Generic Implementation of Free Form Deformation Techniques

Carole Blanc Christophe Schlick

LaBRI [1]*, 351 cours de la libération, 33405 Talence (FRANCE)*
[blanc|schlick]@labri.u-bordeaux.fr

1 Introduction

Sederberg and Parry have introduced the Free Form Deformations (FFD for short) technique [SEDE86]. Intuitively, the idea of FFD is to embed the object that has to be deformed in a parallelpiped made of clay. Then, any deformation applied to the clay will be transmitted to the included object in a similar way. Mathematically, the clay is represented by a volume of threedimensional splines and the parallelpiped is a 3D grid (oriented along the three coordinate axis) of control points that can be moved to generate deformations. The use of splines enables the deformation to be very intuitive according to the interactive displacement of control points. Another advantage of FFD is that they can be applied either globally on the whole object or locally by putting control points only around a given part of the object.

Coquillart has later proposed an extension of the FFD technique [COQU90], that she logically called Extended Free Form Deformations (EFFD). Her intention was to remove one of the limitations of the original FFD algorithm. Indeed, due to the parallelpipedical shape of the lattice, FFD do not provide deformations of arbitrary shapes. Coquillart showed how to use non-parallelpipedical (i.e. cylindrical, prismatical, spherical or even arbitrary shaped) lattices, in order to control precisely the shape of the deformed part of the object. Consequently, a much greater variety of deformations can be obtained. As a counter part, with EFFD the link between the grid and the object is not straightforward as it is with FFD, therefore intensive use of numerical techniques (Newton iterations, for instance) may be needed to compute the deformations.

2 Description

The present paper proposes a generic implementation of FFD and EFFD techniques. The term "generic" focuses on the fact that the implementation depends neither on a given geometric model for surfaces nor on a specific data structures for internal representation. Each deformation routine acts only on points : it takes the coordinates of a point on the original object, computes its displacement according to the deformed lattice, and finally returns the new coordinates of the point.

More precisely for a given point P that has to be deformed, a FFD algorithm realizes three successive steps. First, the initial coordinates of P (usually expressed in the world frame) are converted to the coordinate system in which the grid is defined. Second, the algorithm computes the indices (i, j, k) of the voxel in which P belongs to, as well as the local coordinates (u, v, w) of P relative to the voxel. Finally (i, j, k) are used to find the control points concerned with that voxel, and (u, v, w) become the parameters of a trivariate interpolation/approximation between these control points.

For each of the three steps, several algorithms may be used. Consequently, combining these algorithms together provides many different ways for doing FFD. In our implementation, we propose three alternatives for the first step, three for the second one and two for the last one, which means 18 different FFD techniques. Next sections detail more precisely the alternatives that we have chosen.

[1] Laboratoire Bordelais de Recherche en Informatique (*Université Bordeaux I* and *Centre National de la Recherche Scientifique*). The present work is also granted by the *Conseil Régional d'Aquitaine*.

