Technical Report

UCAM-CL-TR-381
ISSN 1476-2986

Number 381

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Rendering for free form deformations

Uwe Michael Nimscheck

October 1995

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted
by the author for the degree of Doctor of Philosophy to the
University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-381*

Abstract

Sederberg's Free Form Deformation (FFD) is an intuitive modelling technique that lets users sculpt and deform objects without having to worry about internal model representation issues. Unfortunately, displaying these deformed objects is problematic and there exist no algorithms to display general FFD deformed polygonal models. Based on deRose's Bézier composition algorithms, we develop geometrically intuitive composition algorithms to find analytic expressions for deformed objects, which can then be rendered using standard rendering hardware. Alternatively, one can adaptively tessellate deformed objects into a mesh of triangles and display this deformed mesh. The Finite Element method provides us with a wealth of algorithms to mesh all types of objects. We show how to adapt these algorithms to computer graphics problems. The main problem is to define curvature measures to vary the mesh density according to the curvature of the deformed objects. We find such measures and use them to develop a new meshing scheme, based on Lo's advancing front algorithm, to mesh and render FFD deformed objects. Our algorithm is superior to existing schemes both in the quality of the generated meshes and in the variety of solids that it can be applied to.

The major contributions of this dissertation are:

- Development of geometrically intuitive algorithms to determine closed form expressions of FFD deformed surfaces.
- Transformation of tangent and normal vectors into deformed space.
- Development of a new advancing front meshing algorithm that allows to mesh solids that have been deformed by non-uniform B-spline volumes.
- Systematic experiments to assess the performance and limitations of the new meshing algorithm.

Acknowledgements

This report is, apart from some minor corrections, the dissertation which I submitted for my PhD degree. For technical reasons, the colour images in the Appendix have been replaced by greyscale images.

First of all, I should like to thank my supervisor, the late Neil Wiseman, for his help, advice and support during the course of my work. I am particularly grateful for his help and encouragement during my early days in Cambridge.

Also, I should like to thank Malcom Sabin for his advice during the course of my work and for acting as my supervisor after Neil Wiseman's untimely death. His support, especially during the final stages of my work, has been invaluable.

Particular thanks are also due to Peter Robinson for offering help when I needed it most. Further thanks go to Tom Sederberg at Brigham Young University for various helpful comments and to Rich Riesenfeld and Elaine Cohen at Utah, who sparked my interest in splines and geometric modelling.

The Computer Laboratory, and the Rainbow graphics group in particular, have provided a stimulating working environment. Thanks go to Adrian, Chris, Eileen, Jane, Jon, Kwai, Larry, Margaret, Martin, Neil, Oliver, Peter, Simon, Stanley, Stuart and Stefan. Chris and Jon deserve particular acknowledgement for proofreading chapters of this thesis.

I also gratefully acknowledge the financial support provided by Professor Roger Needham in form of a scholarship of the Computer Laboratory. Additionally, this work has been supported by a departmental award and the Isaac Newton Trust.

Finally, thanks go to my parents, whose support and encouragement made this all possible.

Contents

Chapter 1

Introduction

The representation of geometric objects in computers is of importance to areas as diverse as technical design, mechanical engineering, architecture, scientific visualisation and computer graphics. A trend has developed from modelling software that more or less resemble the designer's drawing board to systems that allow him to work with three-dimensional objects. A *geometric modelling system* or *modeller* is a computer-based system for creating, editing and accessing representation of geometric objects. Ways of characterising modelling systems include their internal representation of objects and their design method.

The Free Form Deformation (FFD) technique, introduced by Sederberg and Parry [SP86b], is both a representation and design paradigm that lets the user change the shape of objects in a free-form manner. This thesis deals with representation and display aspects of FFD.

1.1 Free Form Deformation

Modelling systems for Computer-Aided Design can be divided into two main categories: *surface modellers* and *solid modellers.* In surface modelling, 3D surfaces such as polygonal meshes, parametric patches and quadric patches are used to capture and intuitively manipulate the representation of *surfaces.* Solid modelling deals with complete, unambiguous representation of whole *solids.* The fields of solid modelling and surface modelling have been treated independently for a long time. Fundamental limitations of each system make it desirable to combine them. What is particularly sought is a system that lets the user intuitively manipulate the shape of complete solids in a free-form

