas de bug... */
    {
      couleur = char_vers_triplet(v);
      s->couleur[i][0] = couleur.x;
      s->couleur[i][1] = couleur.y;
      s->couleur[i][2] = couleur.z;
    }
}
/* Lecture de la valeur du tag */
static const char * lit_tag(struct segment *s, const Champ *c)
{
  static Buffer chaine;
  int i;
  chaine[0] = '\0';
  i = lit_numero_tag(c) - NUMERO_TAG;
  if ( i >= 0 && i < NB ) /* En cas de bug... */
    {
      sprintf(chaine, "Couleur = %g %g %g",
              s->couleur[i][0], s->couleur[i][1], s->couleur[i][2]);
    }
  return(chaine);
}
CLASSE(segment22,
       struct segment,
       Edite Sauve
```

```
    CHAMP(depart      , L_point P_triplet                   Init3d Extrait)
    CHAMP(arrivee     , L_point P_triplet                   Init3d Extrait)
    CHAMP(couleur     , L_table P_couleurNB                 )
    CHAMP_VIRTUEL(      L_tag(ecrit_tag,lit_tag)            )
    CHAMP_VIRTUEL(      L_affiche_gl(affiche_segment)       )
   )
```

3.14 Amélioration des types

Il est facile d'ajouter un type logique "bipoint3" bien que le "3" n'apporte pas de réelle sémantique par contre cela peut simplifier la définition des champs car on peut donner un type physique par défaut. On peut aussi indiquer beaucoup d'autre valeurs par défaut. On n'a pas besoin de réindiquer la hauteur du Widget car elle est héritée.

```
DEFINITION_TYPE_LOGIQUE(L_bipoint3, /* Nom du type logique défini */
                        L_bipoint,  /* Nom de la surclasse */
                        "Bipoint3", /* Commentaire */
                        c->tp = &infos_codec_bitriplet ;
                        )
```

Si le segment utilise "L_bipoint3" il n'a pas besoin d'indiquer "P_bitriplet" car le type logique l'indique pour lui.

Il est néanmoins plus propre d'indiquer "L_bipoint P_bitriplet" car la valeur pourra être extraite du fils que le type du champ (dans le fils) soit "L_bipoint4" et pas seulement "L_bipoint3".

3.15 Vérifions le type bipoint avec un calcul de distance

Faisons un objet qui prend un bipoint comme paramètre et qui calcule la distance entre les deux points ("o_s12.c") :

```
#include "o_objet.h"
#include "t_segment.h"

struct distance { Bitriplet2 pq ; Flottant d ; } ;

static void changement_bipoint(struct distance *o)
{
  if ( CHAMP_CHANGE(o, pq) )
    o->d = u_distance_triplet(&o->pq.t1, &o->pq.t2) ;
}

CLASSE(segment12,
    struct distance,
    CHAMP(pq , L_bipoint2 P_bitriplet2 Edite Sauve Init3d Extrait)
    CHAMP(d  , L_flottant              Affiche)
    CHANGEMENT(changement_bipoint)
   )
```

Remarques concernant l'extraction :

- Elle se fera même si dans le fils le type est "L_bipoint3" ou "L_bipoint4" car elle demande à extraire un "L_bipoint" qui est plus général.
- Si le type du champ avait été "L_bipoint3" on aurai seulement pu extraire des champs de type "L_bipoint3"

3.16 Ajout d'un bouton pour échanger les deux points

Dans la fenêtre d'édition, on veut mettre un bouton permettant d'échanger le point de départ et d'arrivé du segment ("o_s13.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee; };

static Booleen echange(const Interface_widget *iw)
{
  Triplet tmp;
  struct segment *s;

  s = iw->objet;
  tmp = s->depart;
  s->depart = s->arrivee;
  s->arrivee = tmp;

  return(Faux); /* Ceci indique que l'on veut réévaluer le modèle */
}

CLASSE(segment13,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP_VIRTUEL(LABEL("Echange D & A") L_bouton(echange))
      )
```

La structure "Interface_widget" contient des informations plus détaillée sur l'événement (X11) qui a généré l'événement.

3.17 Ajout d'une bascule pour rendre le segment invisible

La bascule est par défaut de type "Booleen" mais vous pouvez lui donner le type physique que vous voulez, il prendra les valeurs "0" ou "1". Dans l'exemple, la bascule permet de choisir d'afficher ou non le segment ("o_s14.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
#include "u_graphique.h"              /* Pour avoir t_triplet */
#include <GL/gl.h>
struct segment {  Triplet depart, arrivee;
                  Booleen visible;
               };
static void affiche_segment(struct segment *s)
{
  if ( !s->visible ) return;                    /**/
  glBegin(GL_LINES); t_triplet(&s->depart); t_triplet(&s->arrivee); glEnd();
}
CLASSE(segment14,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(visible, L_booleen DEFAUT("1") Edite)              /**/
       CHAMP_VIRTUEL(L_affiche_gl(affiche_segment))
      )
```

3.18 Ajout d'un menu option

Le menu option permet de choisir une valeur parmis un ensemble de valeur prédéfinies. Le menu option a comme valeur la chaine de caractères choisie par l'utilisateur. Dans l'exemple, le menu permet de choisir la fermeture des bornes du segment de droite parmi les quatre possibles ("o_s15.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee;
                 Menu menu;
               };
static const char* information_segment(struct segment *s)
{
  static Buffer tmp;
  sprintf(tmp, "Ouverture : %s", s->menu.selection);
  /* o->menu.index est un entier indiquant le numéro du choix */
  return(tmp);
}
CLASSE(segment15,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(menu    ,
             LABEL("Ouverture :")
             Edite Sauve
             L_option_menu
             TE_TEXTE(          /* Les options possibles */
                      "[]\n"
                      "[[\n"
                      "]]\n"
                      "][\n"
                      )
             DEFAUT("[]") /* Option par defaut */
             )
       INFORMATIONS(information_segment)
      )
```

Il est aussi possible de modifier le contenu du menu durant l'éxécution du programme en changeant la valeur de «menu.description».

Le caractère tabulation dans la description du menu permet de définir des colonnes de tableau. On peut ajouter à la ligne un caractère de code ASCII 1 après lequel vous pouvez indiquer un texte qui sera la valeur réelle de l'option suivit d'un arobase (@) pour indiquer que la ligne est inactive (texte en grisé dans le menu).

3.19 Ajout d'un menu de commande

Le menu de commande contrairement au menu option permet d'appeler directement une fonction une fois le choix fait. Dans le cas du segment, une commande pour annuler le premier point et une autre pour annuler le deuxième ("o_s16.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment { Triplet depart, arrivee;
                 Menu menu;
               };
static Booleen commande(const Interface_widget *iw)
{
  struct segment *s;
  s = iw->objet;
  switch(s->menu.index)
```

```
    {
    case 0:  u_annule_triplet(&s->depart) ; break ;
    case 1:  u_annule_triplet(&s->arrivee) ; break ;
    default: eprintf("Bug : la commande ``%s'' numéro (%d)\n",
                     s->menu.selection, s->menu.index) ;
    }
  return(0); /* Recalcule tout */
}
CLASSE(segment16,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(menu    ,
             LABEL("Commandes agissant sur le segment")
             L_commande_menu(commande)
             TE_TEXTE(          /* Les commandes possibles */
                      "annule depart\n"
                      "annule arrivee\n"
                      )
             )
      )
```

3.20 Affichage du segment en POV

Il faut ajouter un champ particulier qui indique comment traduire l'objet en POV. Ce champ particulier est "déclenché" par l'objet "pov" qui génère le fichier et lance POV ("o_s17.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
struct segment {  Triplet depart, arrivee ; } ;

static void segment_pov(struct segment *s, const Champ *c, const char *v)
{
  fprintf( (FILE*)atoi(v),
          "cylinder { <%g,%g,%g>, <%g,%g,%g>, 0.01 }\n",
          s-> depart.x, s-> depart.y, s-> depart.z,
          s->arrivee.x, s->arrivee.y, s->arrivee.z) ;
}
CLASSE(segment17,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP_VIRTUEL(L_affiche_pov(segment_pov))           /**/
      )
```