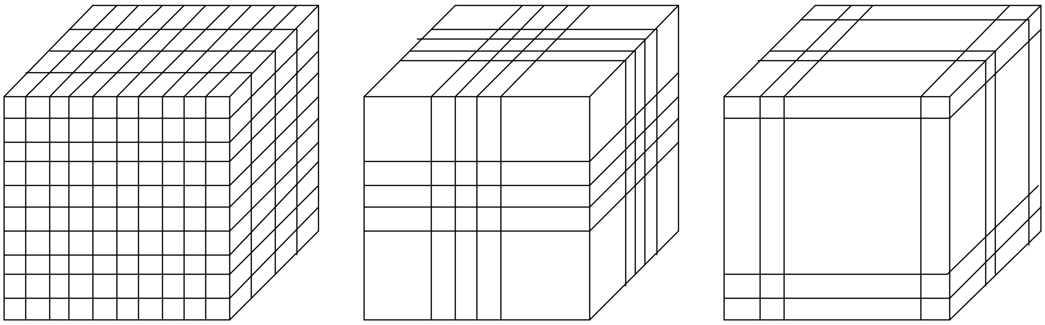

Figure 1: (a) canonic-grid (b) partial-grid (c) general-grid

3 Coordinate Representation

In the original FFD implementation, the undeformed grid was a 3D parallelpiped oriented along the three coordinate axis, with the origin at one vertex. A more natural solution is to put the origin at the center of the grid. Moreover using a *unit-cube frame* [BLAN95] enables to be independent of the position, orientation and scale of the object embedded in the grid. On the same idea, the conversion routines, between unit-cube, unit-cylinder and unit-sphere frames — see `car_to_cyl, cyl_to_car, car_to_sph, sph_to_car` defined elsewhere in the present volume [BLAN95] — allow to create with almost the same cost, cylindrical or spherical grids for doing EFFD. But notice that the vertex coordinates of the deformed grid have always to be given in the same representation as the initial grid.

4 Grid Type

The second step of the FFD algorithm has to compute the indices (i, j, k) of the voxel surrounding the point, and the local coordinates (u, v, w) of the point in the voxel. This calculation is relatively easy and depends only on the way the grid is subdivided. We have implemented three possible grid structures (see Figure 1). In the first one (see `canonic-grid`) the whole grid (ie unit-cube with our coordinates) is regularly sampled in l, m or n voxels according to the axis. In the second one (see `partial-grid`) only a subset of the unit-cube (defined by a bounding box) is regularly sampled. This can be used, for instance, to create local deformations. The last structures is the most general (see `general-grid`). It enables to defines precisely the voxel boundaries (using three array of coordinates `X, Y` and `Z`) and thus to create non-uniform sampling of the grid.

5 Trivariate Interpolation

Once (i, j, k) and (u, v, w) have been found the last step is to compute the actual deformation of the point according to the deformed grid. Usually the grid vertices are considered as the control points of trivariate splines using (u, v, w) as parameters. Several kind of splines have been used with a FFD scheme : Bezier splines of arbitrary degree [SEDE86], piece-wise cubic Bezier splines [COQU90], B-splines with a quasi-uniform [2] knot vector [PURG89], or NURBS [THAL92].

We propose here another solution using trivariate cardinal interpolating splines (using the 64 vertices surrounding the point) which seems to be a good alternative in terms of power/speed tradeoff. Indeed cardinal splines are the simplest cubic splines in terms of computation; and when using natural cardinal splines, no additional information is required for the ending condition [FOLE90] which leads also to the simplest splines in terms of memory cost. Moreover the oscillation problem, which usually presents the use of cardinal splines for direct surface modeling, does not matter here thanks to the indirect action of the splines on the object. Therefore, we really think that cardinal splines represent an efficient

[2] A quasi-uniform Bspline has a uniform knot vector except at the boundaries where there are multiple knots which enable to interpolate the first and the last points

solution for FFD (see `cerp-voxel`). Notice that there is even a cheaper alternative for computing the deformation of the point ; it consists in a simple trilinear interpolation between the 8 vertices of the voxel (see `lerp-voxel`). This solution has been completely neglegted in the FFD literature, although it may provide interesting effects (e.g. sharp-edged deformation).

6 Conclusion

By grouping all the different elements detailled above (cartesian, cylindrical or spherical coordinates, canonic, partial or general grids, linear or cardinal interpolation) we obtain in fact 18 different FFD and EFFD techniques, each of them having its own interest.