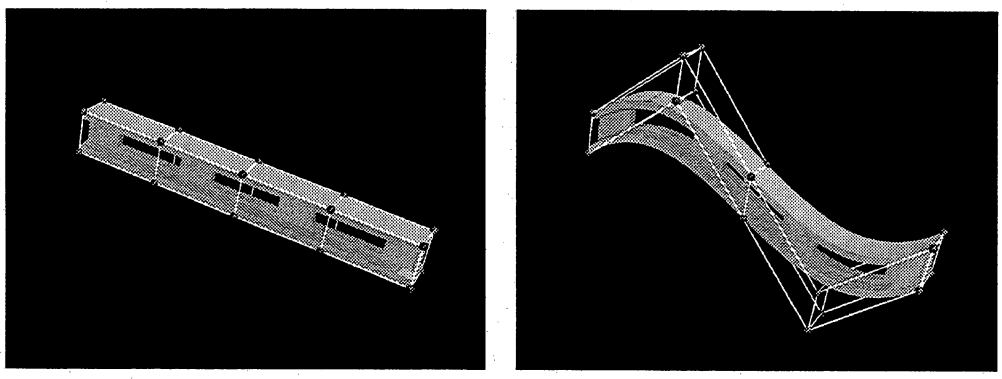
manner.

Combining the virtues of free form surfaces and solid modelling techniques is thus an area of current research. Sederberg [SP86b] categorises the approaches into three main classes:

- The combination of free-form surfaces with solid modellers by including parametric surface patches. One of the main problems of this approach is to ensure that the resulting representation meets the requirements of an abstract solid.
- Using hyperpatches (generated by parametric mapping of unit cubes into 3D space) as solid modelling primitives.
- Modelling directly with volumes bounded by implicit or algebraic surfaces.

A novel approach, known as the Free Form Deformation (FFD) technique, has been introduced by Sederberg and Parry [SP86b]. It is independent of the internal representation of the data in the solid modelling system (such as CSG or b-rep) and allows deformations to be applied either globally or locally.

Intuitively, FFD can be imagined as embedding a solid in a block of clear, pliable plastic. The deformation is applied indirectly by deforming this block of plastic, which in turn deforms the embedded solid. Figure 1.1 shows a deformation applied to a girder. The white lines indicate the edges of the deformation solid.



(a) (b)

Figure 1.1: Free Form Deformation of a girder

Mathematically, FFD is defined in terms of a trivariate tensor product Bernstein polynomial. A point (u, v, w) is assigned a new position (x, y, z) through a mapping

$$\mathbf{f}(u, v, w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{b}_{i,j,k} B_i^m(u) B_j^n(v) B_k^p(w),$$

where $\mathbf{b}_{i,j,k}$ are a set of control points that determine the deformation. Suppose $\mathbf{g}$ is a surface of an object. The deformed surface is then given by the functional composition $\mathbf{f} \circ \mathbf{g}$. FFD thus yields an indirect deformation of objects by deforming the surrounding space that the objects are embedded in.

1.2 Rendering and Meshing Deformed Objects

Currently, there is a lack of robust and efficient algorithms to render FFD deformed objects.

The rendering process of curved surfaces is well investigated. However, these existing surface rendering algorithms are not suited for FFD models, because they usually require the curved surface be defined in parametric form. In FFD, we only have the description of the undeformed surfaces $\mathbf{g}$ and a deformation function $\mathbf{f} : \mathcal{R}^3 \rightarrow \mathcal{R}^3$.

There are two ways of solving this problem:

- determining parametric expressions of the deformed objects that enable us to use existing rendering algorithms
- developing a rendering algorithm that does not require such parametric surface expressions.

In this thesis, we will consider both solutions. The first contribution of this thesis is to develop geometrically intuitive algorithms, based on the blossoming principle and deRose's functional composition paper [DeR88], to obtain a parametric description $\tilde{\mathbf{g}} = \mathbf{f} \circ \mathbf{g}$ of FFD deformed surfaces. Due to the limited applicability in practice, our emphasis is on geometric intuitivity rather than computational efficiency. Aided by the blossoming notation, our algorithms illustrate the geometric relation between the undeformed and deformed surfaces.

We show that the parametric expression $\tilde{\mathbf{g}}$ of the deformed surfaces can only be obtained for a certain class of deformation solids. Moreover, one of the effects of FFD is a dramatic increase in the degree of the deformed surfaces, which makes explicit evaluation prohibitively expensive. These two problems motivate our development of a rendering algorithm that does not require parametric surface expressions.

Modern graphics workstations are capable of rendering very large numbers of shaded triangle primitives, so an alternative algorithm would consist of subdividing the surfaces of the deformed object into meshes of triangles and rendering these meshes. The problem with subdividing the surfaces at a constant density is that aliasing problems will occur due to undersampling in regions of high curvature, whereas in regions of low curvature oversampling will result in an unnecessarily high number of triangles. The obvious solution to this problem is to subdivide adaptively according to curvature.