3.21 Mettre une fenêtre Open GL dans l'éditeur de segment

Ceci peut être utile pour faire une édition graphique de la valeur d'un champ ou de l'objet, dans NanoÉdit il existe déjà "vue", "couleur", "image". L'exemple suivant affiche un segment probabiliste ("o_s18.c") :

```
#include "t_gl.h"                          /**/
#include "o_objet.h"
#include "t_geometrie.h"
#include "u_graphique.h"
struct segment {  Triplet depart, arrivee ;
                  Gl gl ;                  /**/
               } ;
static void segment_trace(const Interface_widget *iw)
{
  struct segment *o ;
```

```
  int i;
  Flottant ac, di;
  Triplet tmp, vec, vecd;

  o = iw->objet;
  vec = u_soustrait_triplet(&o->depart, &o->arrivee);
  vec.z = vec.x;
  vec.x = vec.y;
  vec.y = -vec.z;
  vec.z = 0;
  Gl_start(&o->gl);
  glClearColor(0.,0.,0.,0.);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBegin(GL_POINTS);
  for(i=0;i<1000;i++)
    {
      ac = (rand()%10000)/9999.;
      di = (rand()%10000)/9999. - .5;
      di *= di * di;
      tmp = u_ponderation_2_triplet(&o->depart, ac, &o->arrivee, 1-ac);
      vecd = u_triplet_fois_flottant(&vec, di);
      tmp = u_ajoute_triplet(&tmp, &vecd);
      t_triplet(&tmp);
    }
  glEnd();
  Gl_stop(&o->gl);
}
CLASSE(segment18,
       struct segment,
       CHAMP(depart , L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point P_triplet Edite Sauve Init3d Extrait)
       CHAMP(gl,
             L_gl Edite
             TE_HAUTEUR(500) TE_LARGEUR(500)
             TE_ECRIT(segment_trace))
      )
```

Je ne détaille pas ici (voir fin du document), mais on peut définir plein de méthodes pour manipuler cette fenêtre graphique. Aide contextuelle, raccourci clavier, bouton de la souris, déplacement du curseur, ...

3.22 Mettre une image dans l'éditeur de segment

Parfois, au lieu de faire du GL, on veut simplement afficher une image que l'on vient de calculer dans la fonction "changement". Pour faire cela, il suffit d'ajouter l'image dans la structure C (avec le type "Image") un "éditeur" d'image nommé "E_image" qui affichera automatiquement l'image à chaque changement. La fonction changement se charge d'ailleur d'allouer/libérer l'image et d'indiquer son type.

Une fonction qui prend les événements souris a été ajoutée, elle permet de déplacer le premier point du segment avec le bouton de gauche et le deuxième avec le bouton du milieu. On peut laisser le bouton appuyé pour faire le suivi du mouvement.

Deux modèles ont étés fait avec cet objet, l'un extrait l'image ("EXEMPLES/23_image_3d"), l'autre non ("EXEMPLES/23_image"). Il faut bien comprendre qu'extraire l'image permet de faire des choses très intéressante mais que c'est très lent car toute la matrice de points est convertie en chaine de caractère et inversement... ("o_s23.c") :

```
#include "t_image.h"                    /**/
#include "t_geometrie.h"
#include <math.h>
```

```
struct segment { Image image; Flottant l,h;  Triplet depart, arrivee; };

static void changement_segment(struct segment *o)
{
  int x, y;
  unsigned char *pc;

  if ( !UN_CHAMP_CHANGE(o) )
    return;
  if ( L3_UNSIGNED_CHAR(&o->image) == NULL
       || LAR(&o->image) != o->l
       || HAU(&o->image) != o->h
       ) /* Allocation */
    {
      o->image.codec = codec_HL_l3_unsigned_char;
      LAR(&o->image) = o->l; /* Pourrait être une variable éditable */
      HAU(&o->image) = o->h;
      REALLOUER(L3_UNSIGNED_CHAR(&o->image),o->l*o->h);
    }
  /* Calcul de l'image */
  pc = (unsigned char*) L3_UNSIGNED_CHAR(&o->image);
  for(y=0;y<o->h;y++)
    for(x=0;x<o->l;x++)
      {
        *pc++ = x;
        *pc++ = y;
        *pc++ = 128 + (sqrt(((x - o->depart.x)  *  (x - o->depart.x)
                 + (y - o->depart.y)  *  (y - o->depart.y))) -
                sqrt((x - o->arrivee.x)  *  (x - o->arrivee.x)
                 + (y - o->arrivee.y)  *  (y - o->arrivee.y)))*2;
      }
}
/* En cliquant dans la fenêtre image, on modifie l'un des points */
static Booleen segment_entree(const Interface_widget *iw)
{
  struct segment *o = iw->objet;
  if ( iw->b == 0 ) /* 0 représente le bouton de gauche */
    { o->depart.x  = iw->x; o->depart.y  = o->h - iw->y; }
  else
    { o->arrivee.x = iw->x; o->arrivee.y = o->h - iw->y; }
  return(1); /* 0 Si l'on ne veut pas faire de suivie de mouvement */
}
static void segment_entree_mouvement(const Interface_widget *iw)
{
  segment_entree(iw);
}
CLASSE(segment23, struct segment,
       CHAMP(depart , L_point  P_triplet       Edite Sauve Init3d Extrait)
       CHAMP(arrivee, L_point  P_triplet       Edite Sauve Init3d Extrait)
       CHAMP(l      , L_nombre P_flottant      Edite Sauve Init3d DEFAUT("64"))
       CHAMP(h      , L_nombre P_flottant      Edite Sauve Init3d DEFAUT("64"))
       CHAMP(image  , L_grille_point E_image P_image_l3_unsigned_char
             Affiche TE_HAUTEUR(512) TE_LARGEUR(512)
             TE_ENTREE(segment_entree) TE_MOUVEMENT(segment_entree_mouvement)
             )
       CHANGEMENT(changement_segment)
      )
```

Pour contourner ce problème, on peut extraire un pointeur sur un champ de l'objet. L'objet "inversion" prend une image codée sous la forme d'une grille de point et applique une inversion vidéo. Il suffit d'une ligne pour indiquer que l'on veut extraire un pointeur sur un champ. Le pointeur est une exception dans le fonctionnement de NanoÉdit qui a été faite uniquement pour éviter de transférer de grosses structures de données. Une particularité est que le type de ce que l'on extrait est différent du type dans lequel on stocke, dans cet exemple la donnée est un pointeur mais l'on recherche une image dans le paramètre. C'est pour indiquer cette différence que "TYPE_DANS_FILS" est précisé. "o_inversion.c") :

```
#include "t_image.h"                                /**/
#include "t_geometrie.h"
#include <math.h>

struct inversion { Image image; Image *entree; };

static void changement_image(struct inversion *o)
{
  int taille, i;
  unsigned char *in, *out;

  if ( o->entree==NULL || L3_UNSIGNED_CHAR(o->entree)==NULL )
    return;
  if ( LAR(&o->image) != LAR(o->entree) ||
       HAU(&o->image) != HAU(o->entree) )
    {
      o->image.codec = codec_HL_l3_unsigned_char;
      LAR(&o->image) = LAR(o->entree);
      HAU(&o->image) = HAU(o->entree);
      REALLOUER(L3_UNSIGNED_CHAR(&o->image),LAR(&o->image)*HAU(&o->image));
    }
  out = (unsigned char*) L3_UNSIGNED_CHAR(&o->image);
  in = (unsigned char*) L3_UNSIGNED_CHAR(o->entree);
  taille = LAR(&o->image) * HAU(&o->image) * 3;

  for(i=0;i<taille;i++)
     *out++ = 255 - *in++;
}
CLASSE(inversion, struct inversion,
       CHAMP(image, L_grille_point P_image_l3_unsigned_char E_image
             Affiche TE_HAUTEUR(256))
       CHAMP(entree, L_pointeur_extrait              /**/
             TYPE_DANS_FILS(L_image_l3_unsigned_char) /**/
             Extrait)                              /**/
       CHANGEMENT(changement_image)
      )
```

3.23 Affichage d'une ligne brisée

On peut définir un objet prenant une nombre indéfini de paramètres, par exemple `applique`, `table_point`, ... Ces paramètres ne sont pas extraits car le nombre de champ n'est pas une constante, il dépend du nombre de paramètres.

