Accurate B-spline Free-Form Deformation of Polygonal Objects

Jieqing Feng*
State Key Lab. of CAD&CG
Zhejiang University
Hangzhou, 310027, P.R.China
Email: jqfeng@cad.zju.edu.cn

Pheng-Ann Heng
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: pheng@cse.cuhk.edu.hk

Tien-Tsin Wong
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
Email: ttwong@acm.org

Abstract

Free-form deformation is an important geometric shape modification method in computer animation and geometric modeling. We explore it by means of functional composition via shifting operators. Our method takes a polygonal model as input but yields a curved model described by triangular Bézier patches as output. The proposed method also solves the sample problem of free-form deformation.

keywords: Free-form deformation, functional composition, shifting operators, triangular Bézier patch

1 Introduction

In geometric modeling and computer animation, modification of geometric shape is an industrially important research topic. When an object is described by a small number of vertex or control points, the user can modify shapes by such means as pulling points, conducting transformations,*etc.* When the number of vertices or control points increases, direct manipulation methods no longer suffice. Space deformation can solve the problem well [2].

Barr[1] first proposed the global and local deformation concept, in which the transformation matrix is not constant, but varies according to the space position. That is to say, the deformation is represented as a matrix function of space position, independent of the geometric representation of object. In this setting Free-form Deformation (FFD) was first proposed by Sederberg and Parry[21]. It supplies a general deformation frame. The object is embedded into an intermediate space, regarded as a deformation tool. To deform the object, the user first deforms the embedding space; then the deformation is passed to the object. In [21] the intermediate space was a trivariate tensor product Bézier volume. After that, many extensions were proposed, such as B-spline FFD[14], Extended FFD[8], Rational FFD[16], NURBS FFD[17], continuous FFD[3], *etc.* All FFD methods share the same four processing steps:

1. Define a parametric volume, including parametric space and lattice,
2. Map object into parametric space of volume,
3. Modify shape of volume through editing control points in lattice,
4. Deform the object embedded in volume.

Other extensions include AFFD[9], direct manipulation FFD[15], arbitrary topological lattice FFD[19], Dirichlet FFD[20], which have minor differences in the above steps.

Unlike above FFD methods, which have 3D deformation tool, surface deformation[13], axial deformation [18, 7, 22] and constraints-based space deformation [5] adopt parametric surfaces, axial curve, points and

*Corresponding author

their displacements as deformation control tools respectively. Bechmann gives a detailed survey of FFD methods with different dimensional tools[4].

All of the space deformation methods mentioned above are independent of the geometric representation. Theoretically, the deformation should act on every point of object. But this is impossible, since the object can only be represented as discrete forms in the computer. In fact, only representative vertices or control points of the object are deformed. For an object described by parametric surface, the result object will still appear smooth when the deformation is conducted on the control points. But for a polyhedral object, when too few points are sampled, the actual deformation will be far from the theoretical result. In the worst case, the deformed object remains unchanged. Thus the sampling problem is important for practical use of FFD methods. Unfortunately, little research FFD addresses this problem, except for[14] on B-spline FFD. This work checks the middle point of each triangle mesh edge and evaluates the error between true deformation and linear interpolation. If the error is not less than a given threshold, the triangle is further subdivided at the midpoints. However, such a solution may fail at singular points, which are usually important to the deformation result.

In this paper, for objects represented by triangular meshes we solve this problem through functional composition via shifting operators, where the deformation tool is a B-spline volume. In the proposed method, the B-spline volume is first converted (using cutting planes determined by its knot vectors) to a piecewise continuous Bézier volume, in whose parametric space the object is then subdivided and re-triangulated. In the re-triangulated result, each triangle of the object mesh is within a Bézier volume. Finally, we conduct the functional composition via shifting operators for each Bézier volume. The result of the deformation is a set of triangular Bézier patches, whose degree is the sum of three directional degrees of the B-spline volume, coinciding exactly with the theoretical result of B-spline FFD. Thus the sample problem is solved at the same time.

The rest of this paper is organized as follows. Section 2 establishes notation, and introduces shifting operators, generalized de Casteljau algorithm and the definition of the B-spline volume is used as deformation tool. Section 3 gives the subdivision and re-triangulation algorithm for preprocessing the object to be deformed. In Section 4 a functional composition algorithm is introduced, which depends on shifting operators and the generalized de Casteljau algorithm. Finally, we give implementation results and our conclusions.

2 Preliminary

With the help of shifting operators, first introduced by Chang[6], Bernstein polynomials can be concisely expressed. Some properties of Bézier curves and surfaces can also be easily deduced. Let $\mathbf{r}(u,v,w)$ be a Bézier volume defined as follows:

$$\mathbf{r}(u,v,w) = \sum_{i=0}^{n_u}\sum_{j=0}^{n_v}\sum_{k=0}^{n_w} \mathbf{r}_{ijk} B_{i,n_u}(u) B_{j,n_v}(v) B_{k,n_w}(w) \tag{1}$$

where $\mathbf{r}_{ijk} \in \mathbf{R}^3$, $(u,v,w) \in [0,1]^3$. Define operators E_u, E_v, E_w, I by:

$$E_u\mathbf{r}_{ijk} = \mathbf{r}_{i+1,j,k} \quad E_v\mathbf{r}_{ijk} = \mathbf{r}_{i,j+1,k} \quad E_w\mathbf{r}_{ijk} = \mathbf{r}_{i,j,k+1} \quad I\mathbf{r}_{ijk} = \mathbf{r}_{ijk}$$

With this definition, (1) can be reformulated via binomial expansion as:

$$\mathbf{r}(u,v,w) = [(1-u)I + uE_u]^{n_u}[(1-v)I + vE_v]^{n_v}[(1-w)I + wE_w]^{n_w}\mathbf{r}_{000} \tag{2}$$