7 Source Code

```
/* ------------------------------------------------------------------------ *\
   FFD.H : Carole Blanc (8 June 1994)

   This package provides an implementation of 18 different algorithms
   for doing Free Form Deformations and Extended Free Form Deformations.

\* ------------------------------------------------------------------------ */

#ifndef _FFD_
#define  _FFD_

/*
** This package uses the "Toolbox of Macros Functions for Computer Graphics"
** which provides files : tool.h, real.h, uint.h, sint.h, vec?.h and mat?.h
**
** This package uses the "Easy Transformations between Cartesian, Cylindrical
** and Spherical Coordinates" which provides files : frame.h and frame.c
*/
#include "real.h"
#include "uint.h"
#include "frame.h"

extern void car_canonic_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void car_canonic_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void car_partial_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
extern void car_partial_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
extern void car_general_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
extern void car_general_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
extern void cyl_canonic_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void cyl_canonic_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void cyl_partial_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
extern void cyl_partial_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
```

```
extern void cyl_general_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
extern void cyl_general_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
extern void sph_canonic_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void sph_canonic_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Grid);
extern void sph_partial_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
extern void sph_partial_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              realvec3 *Bound, realvec3 *Grid);
extern void sph_general_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
extern void sph_general_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                              real *X, real *Y, real *Z, realvec3 *Grid);
#endif

/* ------------------------------------------------------------------------ *\
   FFD.C : Carole Blanc (8 June 1994)

   This package provides an implementation of 18 different algorithms
   for doing Free Form Deformations and Extended Free Form Deformations.

\* ------------------------------------------------------------------------ */

#include "ffd.h"
#include "vec3.h"

static realvec3 MaxPoint = {+1.0, +1.0, +1.0};
static realvec3 MinPoint = {-1.0, -1.0, -1.0};

/* ------------------------------------------------------------------------ *\
                          GRID AND VOXEL COORDINATES
\* ------------------------------------------------------------------------ */

/*
** canonic_grid :
**
** Input  : Point = coordinates (in [-1,1]) of the point in the local frame
**          Size  = number of voxels of the grid in the three dimensions
** Output : Point = integer part = coordinates of the voxel in the grid
**                  fractional part = coordinates of the point in the voxel
*/
bool canonic_grid (realvec3 *Point, uintvec3 *Size)
{
  if (OUT_VEC3 (*Point, MinPoint, MaxPoint)) return (FALSE);
  Point->x = Size->x * (Point->x + 1.0) / (2.0 + TOL);
  Point->y = Size->y * (Point->y + 1.0) / (2.0 + TOL);
  Point->z = Size->z * (Point->z + 1.0) / (2.0 + TOL);
  return (TRUE);
}

/*
** partial_grid :
**
** Input  : Point = coordinates (in [-1,1]) of the point in the local frame
```

```
**          Size  = number of voxels of the grid in the three dimensions
**          Bound = values of the grid boundaries (min/max) for the three axis
** Output : Point = integer part = coordinates of the voxel in the grid
**                  fractional part = coordinates of the point in the voxel
*/
bool partial_grid (realvec3 *Point, uintvec3 *Size, realvec3 *Bound)
{
  if (OUT_VEC3 (*Point, Bound[0], Bound[1])) return (FALSE);
  Point->x = Size->x * (Point->x-Bound[0].x)/(Bound[1].x-Bound[0].x+TOL);
  Point->y = Size->y * (Point->y-Bound[0].y)/(Bound[1].y-Bound[0].y+TOL);
  Point->z = Size->z * (Point->z-Bound[0].z)/(Bound[1].z-Bound[0].z+TOL);
  return (TRUE);
}

/*