L'objet suivant, prend un nombre quelconque d'objets en paramètre et lors de son affichage extrait de chacun de ses paramètres les coordonnées d'un point afin de tracer la ligne brisée ("o_s19.c") :

```
#include "o_objet.h"
#include "t_geometrie.h"
#include <GL/gl.h>
struct ligne_brisee { };
static void affiche_lb(struct ligne_brisee *s)
{
  const char *texte;
  Triplet point;

  glBegin(GL_LINE_STRIP);
  POUR_LES_FILS          /* Boucle sur tous les fils non détruit */
```

```
  (s,                          /* De "s" */
   fils,                       /* Nom de l'indice de boucle */
   {
     /* Récupère un champ de type L_point dans le fils */
     texte = c_recupere_type_sans_copie(fils, &infos_L_point);
     /* Teste si la chaine de caractère n'est pas vide */
     if ( texte[0] )
       {
         /* Conversion de la chaine en triplet */
         point = char_vers_triplet(texte);
         /* Affichage */
         glVertex3f(point.x, point.y, point.z);
       }
   }
   );
  glEnd();
}
CLASSE(segment19,
       struct ligne_brisee,
       CHAMP_VIRTUEL(L_affiche_gl(affiche_lb))
       /* Lors de la création de l'objet, il prend TOUS
        * les actifs comme paramètre. */
       PAQUET(NB_ACTIFS)
      )
```

Remarques :

- Cet objet n'est pas optimal, en effet, il est plus performant d'avoir une fonction "changement" qui stocke dans un tableau interne les coordonnées des points. En fait c'est inutile de le faire car l'objet "table de point" est fait pour cela, il n'y a qu'à extraire une table de point toute faite.
- Plusieurs méthodes permettent d'accéder aux paramètres, celle de l'exemple est la plus simple et elle saute automatiquement les paramètres détruits. Il existe "POUR_TOUS_LES_FILS" pour parcourir les détruits et l'on peut accéder directement à un fils avec la macro "FILS(objet,numéro)"
- Dans l'exemple on utilise la fonction "c_recupere_type_sans_copie" mais on peut extraire un champ particulier en donnant son nom au lieu de son type avec la fonction "c_recupere_ici_sans_copie(objet, "nom-champ")"
- On peut de plus mettre une contrainte de type sur certains des paramètres sans les extraire grace à l'option "Obligatoire". Ceci permet d'avoir le type de premier argument dans le menu de création plutot que "(...)" qui n'est pas explicite.

3.24 Modification d'une valeur à l'aide d'un ascenseur

L'utilisation d'un ascenseur permet de modifier en temps réel une valeur. Sa contrepartie est que l'ascenseur est linéaire et donc borné. Si l'on ne défini pas les bornes celles-ci sont 0 et 1.

L'éditeur de type ascenseur est donc à lier avec un triplet de valeur (entières ou flottantes), les deux dernières indiquant le minimum et le maximum ("o_ascenseur.c") :

```
#include "o_objet.h"
#include "t_geometrie.h" /* Pour Triplet */

typedef struct
{
  Triplet valeur;
} Valeur_bornee;

CLASSE(valeur_bornee, Valeur_bornee,
       CHAMP(valeur ,
             L_nombre P_triplet E_ascenseur /* Ou L_flottant_borne */
```

```
        Edite Sauve LABEL("Valeur") DEFAUT("0 -3.141 3.141")
        BULLE_D_AIDE("Le triplet : valeur courante, min, max\n")
        )
   )
```

3.25 Paramètres de type objet

La grande force de NanoÉdit est que l'extraction s'occupe de prendre un contenu dans les objets. Dans notre cas, le segment veut extraire deux points de ses paramètres quelque soit l'objet qui contient le champ point. Dans certain cas, on veut extraire un pointeur sur l'objet paramètre, dans ce cas, il est nécessaire de fixer la classe de l'objet. Dans l'exemple, le segment récupère un pointeur sur un objet "point3" et un "point4" ("o_s20.c") :

```
#include "o_objet.h"

struct segment
{
  Triplet *depart ;
  Triplet *arrivee ;
} ;

CLASSE(segment20,
       struct segment,
       L_entier              /* On considère les pointeur comme entier */
       Affiche               /* On les affiche : pas d'édition/sauvegarde */
       Obligatoire           /* Car on ne veut pas de pointeur NULL */
       CHAMP(depart , CLASSE_PARAMETRE("point3") Extrait)
       CHAMP(arrivee, CLASSE_PARAMETRE("point4") Extrait)
       )
```

3.26 Disposition des champs dans la fenêtre

Vous pouvez choisir la disposition des champs dans la fenêtre par une méthode rappelant celle des *frames* HTML. La taille de la fenêtre est la plus petite contenant tous les champs. Le dernier champ de chaque boite est agrandi pour ne pas laisser de trous.

L'option "BoiteHomogene" indique que tous les champs de la boite doivent se partager l'espace équitablement ("o_s24.c") :

```
#include "o_objet.h"

struct disposition
{
  Booleen a, b, c, d, e, f, g, h, i, j, k, l ;
} ;

CLASSE(disposition, struct disposition,
       Edite L_booleen
       DEBUT_BOITE(HORIZONTALE)
              CHAMP(a,)
              CHAMP(b,)
              CHAMP(c,)
       FIN_BOITE
       DEBUT_BOITE(HORIZONTALE)
              CHAMP(d,)
              DEBUT_BOITE(VERTICALE)
                  DEBUT_BOITE(HORIZONTALE)
                      CHAMP(e,)
                      CHAMP(f,)
                  FIN_BOITE
```

```
            CHAMP(g,)
            CHAMP(h,)
        FIN_BOITE
        CHAMP(i,)
    FIN_BOITE
    DEBUT_BOITE(HORIZONTALE BoiteHomogene)
    CHAMP(j,)
    CHAMP(k,)
    CHAMP(l,)
    FIN_BOITE
)
```

4 Références

Les sections qui suivent ne doivent être lu que lorsque ce qui précède a déjà été assimilé.

4.1 Les types logique

Les types logiques les plus utilisés sont définis dans "TYPES/t_base.h" et "TYPES/t_geometrie.h". Un type logique peut modifier n'importe quel attribut de champ d'un objet (même le type physique) mais évidemment c'est dangereux.

Les types logique permette une définition d'appartenance en plus de l'inclusion (héritage) par deux méthodes :

- Le prédicat : pour un champ donné on appelle cette fonction pour savoir si le champ vérifie le prédicat ou non. Cette fonction permet de réduire la taille de la classe. Par exemple "L_nombre_positif" est un "L_nombre" qui n'accepte que des valeurs positives et "L_triplet" est un "L_valeur" stocké dans un type physique "P_triplet".
- Le sous classage : pour un champ donné on appelle cette fonction pour savoir si le champ peut être considéré d'un type logique donné. Cette fonction permet d'augmenter la taille de la classe. Cette fonction est rarement utile.