Let $\mathbf{c}(u)$ be a Bézier curve. Its expression in terms of shifting operators is:

$$\mathbf{c}(u) = [(1-u)I + uE]^n\mathbf{c}_0 \qquad\qquad u \in [0,1] \tag{3}$$

Then a portion of the curve restricted on interval $[u_0,u_1] \subseteq [0,1]$ is still a Bézier curve. The generalized de Casteljau algorithm is designed for computing the control points of a sub-curve[12]. Here we express it via shifting operators. Let interval $[u_0,u_1]$ be parameterized as:

$$u(t) = (1-t)u_0 + tu_1 \qquad\qquad t \in [0,1]$$

Substituting this into (3) gives:

$$\begin{aligned}
\tilde{\mathbf{c}}(t) &= \mathbf{c}(u(t)) \\
&= [(1-u(t))I + u(t)E]^n\mathbf{c}_0 \\
&= [((1-u_0)I + u_0E)(1-t) + ((1-u_1)I + u_1E)t]^n\mathbf{c}_0 \\
&= \sum_{i=0}^{n} \tilde{\mathbf{c}}_i B_{i,n}(t)
\end{aligned}$$

where the control points: $\tilde{\mathbf{c}}_i = ((1-u_0)I+u_0E)^{n-i}((1-u_1)I+u_1E)^i\mathbf{c}_0$. This is the generalized de Casteljau algorithm in terms of shifting operators, fundamental to our proposed accurate FFD.

Next, we define the initial B-spline volume for the FFD. Let $[X_{min}, X_{max}]\times[Y_{min}, Y_{max}]\times[Z_{min}, Z_{max}]$ be the bounding box of the object to be deformed, in the object coordinate system. Then the user specifies, or the system automatically generates, knot vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ as follows:

$$\mathbf{u} = \{u_0, \cdots, u_{k_u}, u_{k_u+1}, \cdots, u_{n_u}, \cdots, u_{n_u+k_u}\}$$

where k_u is the u-degree of the B-spline volume, n_u is the number of its control points along the u-direction, and $u_0 = \cdots = u_{k_u} = X_{min}$, $u_{n_u} = \cdots = u_{n_u+k_u} = X_{max}$. The knot vectors $\mathbf{v}$ and $\mathbf{w}$ are defined similarly. The B-spline volume $\mathbf{P}(u, v, w)$ determined by $\mathbf{u}, \mathbf{v}, \mathbf{w}$ and with degree k_u, k_v, k_w is:

$$\mathbf{P}(u,v,w) = \sum_{i=0}^{n_u}\sum_{j=0}^{n_v}\sum_{k=0}^{n_w}\mathbf{P}_{ijk}N_{i,k_u}(u)N_{j,k_v}(v)N_{k,k_w}(w) \tag{4}$$

An advantage of the bounding box as the parametric domain of B-spline volume is that we need not map object points into the parametric space of $\mathbf{P}(u, v, w)$ while conducting FFD. According to the above definition, the point coordinates in object space can be used as local coordinates corresponding to $\mathbf{P}(u, v, w)$, saving computation time. Then the B-spline volume is converted to piecewise continuous Bézier volumes through a knot insertion algorithm[12], which is a black box for the user. In total, $(n_u - k_u)\cdot(n_v - k_v)\cdot(n_w - k_w)$ Bézier volumes are generated, defined in $[u_i, u_{i+1}]\times[v_j, v_{j+1}]\times[w_k, w_{k+1}]$, where $k_u \le i < n_u$, $k_v \le j < n_v$, $k_w \le k < n_w$.

3 Subdivision and re-triangulation of objects

In this paper, the object to be deformed is assumed to be expressed as a triangular mesh, because other polygonal faces rarely remain planar. Otherwise, the input object is triangulated first. The subdivision and re-triangulation in this section is performed in the parametric space of B-spline volume. Recalling the definition of knot vectors of B-spline volume, the parametric space is identical with the object space where the object is defined. For clarity, we rewrite knot vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ as $\mathbf{x}, \mathbf{y}, \mathbf{z}$ and $x_i = u_i$, $y_j = v_j$, $z_k = v_k$ for all possible i, j, k.

Let $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$ be a triangle of the object, as shown in Figure 1(a). The edges $\mathbf{P}_0\mathbf{P}_1$, $\mathbf{P}_1\mathbf{P}_2$, $\mathbf{P}_2\mathbf{P}_0$ can be parametrized as linear combination of its vertices. Unlike normalized $[0, 1]$ parameterization, we parameterize three edges as:

$$\begin{array}{lcll} \mathbf{e}_0(t) & = & (1-t)\mathbf{P}_0 + t\mathbf{P}_1 & t \in [0,1] \\ \mathbf{e}_1(t) & = & (2-t)\mathbf{P}_1 + (t-1)\mathbf{P}_2 & t \in [1,2] \\ \mathbf{e}_2(t) & = & (3-t)\mathbf{P}_2 + (t-2)\mathbf{P}_0 & t \in [2,3] \end{array}$$