Existing adaptive meshing algorithms for FFD that do not require a parametric description of the deformed surfaces are due to Parry [Par86] and Griessmair and Purgathofer [GP89]. Both techniques have fundamental limitations: Parry's algorithm is restricted to block, sphere and cylinder primitives. Griessmair's algorithm only meshes convex polygons and can produce badly shaped triangles.

The process of meshing the surfaces of solids has received considerable attention in finite element methods. We therefore survey a number of existing finite element meshing algorithms with respect to their suitability to FFD and identify Lo's advancing front algorithm [Lo85] as a promising basis for an FFD-specific meshing scheme.

The second contribution of this dissertation is therefore the design of an FFD meshing algorithm with the following features:

- It produces triangle meshes with density varying according to the local curvature of surfaces. The triangle size is determined by a curvature measure based on the angle between surface normals. We show applications of our algorithm to a number of FFD meshing problems, including solids containing concave polygons and arbitrarily shaped holes.

- It generates surface triangle meshes for solids bounded by non self-intersecting polygons of any shape, convex or concave.

- It allows objects to be deformed by non-uniform B-spline volumes.

- Mesh compatibility is ensured across shared surface edges.
- Triangles are well shaped and produce good results when rendered.
- It copes with polygons containing holes.

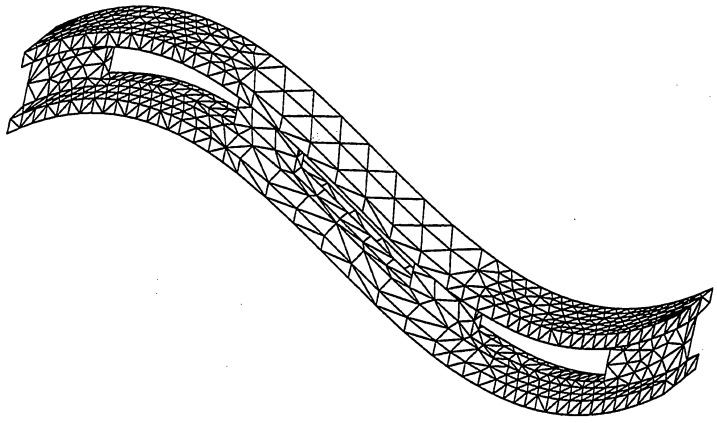


Figure 1.2: Deformed girder, 2318 triangles

Figure 1.2 shows our meshing algorithm applied to the deformed girder shown in Figure 1.1.

1.3 Thesis Overview

The remainder of this thesis is organised as follows:

- In Chapter 2, we provide a consistent framework for Bézier curves, surfaces and volumes using the blossoming principle. The blossoming notation, based on polar forms and introduced into computer graphics by Ramshaw [Ram87], is a relatively recent, geometrically intuitive way of defining Bézier curves and B-splines. The discussion of Bézier volumes is of significance to FFD, because FFD deformation functions are usually expressed as Bézier volumes. This chapter does not contain any new results. However, the extension of blossoming to tensor

product surfaces and volumes has not yet received much attention. We therefore develop a consistent system of notation that will be drawn upon in the following chapters.

- In Chapter 3, we investigate the composition process for curves and surfaces with tensor product Bézier volume deformations. Using the blossoming principle, we derive new, geometrically intuitive, algorithms to find closed form expressions for deformed curves and surfaces.

- Chapter 4 contains an in-depth discussion of the FFD technique and shows how to apply the previously derived composition algorithms to FFD. As a new result, we show how to determine normal vectors of deformed objects analytically.

- In Chapter 5 we discuss the rendering process of objects that have undergone FFD. We discuss existing algorithms and survey finite element based meshing schemes that can be used to adaptively generate meshes of FFD models. The advancing front algorithm is identified as a method suitable for meshing FFD deformed solids.

- In Chapter 6, we develop a new, advancing front based, meshing scheme for FFD deformed solids. Meshing algorithms can be divided into two parts: how to subdivide and where to subdivide. The first part of the chapter deals with the adaptation of the advancing front algorithm to FFD problems. In the second part we develop curvature measures to determine the triangle mesh density.

- Chapter 7 contains applications of our newly developed meshing algorithm. We show meshes and shaded images of a number of solids that have undergone FFD. Shaded colour images of the deformed objects can be found in Appendix A.

- In Chapter 8, we summarise the main results of this thesis and conclude with suggestions for further research.

Chapter 2

Blossoms and Bézier curves

Free Form Deformations are usually defined in terms of Bézier or B-spline volumes. Bézier volumes are three dimensional generalisations of Bézier curves. There are two basic ways of extending the concept of Bézier curves to higher dimensions, these are *Bézier simplexes* and *tensor product hypersurfaces*. This chapter gives a brief introduction to Bézier curves and then extends these ideas to surfaces and volumes.