Si la valeur contenue dans un champ ne vérifie pas son type logique, alors l'objet et le champs apparaissent en rouge (invalide) et le rouge se propage dans tous les noeuds les utilisant. Par exemple, l'objet couleur devient invalide si l'une des 3 composantes devient négative car "L_nombre_positif" n'est plus vérifié.

La hiérarchie actuelle des types logiques est la suivante :

```
 Void, Racine des types logiques
     L_fonction, Fonction de champ
         L_debut_boite_verticale, Début boite verticale (disposition widget)
         L_debut_boite_horizontale, Début boite horizontale (disposition widget)
         L_fin_boite, Fin boite (disposition widget)
         L_prevenir_si_modele_change, Sert a indiquer si l'objet veut être prévenu
de changements dans le modèle global.
         L_tag, Gestion des tag virtuels
         L_affiche_gl, Méthode d'affichage GL
         L_affiche_pov, Méthode de traduction en POV
         L_applique_gl, Méthode de changement d'état courant GL
         L_applique, Méthode de changement d'état courant
     L_void, Racines des type logiques ''normaux''
         L_bouton, Bouton de commande dans une fenêtre
         L_titre, Indique le titre de la fenêtre
         L_gl, Widget GL
         L_geometrie, Indique la géométrie de la fenêtre
         L_attribut_pov, Pour POV c'est un attribut
```

```
    L_valeur, Éditeur de texte
        L_classe, Classe de NanoÉdit
        L_nombre, Valeur
            L_booleen, Booleen
            L_entier, Prédicat(nombre_entier) , Entier
            L_flottant, Prédicat(nombre_flottant) , Flottant
            L_nombre_positif, Prédicat(nombre_positif) , Nombre positif
            L_flottant_borne, Prédicat(valeur_dans_les_bornes) , Nombre flottant borné
        L_observateur, Observateur (point de vue, point visé, haut, focale)
        L_point, Point
            L_point3, Point à 3 composantes
            L_point4, Point à 4 composantes
        L_chaine, Chaine de caractères
            L_fichier_image, Image sans type physique
                L_fichier_image_l3_unsigned_char, Image de triplet unsigned char
                L_fichier_image_l3_float, Image de triplet de float
                L_fichier_image_unsigned_char, Image de triplet unsigned char
        L_boite_min_max, Boite minmax de R3 parrallèle aux axes
        L_solide, Solide
        L_vecteur, Vecteur
            L_vecteur3, Vecteur à 3 composantes
            L_vecteur4, Vecteur à 4 composantes
        L_bipoint, Paire de points
        L_vecteur_lie, Vecteur lié défini par Point et Vecteur
        L_triplet, Prédicat(triplet) , Triplet de valeur
        L_liste, Liste d'entier
            L_liste_objet, Liste de pointeurs sur objets NanoÉdit
            L_sous_actif, Liste de sous_actifs
        L_table, Table
            L_table_point, Table de points
                L_table_point3, Table de points à 3 composantes
                L_table_point4, Table de points à 4 composantes
            L_table_nombre, Table de nombre
                L_table_flottant, Prédicat(table_flottant) , Table de flottant
            L_table_table, Table de table
                L_table_table_point, Table de table de points à 4 composantes
                    L_table_table_point3, Table de table de points à 3 composantes
        L_grille, Grille
            L_grille_point, Grille de point
                L_grille_point3, Grille de point à 3 composantes
                L_grille_point4, Grille de point à 4 composantes
            L_grille_nombre, Grille de nombre
                L_transfo, Prédicat(transfo) , Transformation affine
                L_grille_flottant, Prédicat(grille_flottant) , Grille de flottant
        L_graphe, Graphe quelconque
            L_graphe_de_points, Graphe de points
        L_couleur, Couleur
            L_couleur3, Couleur à 3 composantes
    L_graphe_modele, Affichage du graphe du modèle de NanoÉdit
    L_menu, Menu
        L_option_menu, Menu permettant de choisir un option
        L_commande_menu, Menu déclenchant des exécutions
L_metamorphe, Racines des type logiques métamorphique
    L_variable, Sous_classable(sous_classe_variable), Variable NanoÉdit
    L_operateur, Sous_classable(sous_classe_operateur), Opérateur NanoÉdit
```

4.2 Les types physique

Les types physiques de base sont préfixé par “codec_” et sont définis dans le fichier “UTILITAIRES/codec.h”, et ceux qui sont déjà intégré dans NanoÉdit sont préfixé par “P_” et sont définis dans les fichiers “t_”.

Un des problèmes en C est l'existence de “double” et “float” et que certaines personnes utilisent l'un ou l'autre. Pour ne pas alourdir les programmes, NanoÉdit définit “flottant” et “triplet” cela allège l'écriture mais l'on ne sait pas si “flottant” est un “float” ou un “double”. La table de conversion est dans “UTILITAIRES/u_geometrie.h”.

Les “codec” définissent un type physique par une fonction C. Suivant le mode, la fonction C effectue différentes opérations. Les opérations de base sont la traduction en chaine de caractères dans les deux sens et la libération de l'espace mémoire utilisé par le type physique.

Les types de base sont suffisemment explicites et les autres (énumération, pointeur, union, liste, graphe, ...) sont complexes et je n'ai pas le temps de le définir ici bien qu'en final leur utilisation soit triviale. Par exemple, 25 lignes pour définir la traduction dans les deux sens d'un octree ainsi que la libération mémoire.

Pour les tableaux qui sont très souvent utilisés, voici quelques explications sur la manière de nommé et d'utiliser ces types.

4.2.1 Tableaux

Les noms de type tableaux sont construit logiquement et sont sans ambiguïté. Il sont donc facile à utiliser une fois la notation comprise. Quelques indices généraux :

- Le type d'élément contenu dans le tableau est à la fin du nom.
- Dans tous les cas, les indices d'un tableau sont dans le même ordre qu'en C.
- Les indices sont nommés : Profondeur, Hauteur, Largeur. Leurs abréviations sont donc P, H et L.
- Si le nom de l'indice est en minuscule cela veut dire que la dimension du tableau associée à cet indice est constante.
- Si le nom de l'indice est en minuscule alors il est suivi de la valeur numérique de la dimension du tableau, par exemple “codec_l3_int” est un tableau à une dimension de 3 “int”.

Tableau à une dimension de taille variable Voici l'exemple du tableau de “double”, le codec associé au tableau à une dimension de taille variable est “codec_L” le “L” indiquant la largeur. Voici ce qu'il faut mettre dans le “.h” :

```
typedef struct
{
  int     lar ;                        /* Largeur du tableau */
  double *table ;                      /* Pointeur sur le début du tableau */
} t_L_double ;

CODEC_PROTOTYPE(codec_L_double, t_L_double)
```

Et dans le “.c” :

```
TYPE_PHYSIQUE(codec_L_double,          /* Nom du codec */
              t_L_double,              /* Nom de la structure C à (dé)coder */
              codec_L(                 /* C'est un tableau à 1 dimension */
                      (void**)&a->table,/* L'adresse du pointeur de début */
                      &a->lar,         /* L'adresse de la taille du tableau */
                      codec_double     /* Le type des éléments du tableau */
                     ) ;
              )
```

"codec_L" se charge de l'allocation et de la libération du tableau et de son contenu. Parfois on ne peut pas utiliser directement "t_L_double" car les structures existantes ne contiennent pas la taille du tableau suivi du pointeur sur le début, dans ces cas là, il faut définir un codec spécialement pour votre type.