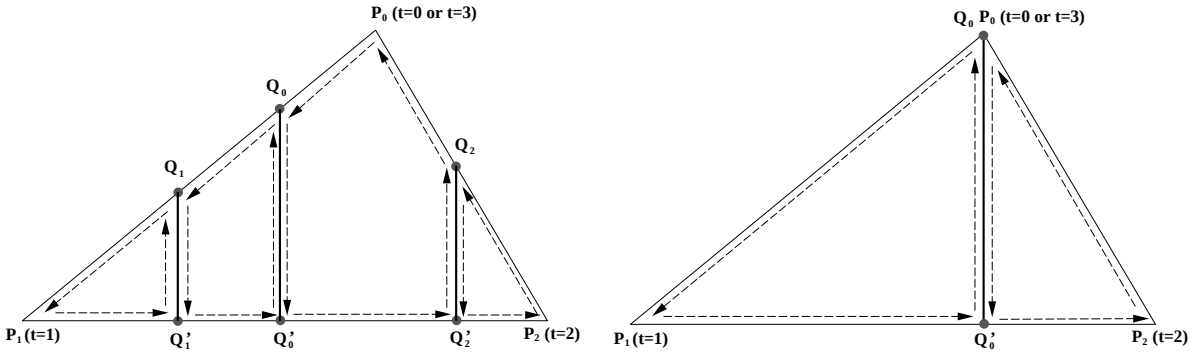
Assuming planes $x = x_i$ $(i = i_0, \cdots, i_1)$ cross the triangle $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$, our purpose is to subdivide and re-triangulate the triangle against the planes $x = x_i$. First we compute the intersections between each plane and triangle. Then we appropriately trace the triangle vertex and intersection points to form loops. Finally, for loops with more than four edges, we triangulate them. These three steps are then repeated for the y and z planes successively. We introduce them in detail in the following sections.

3.1 Computing intersections and generating circular linked lists

To make our algorithm clear, we introduce a data structure for both triangle vertices and intersection points:

```
struct vertex {
   float x,y,z; /* space position coordinate */
   float t;     /* t parameter corresponding to edge parameterization */
   short int Vflag; /* "1" if triangle vertex; "2" if intersection points */
   struct vertex *link; /* link between two intersections */
                        /* NULL if triangle vertex */
   struct vertex *next; /* for linked list */
}
```

Before the intersection points are evaluated, the three triangle vertices are made into a circular singly linked list `Vlist` according to their "`t`" values, as follows. Their "`Vflag`"s are set to "1" and their "`link`"s to "`NULL`". For each plane which crosses the triangle, we compute the intersections—in general, two different



(a) Normal subdivision **(b)** Singular subdivision

Figure 1: Subdivision of a triangle: (a) There are three pairs of intersections are generated. They are marked with grey circles and linked by a dark line segment each other. The circular linked list *Vlist* is $\mathbf{P}_0 \to \mathbf{Q}_0 \to \mathbf{Q}_1 \to \mathbf{P}_1 \to \mathbf{Q}_1' \to \mathbf{Q}_0' \to \mathbf{Q}_2' \to \mathbf{P}_2 \to \mathbf{Q}_2 \to \mathbf{P}_0$. According to our loop generation algorithm, there are four loops generated. They are $\mathbf{P}_0\mathbf{Q}_0\mathbf{Q}_0'\mathbf{Q}_2'\mathbf{Q}_2$, $\mathbf{Q}_0\mathbf{Q}_1\mathbf{Q}_1'\mathbf{Q}_0'$, $\mathbf{Q}_1\mathbf{P}_1\mathbf{Q}_1'$ and $\mathbf{Q}_2'\mathbf{P}_2\mathbf{Q}_2$. In (b) is a singular subdivision case, where the original triangle vertex $\mathbf{P}_0$ is removed. In such a case, the linked list is $\mathbf{Q}_0 \to \mathbf{P}_1 \to \mathbf{Q}_0' \to \mathbf{P}_2 \to \mathbf{Q}_0$. The two loops generated are $\mathbf{P}_1\mathbf{Q}_0'\mathbf{Q}_0$ and $\mathbf{Q}_0'\mathbf{P}_2\mathbf{Q}_0$.

ones. The two intersections are thus linked each other by entry "`link`" and their "Vflag"s are set as "2". According to the parameterization of triangle edge, the "`t`" value of such two points should be in $[0, 3]$. We then insert the two intersections into `Vlist`

If the two intersections are identical, that is, their "`t`" values are equal, the plane is not regarded as crossing the triangle and no new intersections are inserted into `Vlist`. Sometimes the plane may cross one triangle vertex and one edge, as shown in Figure 1(b). When this occurs, first the triangle vertex crossed is deleted from `Vlist`, then two intersections are inserted in `Vlist`. For a plane crossing a triangle edge, we do not regard the plane as intersecting the triangle, so no intersection are generated.

After all possible intersections are obtained, we generate a circular linked list, in which two intersections that belong to the same cutting plane are linked to each other. This information is important for us to generate loops in the following step.

3.2 Generating loops from linked list

Here we show how to generate the loop from the circular linked vlist `Vlist`(See Figure 1). A loop generation start from a `startV`. The `startV` is selected from the vertices of `Vlist` whose "`Vflag`" is 1. If the vertex is visited, its "Vflag" number minus 1. After all loops are found, all "`Vflag`"s of vertices in `Vlist` become "0", since each intersection point is used twice and triangle vertice are used once. Traversal of the `Vlist` to generate the loop follows a rule: the first loop edge is the triangle edge which is determined by "`next`", then an intersection edge which is determined by "`link`". This loop generation algorithm has the property that every loop generated has the same vertex order as the original triangle, *i.e.*, if the original triangle's vertices are clockwise order, so are the loop's. This is important, for the result triangular mesh to maintain its normal. The descriptive algorithm is listed as follows:

```
Generate_all_Loops(Vlist)
{
    While( (startV=Get_StartV(Vlist)) != NULL )
         Find_A_Loop(startV, loop);
}

Get_StartV(Vlist)
{
    if(Vlist->Vflag==1) return Vlist;
    ptrV=Vlist->next;
    while(ptrV!=Vlist) {
        if(ptrV->Vflag==1)  return ptrV;
        else ptrV=ptrV->next;
    }
    return NULL;
```

```
}

Find_A_Loop(startV, loop)
{
     add_vertex_to_loop(startV, loop);
     startV->Vflag--;
     ptrV=startV->next;
     status=TRIANGLE_EDGE;

     while(ptrV!=startV)
     {
          add_vertex_to_loop(ptrV, loop);
          ptrV->Vflag--;
          if(status==TRIANGLE_EDGE) {
              if(ptrV->link==NULL)
                   ptrV=ptrV->next;
              else {
                  ptrV=ptrV->link;
                  status=INTERSECTION_EDGE;
              }
          }
          else if(status==INTERSECTION_EDGE) {
              ptrV=ptrV->next;
              status=TRIANGLE_EDGE;
          }
     }
}
```