** general_grid :
**
** Input  : Point = coordinates (in [-1,1]) of the point in the local frame
**          Size  = number of voxels of the grid in the three dimensions
**          X,Y,Z = array of voxel boundary values for the three axis
** Output : Point = integer part = coordinates of the voxel in the grid
**                  fractional part = coordinates of the point in the voxel
*/
bool general_grid (realvec3 *Point, uintvec3 *Size, real *X, real *Y, real *Z)
{
  realvec3 Bound[2];
  register uint p;

  MAKE_VEC3 (Bound[0], X[0], Y[0], Z[0]);
  MAKE_VEC3 (Bound[1], X[Size->x], Y[Size->y], Z[Size->z]);
  if (OUT_VEC3 (*Point, Bound[0], Bound[1])) return (FALSE);
  for (p = 0; p < Size->x; p++)
    if (Point->x < X[p+1]) {
        Point->x = p + (Point->x - X[p]) / (X[p+1] - X[p] + TOL); break;
    }
  for (p = 0; p < Size->y; p++)
    if (Point->y < Y[p+1]) {
        Point->y = p + (Point->y - Y[p]) / (Y[p+1] - Y[p] + TOL); break;
    }
  for (p = 0; p < Size->z; p++)
    if (Point->z < Z[p+1]) {
        Point->z = p + (Point->z - Z[p]) / (Z[p+1] - Z[p] + TOL); break;
    }
  return (TRUE);
}

/* ------------------------------------------------------------------------ *\
                          TRIVARIATE INTERPOLATIONS
\* ------------------------------------------------------------------------ */

/*
** lerp_voxel : Triple linear interpolation
**
** Input  : Point = integer part = coordinates of the voxel in the grid
**                  fractional part = coordinates of the point in the voxel
**          Grid  = coordinates of the deformed grid vertices in a given frame
** Output : Point = coordinates of the deformed point in the same frame
```

```
*/
void lerp_voxel (realvec3 *Point, uintvec3 *Size, realvec3 *Grid)
{
  realvec3 Coord, PointA, PointB;
  uintvec3 Voxel;
  uint     Index;

/*
** Find the coordinates of the voxel in the grid and of the point in the voxel
*/
  COPY_VEC3 (Voxel, *Point);
  SUB_VEC3 (Coord, *Point, Voxel);
/*
** Interpolation between the four points on the floor of the voxel
*/
  Index = Voxel.x + (1 + Size->x) * (Voxel.y + (1 + Size->y) * Voxel.z);
  LERP_VEC3 (PointA, Coord.x, Grid[Index], Grid[Index+1]);
  Index += 1 + Size->x;
  LERP_VEC3 (PointB, Coord.x, Grid[Index], Grid[Index+1]);
  LERP_VEC3 (*Point, Coord.y, PointA, PointB);
/*
** Interpolation between the four points on the ceiling of the voxel
*/
  Index += Size->x * (1 + Size->y);
  LERP_VEC3 (PointA, Coord.x, Grid[Index], Grid[Index+1]);
  Index += 1 + Size->x;
  LERP_VEC3 (PointB, Coord.x, Grid[Index], Grid[Index+1]);
  LERP_VEC3 (PointA, Coord.y, PointA, PointB);
/*
** Interpolation between the floor and the ceiling of the voxel
*/
  LERP_VEC3 (*Point, Coord.z, *Point, PointA);
}

/*
** This macro function computes the coefficient of the cardinal interpolation
** polynomial (for one coordinate) according to the position of grid vertices.
** When the current voxel is at the grid boundary, one additional vertice is
** computed for each dimension, in order to create natural cardinal splines.
*/
#define CARDINAL(i,n) {\
          a = 0.5 * Coord.n; b = a + Coord.n; Coord.n *= a;\
          c = Coord.n * (2.0 - a - a); d = Coord.n * (4.0 - b - b);\
          Coeff[0].n = c - a; Delta[0].n = -i;\
          Coeff[1].n = 1.0 - d - Coord.n; Delta[1].n = 0;\
          Coeff[2].n = a + d; Delta[2].n =  i;\
          Coeff[3].n = Coord.n - c; Delta[3].n = i + i;\
          if (Voxel.n == 0) {\
            Coeff[2].n -= Coeff[0].n;\
            Coeff[1].n += Coeff[0].n + Coeff[0].n;\
            Coeff[0].n = 0.0; Delta[0].n = 0;\
          }\
          if (Voxel.n == Size->n - 1) {\
            Coeff[1].n -= Coeff[3].n;\
            Coeff[2].n += Coeff[3].n + Coeff[3].n;\
            Coeff[3].n = 0.0; Delta[3].n = i;\
          }\
```

```
        }
/*
** cerp_voxel : Triple cardinal interpolation
**
** Input  : Point = integer part = coordinates of the voxel in the grid
**                  fractional part = coordinates of the point in the voxel
**          Grid  = coordinates of the deformed grid vertices in a given frame
** Output : Point = coordinates of the deformed point in the same frame
*/
void cerp_voxel (realvec3 *Point, uintvec3 *Size, realvec3 *Grid)
{
  realvec3 Coord, Coeff[4];
  uintvec3 Voxel, Delta[4];
  uint     i, j, k, Base, Index;
  real     a, b, c, d, Real;
/*
** Find the coordinates of the voxel in the grid and of the point in the voxel
*/
  COPY_VEC3 (Voxel, *Point); SUB_VEC3 (Coord, *Point, Voxel);
/*
** Compute the coefficients (Coeff[0]...Coeff[3]) of the cardinal polynomial
** and the indices (Delta[0]...Delta[3]) to acces to the 64 control points
*/