On peut évidemment faire des tableaux de tableaux, par exemple un tableau de tableau d'entier s'appelera "codec_L_L_int" chacun des tableaux contenu peut avoir une taille différente (ce n'est pas un tableau à deux dimension).

Tableau à une dimension de taille fixe Dans le ".h" :

```
typedef struct
{
  int table[5] ;
} t_l5_int ;

CODEC_PROTOTYPE(codec_l5_int, t_l5_int)

TYPE_PHYSIQUE(codec_l5_int,
              t_l5_int,
              codec_l(                    /* C'est un tableau à 1 dimension */
                      (void*)a->table,    /* L'adresse du début */
                      5,                  /* La taille du tableau */
                      codec_int           /* Le type des éléments du tableau */
                     ) ;
              )
```

Chaque tableau d'une taille particulière nécessite la création d'un type.

Tableau à deux dimensions de taille variable Ce tableau à deux dimensions est stocké en un seul bloc de mémoire comme en C. Dans le ".h" :

```
typedef struct
{
  int hau ;                 /* hauteur (nombre de lignes) */
  int lar ;                 /* largeur (nombre de colonnes) */
  t_l3_double *table ;      /* Pointeur sur le premier élément */
} t_HL_l3_double  ;

CODEC_PROTOTYPE(codec_HL_l3_double, t_HL_l3_double)
```

Vous avez remarqué que "hau" est déclaré avant "lar", ceci afin d'utiliser systématiquement le même ordre des indices que lors de l'utilisation d'un tableau C.

```
TYPE_PHYSIQUE(codec_HL_l3_double,
              t_HL_l3_double ,
              codec_HL(
                       (void**)&a->table,
                       &a->hau,
                       &a->lar,
                       codec_l3_double
                      ) ;
              )
```

Tableau avec indirection à deux dimensions de taille variable Dans ce cas, on a un tableau de pointeur sur les lignes du tableau. On représente le pointeur par un "_" entre le "H" et le "L". Dans le ".h" :

```
typedef struct
{
  int hau ;                 /* hauteur (nombre de lignes) */
  int lar ;                 /* largeur (nombre de colonnes) */
  t_l3_double *table ;      /* Pointeur sur le premier élément */
} t_H_L_l3_double  ;

CODEC_PROTOTYPE(codec_H_L_l3_double, t_H_L_l3_double)
```

Le ".c" :

```
TYPE_PHYSIQUE(codec_H_L_l3_double,
              t_H_L_l3_double ,
              codec_H_L(
                       (void***)&a->table,
                       &a->hau,
                       &a->lar,
                       codec_l3_double
                      ) ;
              )
```

Autres tableaux existants

- "codec_hl" : Tableau à deux dimensions de taille fixe.
- "codec_PHL" : Tableau à trois dimensions de taille variable.
- "codec_h_L" : Tableau à deux dimensions avec un nombre de lignes fixe.
- "codec_HH" : Tableau à deux dimensions carré.
- "codec_L_h : Comme "h_L" mais la traduction textuelle est transposée. C'est utile par exemple si vous avez le type C :

  ```
  struct table_point
  {
    int nb ;
    float *x, *y, *z ;
  } ;
  ```

  Si vous créez pour cette structure "codec_L_h3" cela vous permettra de communiquer les valeurs aux structures qui contienne un tableau de triplet. En effet, une simple transposition suffit.
- "codec_L_hh" : comme pour "L_h" le tableau est transposé mais en plus il contient deux lignes de types différents, c'est pour cela que c'est "hh" et non "h2". C'est utile pour le type C suivant :

  ```
  struct table_point
  {
    int nb ;
    t_l3_double *points ;
    float *poids ;
  } ;
  ```

  En utilisant ce codec, on peut communiquer sans probleme avec des "l4_double" car on peut écrire le poids derrière les coordonnées du point.
- "codec_H_L_pp" : comme le précédent mais appliqué à deux tableaux de deux dimensions qui doivent être mis ensemble. Par exemple une grille de points et une grille de poids.

Utilitaires sur les tableaux et grilles Il existe des fonctions d'accès générique aux éléments des tableaux et grille :

```
/* Déclaration itérateur de table */

struct informations_table it ;

/* Initialise l'itérateur avec le type de la table et sa valeur
 * it.dimension     : Dimension de l'espace (1,2,3)
 * it.taille[0/1/2] : Taille des dimension (même ordre qu'en C)
 */
it = codec_informations_table(codec_L_float, void *table) ;

/* Positionnement de l'élément courant dans la table ou la grille */
codec_positionne_element_table(&it, j, /* i si grille */ ) ;

/* Récupère la valeur de l'élément courant et passe au suivant
 * si c'est le dernier, revient au premier */
valeur = codec_recupere_element_courant(&it) ;

/* Sauvegarde une valeur dans l'élément courant et passe au suivant */
codec_sauve_element_courant(&it, "0") ;
```

Ces fonctions ont permis d'écrire "u_table.c" qui contient des fonctions "changement" génériques et des fonctions d'affichage GL ou POV des tables ou grilles d'élement quelconques. Le moyen le plus simple de profiter de tout cela est de nommer "t" le champs contenant la table ou la grille et dvous mettre en sous-classe de "table_non_contrainte", "table_contrainte", "grille_non_contrainte", "grille_contrainte".

Sinon, vous pouvez directement utiliser les fonctions utilitaires, certaine n'ont pas besoin que la table/grille soit un champ "NanoÉdit" :

- `changement_table`, `changement_grille` pour extraire la table/grille des fils ou la remettre dedans suivant ce qui a changé.
- `t_table_pov_general`, `t_table_pov`, `t_grille_pov_general`, `t_grille_pov` pour faire un affichage POV de tables ou des grilles. Les options d'affichage peuvent être combinées avec un OU binaire :
  - T_PONCTUEL : affiche une petite sphère en chaque point.
  - T_FILAIRE : affichage filaire simulé avec des cylindres.
  - T_POLYGONE : affichage filaire table, en raccordant les 2 extrèmes.
  - T_FACETTE : affichage facette classique.
  - T_LISSAGE : utilise la normale fournie en paramètre pour faire un lissage.
  - T_DEGENERE_(HAUT/BAS/GAUCHE/DROITE) indique qu'un coté de la grille est dégénéré (tous les points identiques).
  - T_FACETTE_(HAUT/BAS/GAUCHE/DROITE) indique qu'il faut générer une facette pour ce coté de la grille
  - T_RACCORDE_(GD/HB) indique qu'il faut raccorder le coté gauche au droit ou le haut et le bas. Les lignes/colonnes ne doivent pas être identiques.
- `t_table_general`, `t_table`, `t_grille_general`, `t_grille` pour faire un affichage GL de tables ou des grilles. Aux options précédentes, il faut ajouter :
  - T_TAG, T_TAG_DESTRUCTION, T_TAG_POIDS : affichage des TAGS désirés.
  - T_CULL_FACE : n'affiche pas les facettes à l'envers.
- `lit_tag_table`, `ecrit_tag_table`, `lit_tag_grille`, `ecrit_tag_grille` pour permettre de modifier interactivement les "tag" des tables ou des grilles.

4.2.2 Autres codec

Voici d'autre codec de base :

- "codec_enum" assure le passage d'un entier à une chaine de caractères (par exemple 0:Cube 1:Sphère 2:Cone ...)
- "codec_union" permet de choisir entre plusieur choix. Par exemple, il est utilisable pour coder un arbre, le type étant NOEUD ou FEUILLE et l'information dans les deux cas n'étant pas la même.
- "codec_pointeur" permet de coder un pointeur typé, le pointeur pouvant être NULL.
- "codec_pointeur_non_null" idem mais non NULL.

Des macros C permettent de générer automatiquement quelques codec, dans tous les cas, la donnée utilisateur est en fin de la structure de base.