3.3 Re-triangulation

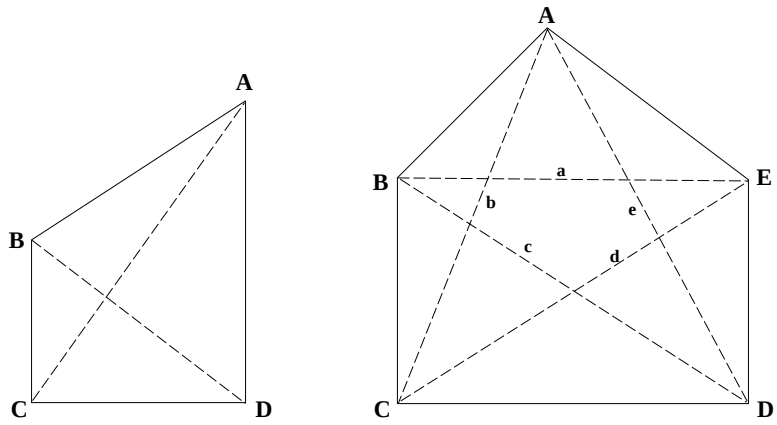


Figure 2: Triangulation of a loop: (Left) choose the diagonal with shorter length; here **BD**, so that triangles **ABD** and **BCD** are generated; (Right) If minimal sum of diagonals is $b+e$, the diagonals **AC** and **AD** are selected for triangulation, and triangles **ABC**, **ACD**, **ADE** are generated.

According to the above description, there are in all three kinds of loop generated, whose edge numbers are 3, 4 or 5. Any loop with 4 or 5 edges will be triangulated (Figure. 2). Here the triangulation criterion adopted is the minimization of the sum of triangle edge lengths. By this criterion, a 4-edge loop will be triangulated by using the shorter diagonal. For a 5-edge loop, each vertex emits two diagonals. We choose the vertex for which the sum of the lengths of these two is least.

4 Functional composition of triangle and Bézier volume

After subdivision and re-triangulation, each triangular mesh of object must lie within the parametric space of a Bézier volume. Let $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$ be a triangle, whose local coordinates corresponding to the Bézier volume

within which it lies are (u_0, v_0, w_0), (u_1, v_1, w_1), (u_2, v_2, w_2) respectively. This triangle can be parametrized by barycentric coordinates as follows:

$$u(x,y,z) = u_0x + u_1y + u_2z \tag{5}$$
$$v(x,y,z) = v_0x + v_1y + v_2z \tag{6}$$
$$w(x,y,z) = w_0x + w_1y + w_2z \tag{7}$$

where $x, y, z \geq 0$ and $x + y + z = 1$. After substituting above equations into (2) and functional composition, we can get a triangular Bézier surface patch, which is the accurate deformation result of the triangle $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$.

The functional composition for Bernstein polynomials has been studied by DeRose *et al.*[10, 11]. Unlike their methods, we explore this problem with shifting operators, with whose help the procedure of functional composition becomes clear and intuitive. In this section, we study composition of a Bézier volume and a linear triangle.

By substituting (5-7) into (2), we can get:

$$\begin{aligned}
\mathbf{R}(x,y,z) &= \mathbf{r}(u(x,y,z), v(x,y,z), w(x,y,z)) \\
&= [(1-u(x,y,z))I + u(x,y,z)E_u]^{n_u}[(1-v(x,y,z))I + v(x,y,z)E_v]^{n_v} \\
&\quad [(1-w(x,y,z))I + w(x,y,z)E_w]^{n_w}\mathbf{r}_{000} \\
&= [((1-u_0)I + u_0E_u)x + ((1-u_1)I + u_1E_u)y + ((1-u_2)I + u_2E_u)z]^{n_u} \\
&\quad [((1-v_0)I + v_0E_v)x + ((1-v_1)I + v_1E_v)y + ((1-v_2)I + v_2E_v)z]^{n_v} \\
&\quad [((1-w_0)I + w_0E_w)x + ((1-w_1)I + w_1E_w)y + ((1-w_2)I + w_2E_w)z]^{n_w}\mathbf{r}_{000}
\end{aligned}$$

Note $Au_i = (1-u_i)I + u_iE_u$, $Av_i = (1-v_i)I + v_iE_v$ and $Aw_i = (1-w_i)I + w_iE_w (i = 0, 1, 2)$. With help of these notations, above $\mathbf{R}(x, y, z)$ can be simplified as:

$$\begin{aligned}
\mathbf{R}(x,y,z) &= [Au_0x + Au_1y + Au_2z]^{n_u}[Av_0x + Av_1y + Av_2z]^{n_v}[Aw_0x + Aw_1y + Aw_2z]^{n_w}\mathbf{r}_{000} \\
&= \left[\sum_{i_u+j_u+k_u=n_u} Au_0^{i_u}Au_1^{j_u}Au_2^{k_u}B^{n_u}_{i_u,j_u,k_u}(x,y,z)\right]\left[\sum_{i_v+j_v+k_v=n_v} Av_0^{i_v}Av_1^{j_v}Av_2^{k_v}B^{n_v}_{i_v,j_v,k_v}(x,y,z)\right] \\
&\quad \left[\sum_{i_w+j_w+k_w=n_w} Aw_0^{i_w}Aw_1^{j_w}Aw_2^{k_w}B^{n_w}_{i_w,j_w,k_w}(x,y,z)\right]\mathbf{r}_{000} \\
&= \sum_{i+j+k=N} \mathbf{R}_{ijk}B^N_{ijk}(x,y,z)
\end{aligned}$$