  Index = 1; CARDINAL (Index, x);
  Index = 1 + Size->x; CARDINAL (Index, y);
  Index = (1 + Size->x) * (1 + Size->y); CARDINAL (Index, z);
/*
** Now we can do the triple loop
*/
  MAKE_VEC3 (*Point, 0.0, 0.0, 0.0);
  Base = Voxel.x + (1 + Size->x) * (Voxel.y + (1 + Size->y) * Voxel.z);
  for (k = 0; k < 4; k++)
    for (j = 0; j < 4; j++)
      for (i = 0; i < 4; i++) {
        Index = Base + Delta[i].x + Delta[j].y + Delta[k].z;
        Real = Coeff[i].x * Coeff[j].y * Coeff[k].z;
        MULS_VEC3 (Coord, Grid[Index], Real);
        INC_VEC3 (*Point, Coord);
      }
}

/* ------------------------------------------------------------------------ *\
                            FREE-FORM DEFORMATIONS
\* ------------------------------------------------------------------------ */

/*
** Here are the 18 functions that define 18 different FFD techniques.
**
** Each function is obtained by a specific combination of three elements :
** - Coordinate representation for the grid (cartesian, cylindrical, spherical)
** - Type of the grid (canonic, partial, general)
** - Type of the trivariate interpolation in the grid (linear, cardinal)
**
** Input : Point = World cartesian coordinates of the point before deformation
**         Frame = Local frame in which the grid is defined
**         Size  = number of voxels of the grid in the three dimensions
**         Bound = min and max coordinates for grid (only for xxx_partial_xxx)
```

```
**          X,Y,Z = array of voxel boundaries (only for xxx_general_xxx)
**          Grid  = coordinates of the deformed grid vertices in the local frame
**
** Output : Point = World cartesian coordinates of the point after deformation
*/
void car_canonic_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (canonic_grid (Point, Size)) lerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

void car_canonic_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (canonic_grid (Point, Size)) cerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

void car_partial_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       realvec3 *Bound, realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (partial_grid (Point, Size, Bound)) lerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

void car_partial_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       realvec3 *Bound, realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (partial_grid (Point, Size, Bound)) cerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

void car_general_lerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       real *X, real *Y, real *Z, realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (general_grid (Point, Size, X, Y, Z)) lerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

void car_general_cerp (realvec3 *Point, frame Frame, uintvec3 *Size,
                       real *X, real *Y, real *Z, realvec3 *Grid)
{
  world_to_car (Point, Frame);
  if (general_grid (Point, Size, X, Y, Z)) cerp_voxel (Point, Size, Grid);
  car_to_world (Point, Frame);
}

/*
** cyl_canonic_lerp, cyl_canonic_cerp, cyl_partial_lerp, cyl_partial_cerp,
** cyl_general_lerp, cyl_general_cerp, sph_canonic_lerp, sph_canonic_cerp,
** sph_partial_lerp, sph_partial_cerp, sph_general_lerp, sph_general_cerp
```

```
** differ only from the previous ones by replacing the substring "car" by
** "cyl" or "sph". Therefore, they have been cancelled on this listing...
*/
```

8 References

[BLAN95] C.Blanc, and C. Schlick, *Easy transformations between cartesian, cylindrical and spherical coordinates*, Technical Report, 1995.

[COQU90] S. Coquillart, *EFFD : A Sculpturing Tool for 3D Geometric Modeling*, Computer Graphics, v24, n4, pp 183-191, 1990.

[FOLE90] J. Foley, and A. VanDam, and S. Feiner, and J. Hughes, *Computer graphics : principles and practice*, Addison-Wesley, 1990.

[PURG89] W. Purgathofer and J. Griessmair, *Deformation of solids with trivariate B-splines*, Proc. of Eurographics 89, pp 137-148, 1989.

[SCHLICK95] C. Schlick, *A toolbox of macro functions for computer graphics*, Graphics Gems V, Academic Press, 1995.

[SEDE86] T. Sederberg and S. Parry, *Free Form Deformations of Solid Geometric Models*, Computer Graphics, v20, n4, pp 151-160, 1986.

[THAL92] P. Kalra and A. Mangili and M-N. Thalmann and D. Thalmann, *Simulation of facial muscle actions based on rational free-form deformation*, Proc. of Eurographics 92, pp 59-69, 1992.