- "CODEC_GRAPHE" permet de définir le graphe orienté avec les données que l'on veut sur le graphe, les noeuds et les arêtes. Évidemment comme tous les autres codec il s'occupe de l'allocation et de la libération.
- "CODEC_LISTE" permet de définir une liste de n'importe quoi.

Exemples de types construit :

- "codec_arbre_N_int_F_int" est un arbre binaire ou les Noeuds et les Feuilles contiennent un entier.
- "codec_octree"

4.3 Les types physique utilisés par NanoÉdit

```
codec_champ_texte                    (C: void                ,  0 octets)
codec_vide                           (C: void                ,  0 octets)
codec_classe                         (C: Classe*             ,  4 octets)
codec_char_decimal                   (C: char                ,  1 octets)
codec_actif                          (C: int                 ,  4 octets)
codec_adresse_objet                  (C: void*               ,  4 octets)
codec_int                            (C: int                 ,  4 octets)
codec_liste_int                      (C: t_liste_int*        ,  4 octets)
codec_virtuel                        (C: void                ,  0 octets)
codec_observateur                    (C: t_observateur       , 88 octets)
codec_l3_double                      (C: t_l3_double         , 24 octets)
codec_double                         (C: double              ,  8 octets)
codec_h4l3_double                    (C: t_l4_l3_double      , 96 octets)
codec_char_etoile                    (C: char*               ,  4 octets)
codec_option_menu                    (C: Menu                , 16 octets)
codec_commande_menu                  (C: Menu                , 16 octets)
codec_gl                             (C: Gl                  , 12 octets)
codec_L_l3_double                    (C: t_L_l3_double       ,  8 octets)
codec_L_l4_double                    (C: t_L_l4_double       ,  8 octets)
codec_L_double                       (C: t_L_double          ,  8 octets)
codec_H_L_l3_double                  (C: t_H_L_l3_double     , 12 octets)
codec_H_L_l4_double                  (C: t_H_L_l4_double     , 12 octets)
codec_H_L_double                     (C: t_H_L_double        , 12 octets)
codec_l2_l3_double                   (C: t_l2_l3_double      , 48 octets)
codec_L_L_l3_double                  (C: t_L_L_l3_double     ,  8 octets)
codec_void                           (C: void                ,  0 octets)
codec_bitriplet                      (C: Bitriplet           , 48 octets)
codec_fichier_image_l3_unsigned_char (C: Image               , 40 octets)
```

```
codec_graphe_N_l6_float            (C: t_graphe_N_l6_float      ,  12 octets)
codec_l4_double                    (C: t_l4_double              ,  32 octets)
codec_L_l4_double1                 (C: t_L_l4_double            ,   8 octets)
codec_modele                       (C: Liste**                  ,   4 octets)
codec_operateur                    (C: Operateur                ,  12 octets)
codec_fichier_image_l3_float       (C: Image                    ,  40 octets)
codec_image_l3_float               (C: Image                    ,  40 octets)
codec_image_l3_unsigned_char       (C: Image                    ,  40 octets)
codec_fichier_image_unsigned_char  (C: Image                    ,  40 octets)
codec_image_unsigned_char          (C: Image                    ,  40 octets)
codec_graphe_N_l3_float            (C: t_graphe_N_l3_float      ,  12 octets)
```

4.4 Les types d'éditions

Ils permettent d'associer un Widget d'affichage/édition à une valeur que l'on souhaite intégrer dans la fenêtre d'édition. Ils sont peu nombreux et complexe à ajouter, je ne redige donc pas la documentation. Voici la liste actuelle :

```
editepas
    E_bouton
    E_texte
        E_long_texte
    E_booleen
    E_titre
    E_option_menu
    E_commande_menu
    E_gl
        E_image
    E_geometrie
    E_ascenseur
    E_graphe_modele
```

4.5 Tout ce que l'on peut mettre dans un champ

- "`NOM(...)`" : Le nom du champ (par défaut le nom dans la structure C, on ne le change généralement pas), celui-ci est utilisé pour nommer :
  - les Widgets (utile pour les ressources X11).
  - les champs dans les fichiers de sauvegarde.
  - ...
- "`L_...`" La classe (ou type) logique qui indique la sémantique du champ. Parfois, le type logique est suffisant pour associer un type de stockage au champ.
- "`P_...`" Le type de stockage. Ce type doit correspondre au type utilisé dans la structure du langage C. Ce type détermine comment le champ est traduit de chaîne de caractères en valeur ou inversement.
- "`E_...`" L'éditeur interactif utilisé pour l'édition du champ.
- "`DEFAUT("...")`" indique la valeur par défaut du champ.
- "`BULLE_D_AIDE("...")`" associée au champ.
- "`LABEL("...")`" label indiqué à gauche du champ, normalement le nom du champ.
- "`TE_HAUTEUR(...)`" hauteur du Widget en pixel.
- "`TE_LARGEUR(...)`" largeur du Widget en pixel.
- "`CLASSE_PARAMETRE("cls")`" Extraction par type d'objet

– “DEBUT_BOITE(HORIZONTAL|VERTICALE [BoiteHomogene]), FIN_BOITE” disposition des Widgets dans la fenêtre.
– Des attributs booleens pouvant être préfixé par Non.
  – “Edite” la valeur du champ est éditable.
  – “Sauve”le champ doit être sauvegardé dans les fichiers.
  – “Affiche” le champ doit être affiché.
  – “Extrait” la valeur du champ est extraite d'un paramètre de l'objet, cela sera détaillé plus tard.
  – “Obligatoire” le champs est obligatoirement Extrait.
  – “Label” un label est indiqué à gauche du champ.
  – “InitChaine” lors de la création de l'objet le premier champ contenant cette option est initialisé avec la chaîne de caractère de la fenêtre message.
  – “Init3d” lors de la création de l'objet le premier champ contenant cette option est initialisé à la position du curseur. Sinon, cette option permet de décider des champs dont la valeur sera modifiée interactivement en 3D en utilisant les touches '1', '2', ...
  – “Cree” un widget doit être créé mais non-visible
  – “Initialiser” ce champs doit être initialisé à la création de l'objet (option par défaut).
– “TYPE_DANS_FILS(_...)” pour indiqué dans le cas d'une extraction que l'on veut extraire une donnée différente de son propre type logique (Utilisé pour faire des objets traducteurs de type comme point_triplet.

Remarques :
– Les instructions décrivant un champ peuvent être mise en commun pour un ensemble de champs en les indiquant en dehors d'un champ. Elles s'applique alors sur les champs suivants.
– L'ordre des Widgets dans la fenêtre est l'ordre des champs.

5 Pour allez plus loin

Quelques possibilités intéressantes.