Where $N = n_u + n_v + n_w$, and $\mathbf{R}_{ijk}$ is:

$$\frac{\displaystyle\sum_{\substack{i_u+i_v+i_w=i \\ j_u+j_v+j_w=j \\ k_u+k_v+k_w=k}} \left(C^{n_u}_{i_u,j_u,k_u}C^{n_v}_{i_v,j_v,k_v}C^{n_w}_{i_w,j_w,k_w}Au_0^{i_u}Au_1^{j_u}Au_2^{k_u}Av_0^{i_v}Av_1^{j_v}Av_2^{k_v}Aw_0^{i_w}Aw_1^{j_w}Aw_2^{k_w}\mathbf{r}_{000}\right)}{C^N_{i,j,k}} \tag{8}$$

The control points $\mathbf{R}_{ijk}$ of the Bézier triangular patch can be evaluated by the generalized de Casteljau algorithm. We denote the bracketed expression in (8) by $Auvw_{ijk}$. The algorithm to compute control point $\mathbf{R}_{ijk}$ is:

```
Compute_Rijk()
{
    Rijk=0;
    for(iu=0; iu<=nu; iu++)
    for(ju=0; ju<=(nu-iu); ju++)  {
        ku=nu-iu-ju;
        for(iv=0; iv<=nv; iv++)
        for(jv=0; jv<=(nv-iv); jv++)  {
            kv=nv-iv-jv;
            iw=i-iu-iv; jw=j-ju-jv; kw=k-ku-kv;
            if(iw>=0 && jw>=0 && kw>=0 )
```

```
                Rijk = Rijk+Compute_Auvwijk();
        }
    }
    Rijk=Rijk/C(N,i,j,k);
}

Compute_Auvwijk()
{
    for(j=0; j<=nv; j++)
    for(k=0; k<=nw; k++)
    for(l=1; l<=nu; l++)  {
        if(l<=iu) u=u0;
        else if(l<=(iu+ju)) u=u1;
        else u=u2;
        for(i=0; i<=(nu-l); i++)
            r[i][j][k]=(1-u)*r[i][j][k]+u*r[i+1][j][k];
    }

    for(k=0; k<=nw; k++)
    for(l=1; l<=nv; l++)  {
        if(l<=iv) v=v0;
        else if (l<=(iv+jv)) v=v1;
        else v=v2;
        for(j=0; j<=(nv-l); j++)
            r[0][j][k]=(1-v)*r[0][j][k]+v*r[0][j+1][k];
    }

    for(l=1; l<=nw; l++) {
        if(l<=iw) w=w0;
        else if(l<=(iw+jw))  w=w1;
        else  w=w2;
        for(k=0; k<=(nw-l); k++)
            r[0][0][k]=(1-w)*r[0][0][k]+w*r[0][0][k+1];
    }

    Auvwijk=r[0][0][0]*C(nu,iu,ju,ku)*C(nv,iv,jv,kv)*C(nw,iw,jw,kw);
    return(Auvwijk);
}
```

The outward normal of the triangular Bézier patch is determined by the order of $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$. For example, if $\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$ is counter-clockwise seen from outside the object, the normal of result patches also point outside. This is important for rendering.

5 Discussion and implementation

Our subdivision and re-triangulation are related to knot vectors which are independent of the size of the triangular mesh. There may be very small triangles produced, for which it is unnecessary to conduct the composition computation, since there is almost no sample problem. Thus during our implementation of proposed method, we check the size of each triangle, as measured by maximal edge length. When this is less than a given threshold, as in the original FFD, only three vertices are deformed. The resulting triangular mesh can be thought of as a triangular Bézier patch of degree 1. For a triangle parallel to a coordinate plane in the parametric surface, namely $u = u_0$, the composition can be simplified along the u direction so that the degree of patch is $k_v + k_w$, rather than $k_u + k_v + k_w$. Secondly, during the user's interaction, it is enough that the deformed object is displayed as wire frame. In implementation, this can achieved by deforming the edges of triangular meshes. The algorithm for composition between Bézier volume and line segments can be deduced similarly to the procedure in Section 4.

For modeling environments that do not support Bézier patches, the computed curved model can be tesselated back to polygons through deCastejau algorithm[12]. Though the result is no longer completely accurate, it still avoids tearing and other sampling problem. For the environments that do support Bézier

patches, this technique is still useful, both for objects whose rest shape is naturally polygonal, and for polygonal models that are acquired externally.

As a direct extension, the continuous FFD proposed by Bechmann[3] can be processed similarly. This is omitted here. The main drawback of the proposed algorithm is that it will take more time to deform an object compared to the direct method. It can be balanced by rendering wireframe of deformed object for interactive modeling. Another problem is that the proposed technique can not be used on curved model directly unless the deformation tool is a Bézier volume. In such a situation, it is difficult to get accurate result of curved object subdivision against the knot vectors.

We have implemented the proposed algorithm on an SGI Indy workstation. In each example from Figure. 3 to Figure. 9, (a) displays an object to be deformed, represented as triangular mesh. Part (b) gives the result when FFD acts on the vertex of object. The B-spline control lattice is shown by points and dashed lines. Part (c) shows the object after subdivision and re-triangulation. If the B-spline volume is identical with a Bézier volume, no subdivision is performed. The parametric space of B-spline volume is described by dashed lines. Part (d) is the accurate FFD result, where each mesh is a triangular Bézier surface patch.

6 Conclusion

In this paper, B-spline FFD is explored by means of functional composition via shifting operators. The object is first subdivided and triangulated so that each triangular mesh must lie within a Bézier volume. Since the deformation result is precise, there is no sample problem any more. The proposed method can be easily integrated into existing geometric modeling and computer animation system.

7 Acknowledgements

The authors would thank Professor Dominique Bechmann from University of Louis Pasteur in France, who gave us valuable suggestions during our implementation, Dr. Timothy Poston from CIEMed in Singapore, who carefully read the manuscripts and gave helpful suggestions. The authors also thank the Dr. Ronen Barzel and journal reviewers for their constructive comments.

References

[1] A. Barr, "Global and Local Deformation of Solid Primitives", *Computer Graphics*, Vol.18, No.3, pp. 21–30, 1984.

[2] D. Bechmann, "Space Deformation Survey", *Computers&Graphics*, Vol.18, No.4, pp. 571–586, 1994.

[3] D.Bechmann, Y.Bertrand *et al.*, "Continuous Free-Form Deformation", *COMPUGRAPHICS'96 and a special issue of Computer Networks and ISDN systems*, Elsevier Science, Paris, Dec. 1996.

[4] D. Bechmann,"Multidimensional Free-Form Deformation Dools", *Eurographics'98, State of the art report*, 1998.

[5] P. Borrel, D. Bechmann, "Deformation of N-dimensional Objects", *Symposium on Solid Modeling Foundations and CADCAM Applications*, ACM Press, Austin, Texas, June 1991. Published in *International Journal of Computational Geometry & Applications*, Vol.1, No.4, pp. 137–155, 1991.

[6] G. Chang, "Bernstein Polynomial via the Shifting Operators", *American Mathematical Monthly*, Vol.91, pp. 634–638, 1984.

[7] Y. Chang, A. Rockwood, "A Generalized de Casteljau Approach to 3D Free-Form Deformation", *Siggraph97*, pp. 187–196, 1997.

[8] S. Coquillart,"Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling", *Computer Graphics*, Vol.24, No.4, pp. 187–193, 1990.

[9] S. Coquillart, P. Jancene, "Animated Free-Form Deformations: An Interactive Animation Technique", *Computer Graphics (Siggraph91)*, Vol.25, No.4, pp. 23–26, 1991.

[10] T. DeRose, "Compositing Bézier simplex", *ACM Trans. on Graphics*, Vol.7, No.3, pp. 198–221, 1988.

[11] T. DeRose, R. Goldman *et al.*, "Functional Composition Algorithms via Blossoming", *ACM Trans. on Graphics*, Vol.12, No.3, pp. 113–135, 1993.

[12] G. Farin, *Curves and Surfaces for Computer-Aided Geometric Design*, 2nd Edition, Academic Press, New York, 1990.

[13] J. Feng, L. Ma *et al.*, "A New Free-Form Deformation through the Control of Parametric Surfaces", *Computers & Graphics*, Vol.20, No.4, pp. 531–539. 1996.

[14] J. Griessmair, W. Purgathofer, "Deformation of Solids with trivariate B-splines", *Eurographics'89*, pp. 137–148, 1989.

[15] W. Hsu, J. Hughes *et al.*, "Direct Manipulation on Free-Form Deformation", *ACM Computer Graphics(Siggraph92)*, Vol.26, No.2, pp. 177–184, 1992.

[16] P. Karla, A. Mangli *et al.*, "Simulation of Facial Muscle Actions Based on Rational Free-Form Deformation",*Computer Graphics Forum (Eurographics92)*, Vol.2, No.3, pp. 59–69, 1992.

[17] H. Lamousin, W. Waggenspack, "NURBS-based Free-form Deformation", *IEEE Computer Graphics and Applications*, Vol.14, No.9, pp. 59–65, 1994.

[18] F. Lazarus, S. Coquillart *et al.*, "Axial Deformation: An Intuitive Deformation Technique", *Computer-Aided Design*, Vol.26, No.8, pp. 607–613, 1994.

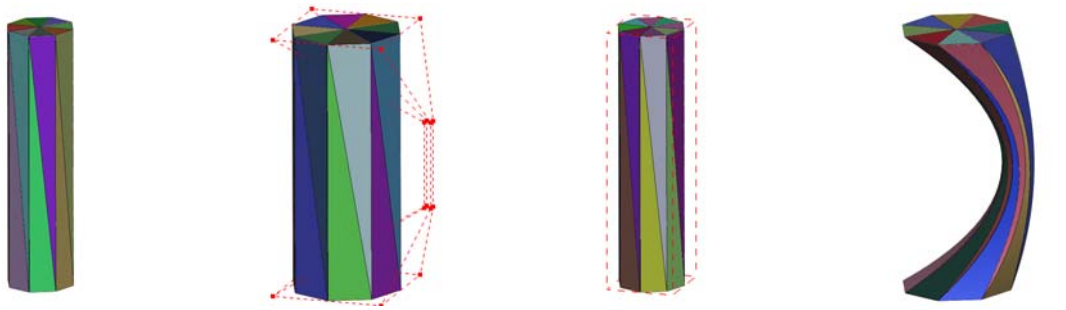
[19] R. MacCracken, K. Joy, "Free-Form Deformation with Lattice of Arbitrary Topology", *Proceedings of SIGGRAPH96*, pp. 181–183, 1996.

[20] L. Moccozet, N. Magnenat-Thalmann, "Dirichlet Free-Form Deformations and Their Application to Hand Simulation", *Computer Animation'97*, IEEE Computer Society, pp. 93–102, 1997.

[21] T. Sederberg, S. Parry, "Free-Form Deformation of Solid Geometric Models", *Computer Graphics*, Vol.20, No.4, pp. 537–541, 1986.