5.1 Destruction d'un objet

Il est nécessaire de créer une procédure de destruction d'objet si des ressources (mémoire, fichier, ...) ont été allouées par les fonctions d'affichage ou bien de changement. *Il n'est pas nécessaire de s'occuper des champs, NanoÉdit les connait, il peut donc les détruire tous seul*

L'exemple suivant lance une commande UNIX puis, à chaque affichage récupère ce que la commande UNIX lui donne. Evidemment, lors de la destruction de l'objet, il faut arrêter le processus (“o_destruction.c”) :

```
#include <GL/gl.h>
#include "o_objet.h"
#include "s_string.h"
#include "u_graphique.h"
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/wait.h>
struct delet
{
  int pid;                  /* PID du processus lancé */
  int pipe[2];              /* Pipe de communication */
  FILE *entree;             /* Fichier lisant dans le pipe */
};
```

```c
/* Cette fonction d'affichage va afficher a chaque fois
 * une chaine de caractere differentes.
 * Cette chaine de caractere vient d'une commande UNIX. (ls -lsR /) */
static void affichage_ls(struct delet *o)
{
  Buffer ligne;

  if ( o->pid==0 )
    {
      /* PREMIER AFFICHAGE DE L'OBJET SEULEMENT
       * Ouvre un pipe pour communiquer et lance le processus */
      pipe(o->pipe);
      /* Lance le processus */
      o->pid = fork();
      if ( o->pid == 0 )
        {
          /* Dans le fils */
          dup2(o->pipe[1],1); /* La sortie standard de la suite = pipe */
          execlp("ls","ls","-R","/",NULL);
          exit(1);
        }
      /* Dans le pere */
      o->entree = fdopen(o->pipe[0],"r");
    }
  /* On affiche un texte différent à chaque affichage! */
  fgets(ligne,sizeof(ligne),o->entree);
  glRasterPos3f(0.,0.,0.);
  t_chaine( ligne );
}
/* Destruction de l'objet */
static void destruction(struct delet *o)
{
  int status;

  if ( o->pid )         /* + Fiable */
    {
      kill(o->pid, SIGTERM);
      waitpid(o->pid, &status, 0); /* Evite le Zombi */
      fclose(o->entree);
      close(o->pipe[0]);
      close(o->pipe[1]);
    }
}
/*
 *
 */
CLASSE(destruction, struct delet,
       DESTRUCTION(destruction)
       CHAMP_VIRTUEL(L_affiche_gl(affichage_ls))
       )
```

5.2 Méthodes sur une zone graphique

La zone GL permet d'associer des méthodes aux événements. Le prototype C de la méthode est :

`static valeur_retournée methode_widget(const Interface_widget *iw)`

Le contenu de la structure `Interface_widget` est :

```c
typedef struct
{
  void *objet ;                /* L'objet recevant l'événement */
  const Champ *ch ;            /* Le champ (Widget) recevant l'événement */
  Widget w ;                   /* Le Widget recevant l'événement */
```

```
  int b ;                    /* 0: Bouton gauche, 1: milieu, 2: droite */
  int  x, y ;                /* Position du curseur */
  const char *texte ;        /* Texte (touche clavier, ...) */
} Interface_widget ;
```

Les méthodes sont :

Événement	Option champ	Retour	Champ	Commentaire
Clavier	`TE_CLAVIER`	`void`	texte	La description de la touche tapée
Affichage	`TE_ECRIT`	`void`		On doit faire l'affichage
Aide	`TE_AIDE`	`const char*`	x, y	On retourne le message d'aide
Bouton	`TE_ENTREE`	`Booleen`	b, x, y	On retourne “Vrai” pour appeler la méthode “mouvement” continuellement
Mouvement	`TE_MOUVEMENT`	`void`	x, y	Position courante du curseur

Voir “o_s18.c” pour l'utilisation.

5.3 Mise à jour d'un objet par un événement extérieur à l'éditeur

Par exemple l'affichage de l'heure toutes les secondes. Pour activer la mise à jour, il faut rendre actif l'objet “date”. “o_date.c” :

```
#include <time.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "u_xt_interface.h"
#include "o_objet.h"

struct dada
{
  int secondes ;              /* Nombre de secondes depuis la création */
  int start ;                 /* Date de création */
  int          pid ;          /* Pour pouvoir tuer date */
  int          pipe[2] ;      /* Pour lire la date */
} ;
/* Cette fonction est appelée chaque fois qu'il y a quelque chose
 * à lire dans le fichier */
static Parcour chose_a_lire(void* objet, int fildes)
{
  time_t date ;
  struct dada *o ;

  o = objet ;
  read(o->pipe[0],&date,sizeof(date)) ; /* Lit la date */
  o->secondes = time(NULL) - o->start ;
  /* D'habitude on indique pas que l'on change des champs de l'objets.
   * En effet, le changement est la conséquence d'un autre changement
   * d'un autre champ.
   * Ici, le champ se modifie de lui-même, il faut prévenir NanoÉdit.
   */
  NOTE_UN_CHAMP_CHANGE(o) ;
  /*
   * Il faut ensuite propager le changement pour tout mettre à jour.
   */
  if ( ACTIF(o) ) /* Pas tout le temps sinon c'est trop consommateur CPU */
    o_appel_changement_sommet(o) ;
  return(Continue) ;
}
```

```
static void changement(struct dada *o)
{
  if ( o->pid || CREATION(o) )    /* Le processus est déjà lancé */
    return;

  pipe(o->pipe);              /* Ouvre le fichier de communication */
  o->pid = fork();            /* Création d'un nouveau processus */
  if ( o->pid )
    {
      /* Pere :
       * On demande de nous prévenir en cas
       * d'arrivé de données dans le pipe. */
      xt_ajoute_fichier(o, chose_a_lire, o->pipe[0]);
      o->start = time(NULL);
    }
  else
    {
      time_t date;
      /* fils */
      for(;;)                 /* Le process est tué par un kill */
        {
          time(&date);        /* Récupere le temps */
          write(o->pipe[1], &date,sizeof(date)); /* Envoit le temps */
          sleep(1);           /* Attend un peu */
        }
    }
}

static void destruction(struct dada *o)
{
  int status;

  if ( o->pid )
    {
      kill(o->pid,SIGTERM); /* On tue le processus */
      waitpid(o->pid,&status,0); /* On attend sa mort */
      close(o->pipe[0]);
      close(o->pipe[1]);
      xt_enleve_fichier(o, chose_a_lire, o->pipe[0]);
      o->pid = 0;
    }
}

CLASSE(date, struct dada,
       CHAMP(secondes, L_nombre P_entier Affiche)
       CHANGEMENT(changement)
       DESTRUCTION(destruction)
       )
```

5.4 Héritage

Comme vu précédemment, la classe est définie par langage, indiquer une sur-classe revient à interpréter la définition de la sur-classe avant la sienne.

La surclasse est indiquée dans la définition de la classe par son nom : `SUR_CLASSE("nom surclasse")`. On hérite de *toutes* les informations de la surclasse y compris les champs et l'état courant.

La plupart du temps on ne veut pas hériter des champs, pour cela on indique dans la définition de la classe `HERITE_PAS_DES_CHAMPS`. Si néanmoins l'on veut hériter des champs, il faut que la structure C utilisée contienne comme premier champ la structure C de la sur-classe.

5.5 Débuggage

Si vous voulez tracer le fonctionnement de l'un de vos noeud, il vous suffit de faire :

```
if ( TRACE(objet) )
   {
   eprintf("Message de trace simple\n") ;
   }
if ( TRACE2(objet) )
   {
   eprintf("Message de trace détaillé\n") ;
   }
```

5.6 Paquet et Décalage

Dans la définition de la classe vous pouvez indiquer `PAQUET(taille)` et `DECALAGE(nombre)`. La valeur par défaut des deux champs est `NB_NON_DEFINI`. La taille de paquet peut prendre la valeur `NB_ACTIFS` qui indique que le noeud prend tous les actifs comme paramètre.

Par défaut la taille du paquet est déduite des arguments d'ou l'on extrait des valeurs et le décalage a la même valeur que le paquet.

Remarque : le segment de droite permet de faire une ligne brisée, le décalage n'est donc pas égale à la taille du paquet.

5.7 Calcul en arrière plan

Pour les gens qui en ont besoin, on peut faire des calculs en arrière plan tout en continuant à travailler et même à modifier les données sur lesquelles sont fait les calculs. L'exemple est celui d'un système de particule ou l'on fait l'affichage graphique à chaque étape du calcul.