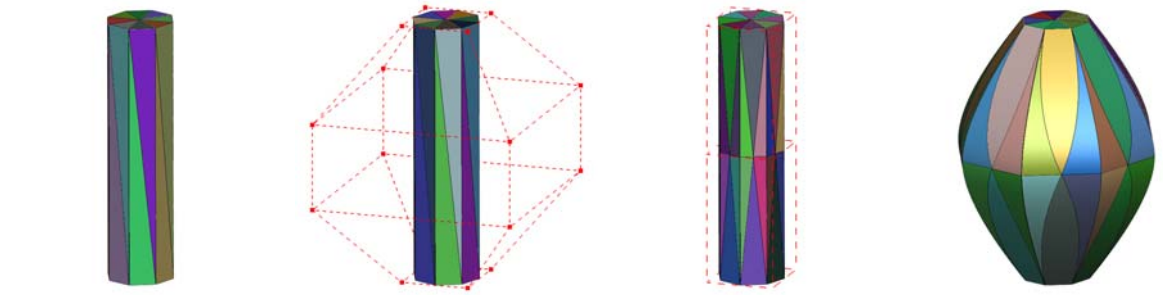
[22] K. Singh, E. Fiume, "Wires: A Geometric Deformation Technique", *Siggraph98*, pp. 404-413 , 1998.

Appendix: Implementation Results



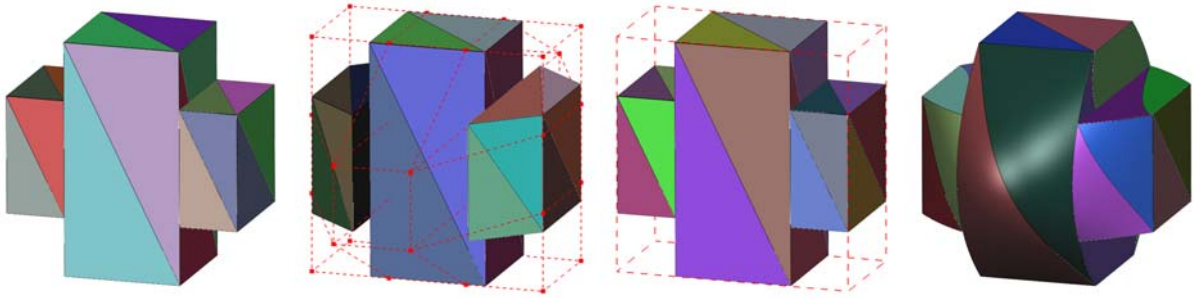
(a) Original object **(b)** Rough FFD **(c)** Object in parametric space **(d)** Accurate FFD

Figure 3: The degree of the B-spline volume is 1×3×1, with 2×4×2 control points.



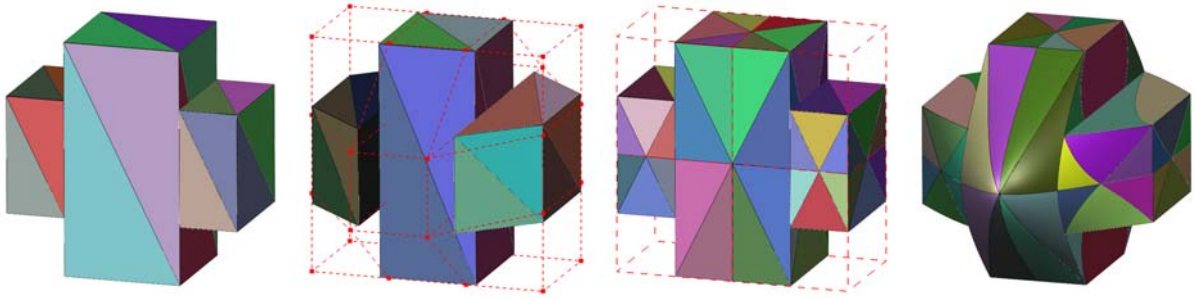
(a) Original object **(b)** Rough FFD **(c)** Subdivision and re-triangulation **(d)** Accurate FFD

Figure 4: The degree of the B-spline volume is 1×2×1, with 2×4×2 control points. Since there are not enough sample points on the object, the deformed version (b) created by the original FFD method remains unchanged.



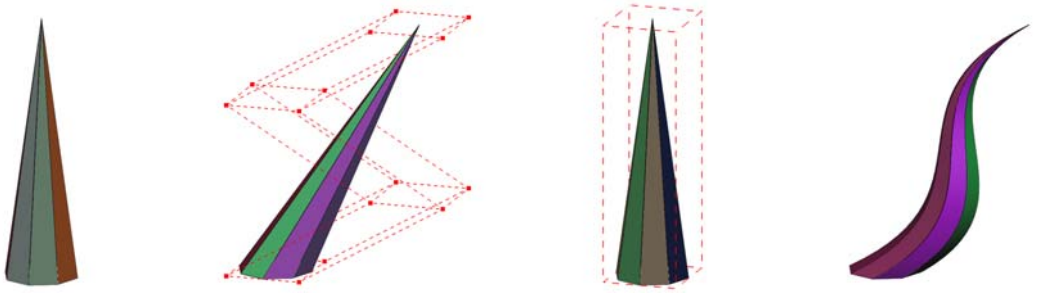
(a) Original object **(b)** Rough FFD **(c)** Object in parametric space **(d)** Accurate FFD

Figure 5: The degree of the B-spline volume is 3×1×3, with 4×2×4 control points.



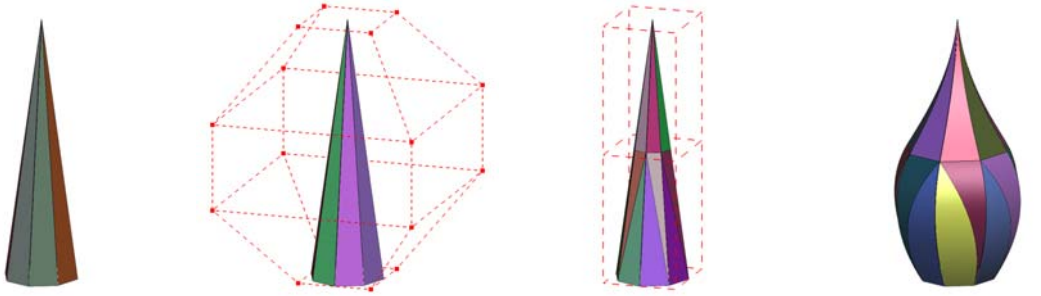
(a) Original object **(b)** Rough FFD **(c)** Subdivision and re-triangulation **(d)** Accurate FFD

Figure 6: The degree of the B-spline volume is 2×1×2, with 4×2×4 control points. The faces which are towards us in the three blocks begin as coplanar neighbours in (a). Because of the sample problem, they are disconnected in the rough FFD result (b).



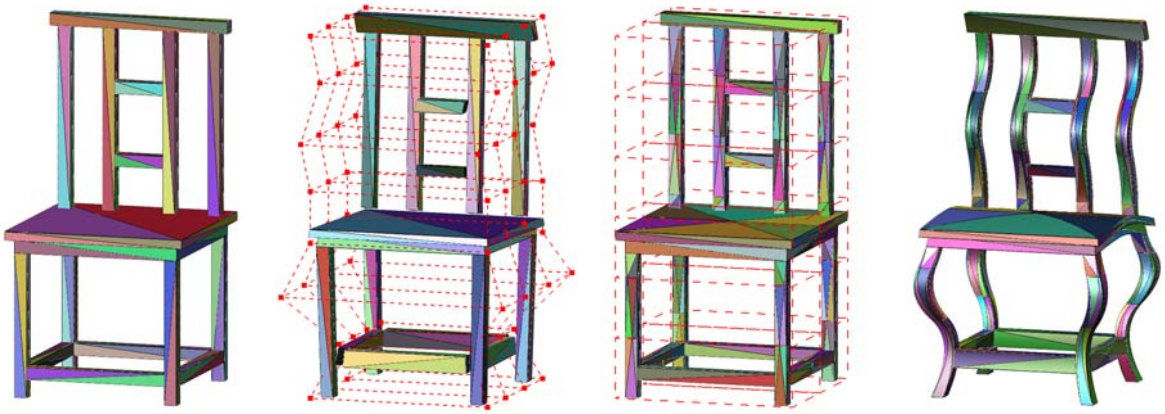
(a) Original object **(b)** Rough FFD **(c)** Object in parametric space **(d)** Accurate FFD

Figure 7: The degree of the B-spline volume is 1×3×1, with 2×4×2 control points.



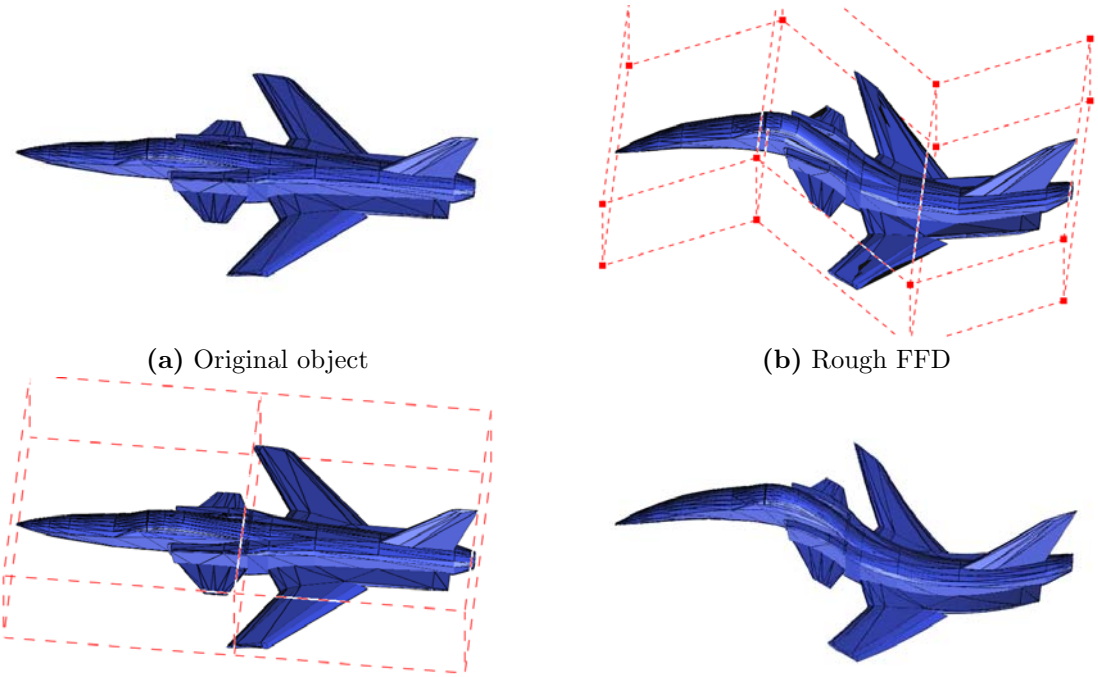
(a) Original object **(b)** Rough FFD **(c)** Subdivision and retriangulation **(d)** Accurate FFD

Figure 8: The degree of the B-spline volume is 1×2×1, with 2×4×2 control points.



(a) Original object **(b)** Rough FFD **(c)** Subdivision **(d)** Accurate FFD

Figure 9: The degree of the B-spline volume is 1×2×2, with 2×8×4 control points.



(a) Original object

(b) Rough FFD

(c) Subdivision and re-triangulation

(d) Accurate FFD

Figure 10: The degree of the B-spline volume is 2×1×1, with 4×2×2 control points.