Pour ceux qui veulent entrer leur propre graphe, la syntaxe est la suivante : une liste de noeud, un noeud étant une liste d'indice de noeud connecté suivie des coordonnées du point ("OBJETS/o_particule.c") :

```
#include <GL/gl.h>
#include <math.h>
#include "u_xt_interface.h"
#include "t_geometrie.h"
#include "u_graphique.h"
#include "u_repartiteur.h"
#include "t_gl.h"

#define BASETAG 1000       /* Numéro d'identification du premier atome */
#define D 3                /* Dimension de l'espace */
#define POS(i)  (o->g.noeuds[i].donnee.table)
#define VIT(i)  (o->g.noeuds[i].donnee.table+D)
#define DIM(PROG) do { int dim; for(dim=0; dim<D; dim++) { PROG; } } while(0)

struct systeme_particule
{
  t_graphe_N_16_float g;          /* Le graphe de particules */
  Flottant de, dt, k, f;
};
/*
 * Affichage du système de particules
 */
static void affiche_particules(struct systeme_particule *o)
{
  int i,j,k;
  Triplet p;

  /* Affiche les arêtes */
```

```
  glBegin( GL_LINES ) ;
  for(i=0;i<o->g.nb_noeuds;i++)
    {
      for(j=0;j<o->g.noeuds[i].nb_aretes;j++)
        {
          k = o->g.noeuds[i].aretes[j].destination ;
          if ( k < i )
            {
              glVertex3fv(POS(i)) ;
              glVertex3fv(POS(k)) ;
            }
        }
    }
  glEnd() ;
  /* Affiche les sommets */
  if ( ACTIF(o) )
    {
      /* Si le système est actif, on trace des tag sur les points
       * pour permettre à l'utilisateur de les déplacer */
      for(i=0;i<o->g.nb_noeuds;i++)
        {
          DIM( (&p.x)[dim] = POS(i)[dim] ) ;
          t_tag(BASETAG+i, &p) ;
        }
    }
  else
    {
      glBegin(GL_POINTS) ;
      for(i=0;i<o->g.nb_noeuds;i++)
        glVertex3fv(POS(j)) ;
      glEnd() ;
    }
}
/* L'utilisateur déplace un tag */
static void ecrit_tag(struct systeme_particule *o,const Champ *c,const char *v)
{
  int i ;

  i = lit_numero_tag(c) - BASETAG ;
  if ( i >= 0 && i < o->g.nb_noeuds )
    sscanf(v, "%g%g%g", POS(i), POS(i)+1, POS(i)+2) ;
}
/* Lecture de la position d'un tag */
static const char * lit_tag(struct systeme_particule *o, const Champ *c)
{
  static char buf[99] ;
  int i ;

  i = c->parametre - BASETAG ;
  if ( i >= 0 && i < o->g.nb_noeuds )
    sprintf(buf, "%g %g %g", POS(i)[0], POS(i)[1], POS(i)[2]) ;
  else
    strcpy(buf, "") ;
  return(buf) ;
}
/* Déplacement des particules */
static Booleen u_calcul(void *objet)
{
  struct systeme_particule *o = objet ;
  float d, n, force[D], somme[D] ;
  int i, j, k ;

  for(i=0;i<o->g.nb_noeuds;i++)
    {
```

```
      DIM( somme[dim] = 0. ) ;
      for(j=0;j<o->g.noeuds[i].nb_aretes;j++)
        {
          k = o->g.noeuds[i].aretes[j].destination ;
          n = 0. ;
          DIM( force[dim] = POS(k)[dim] - POS(i)[dim] ;
               n += force[dim] * force[dim] ; ) ;
          d  = o->k*(sqrt(n) - o->de) ;
          DIM( somme[dim] += force[dim] * d ) ;
        }
      DIM( VIT(i)[dim] += (somme[dim] - VIT(i)[dim]*o->f) * o->dt ) ;
    }
  for(i=0;i<o->g.nb_noeuds;i++)
    DIM( POS(i)[dim] += VIT(i)[dim]*o->dt ) ;
  /* Demande à l'éditeur de mettre tout à jour (notamment afficher) */
  NOTE_UN_CHAMP_CHANGE(o) ;
  o_appel_changement_sommet(o) ;

  return(Vrai) ; /* Continuer en arrière plan */
}

static Booleen arrete_systeme(const Interface_widget *iw)
{
  u_enleve_action(u_calcul, iw->objet) ;
  return(Faux) ;
}
static Booleen demarre_systeme(const Interface_widget *iw)
{
  u_ajoute_action(PRIORITE_ARRIERE_PLAN, u_calcul, iw->objet) ;
  return(Faux) ;
}
static void destruction_systeme(struct systeme_particule *o)
{
  u_enleve_action(u_calcul, o) ;
}

CLASSE(systeme_particule, struct systeme_particule,
       Edite Sauve Extrait
       CHAMP(g, L_graphe_de_points P_graphe_N_l6_float TE_HAUTEUR(200))
       CHAMP(de, LABEL("Distance"  ) L_nombre P_flottant DEFAUT("1"   ))
       CHAMP(dt, LABEL("Delta t"   ) L_nombre P_flottant DEFAUT("0.01"))
       CHAMP(k , LABEL("Raideur"   ) L_nombre P_flottant DEFAUT("1"   ))
       CHAMP(f , LABEL("Frottement") L_nombre P_flottant DEFAUT("0.1" ))
       CHAMP_VIRTUEL(LABEL("Demarre") L_bouton(demarre_systeme)) /**/
       CHAMP_VIRTUEL(LABEL("Arrete") L_bouton(arrete_systeme)) /**/
       CHAMP_VIRTUEL(L_affiche_gl(affiche_particules))
       CHAMP_VIRTUEL(L_tag(ecrit_tag,lit_tag))
       DESTRUCTION(destruction_systeme)
       MENU("Simulateurs/Système de particules")
       EVENEMENT("SP")
       )
```

Un autre exemple de calcul en arrière plan peut être trouvé dans l'objet "nuage", en effet le chargement du fichier ce fait 10000 lignes par 10000 lignes avec un réaffichage toute les secondes.

5.8 Configuration de l'affichage

On peut utiliser NanoÉdit avec Motif ou GTK, le changement nécessite une recompilation totale. Le choix est fait dans le script "configuration"

5.8.1 Avec Motif

Il est possible de modifier l'apparence de la fenêtre d'édition de l'objet en utilisant des ressources X11 (je vous renvoie alors sur la documentation X11 et les commandes `editres` et `xrdb`). Par exemple en créant ou éditant le fichier `NanoEdit.ad` dans le répertoire contenant le source de votre objet :

```
NanoEdit.edite.segment1*label_classe.fontList: -*-*-medium-r-*-*-*-180-*-*-m-*-iso8859-1
NanoEdit.edite.segment1*classe.background: Red
NanoEdit.edite.segment1*actif.borderWidth: 10
```

5.8.2 Avec GTK

Vous éditez un fichier nommé `gtkrc` dans le repertoire contenant le source de votre objet :

```
style "grosse-fonte" {
font = "-*-*-bold-r-*-*-*-180-*-*-m-*-iso8859-1"
}
style "fond-rouge" {
bg[NORMAL] = {1.0,0,0}
}

widget "segment1*label_classe" style "grosse-fonte"
widget "segment1*trace"        style "fond-rouge"
```

5.9 Makefile

Quelques buts utiles :

- "depends" recalcul les dépendances (si vous avez des '#include')
- "tags" calcul le fichier de TAGS, celui permet lorsque l'on est sous EMACS d'aller instantanément sur la définition d'une fonction. On positionne le curseur sur l'appel de la fonction et on tape 'M-.'

5.10 Edition des liens dynamique

Elle n'est faite que pour les fichier objets (commençant par "o_') et malheureusement, je n'arrive pas à faire en sorte que sur toute les machines on puisse appeler une fonction d'un fichier objet vers un autre fichier objet. C'est pour cela que je conseille de déclarer TOUTES les fonctions des fichiers "o_" en "static" pour ne pas l'oublier.