Bézier curves and their generalisations are well understood concepts, see Farin [Far93]. Our introduction makes use of the blossoming principle, which has been recently introduced into computer graphics to provide a simple and intuitive way of developing Bézier curve theory.

This chapter introduces Bézier curves and provides a consistent framework for our subsequent investigation of FFD. The discussion is self contained, but it is not intended to serve as a general introduction to the theory of Bézier curves. Readers with no knowledge of Bézier curves will probably find it helpful to read an introductory text such as Boehm [BFK84] or Farin [Far93] first. Readers familiar with the concept of blossoming will still find it useful to read this chapter to familiarise themselves with the notation we develop to label curves and simplices. Particular attention is drawn to the section on composing Bézier curves (Section 2.2.5) and on our labelling scheme for surfaces and volumes (Section 2.4).

2.1 Polar Forms and Blossoming

Polar forms were developed as a mathematical tool for the analysis of polynomials (van der Waerden [vdW70]). They have recently been introduced into the field of computer graphics through the work of de Casteljau [dFdC86] and Ramshaw [Ram88]. A good introduction can be found in Seidel [Sei93]. The main benefit of the polar form approach is its simplicity and clarity, which make it easier to understand the standard theorems and algorithms of Bézier and B-spline theory.

The idea of polar forms is to replace a function of high degree in one variable by an equivalent function of degree one but with many variables. Ramshaw was initially not aware of the name polar form and coined the term *blossoming*. We will use blossoming, as this is the name most widely used in the computer graphics and computer aided design areas.

Let $\mathbf{f}(u)$ be a polynomial map $\mathcal{R} \rightarrow \mathcal{R}^d$ of degree n. Then there is a *unique* multiaffine polar form $\Delta(u_1, \ldots, u_n) : \mathcal{R}^n \rightarrow \mathcal{R}^d$, called the *blossom* of $\mathbf{f}$, that has the following properties:

- It produces $\mathbf{f}$ on the *diagonal*:

$$\mathbf{f}(u) = \Delta(u, \ldots, u). \tag{2.1}$$

- It is *symmetric* for any permutation of variables:

$$\Delta(u_1, \ldots, u_n) = \Delta(u_{i_1}, \ldots, u_{i_n}), \tag{2.2}$$

  we will make frequent use of this property in the algebraic manipulations that follow.

- It is *multiaffine*. Generally, a function $F(u)$ is called *affine* if it satisfies $F(\sum_i \alpha_i u_i) = \sum_i \alpha_i F(u_i)$ for all $\alpha \in \mathcal{R}$ with $\sum_i \alpha_i = 1$. It is multiaffine if it is affine in each argument (while the others are held constant), that is

$$\Delta(u_1, \ldots, \sum_i \alpha_i u_{j_i}, \ldots, u_n) = \sum_i \alpha_i \Delta(u_1, \ldots, u_{j_i}, \ldots, u_n) \tag{2.3}$$

  for all $j = 1, \ldots, n$.

Every polynomial has a unique polar equivalent. Let $\mathbf{f}(u)$ be a degree n polynomial expressed in monomial form

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{a}_i u^i.$$

The corresponding blossom of $\mathbf{f}(u)$ is then given in Seidel [Sei93]:

$$\Delta(u_1, \ldots, u_n) = \sum_{i=0}^{n} \mathbf{a}_i \binom{n}{i}^{-1} \sum_{\substack{S \subseteq \{1,\ldots,n\} \\ |S| = i}} \prod_{j \in S} u_j \tag{2.4}$$

with

$$\binom{n}{i} \equiv \frac{n!}{i!(n-i)!}.$$

Example 1 *For a cubic polynomial*

$$\mathbf{f}(u) = \mathbf{a}_0 + \mathbf{a}_1 u + \mathbf{a}_2 u^2 + \mathbf{a}_3 u^3$$

the corresponding blossom is given by

$$\Delta(u_1, u_2, u_3) = \mathbf{a}_0 + \frac{\mathbf{a}_1}{3}(u_1 + u_2 + u_3) + \frac{\mathbf{a}_2}{3}(u_1 u_2 + u_2 u_3 + u_3 u_1) + \mathbf{a}_3(u_1 u_2 u_3).$$

Since blossoms often have a number of equal arguments, it is convenient to have a shorthand notation. Let

$$\Delta(r^{<i>}, s^{<j>}) \equiv \Delta(\underbrace{r, \ldots, r}_{i}, \underbrace{s, \ldots, s}_{j}) \tag{2.5}$$

where the superscript $< i >$ denotes a multiplicity i of its argument. Note that the symmetry property ensures that the ordering of the arguments does not matter.

2.1.1 Derivatives

The general formula for the kth derivative of a degree n polynomial $f(u)$ in blossom notation is

$$f^{(k)}(u) = \frac{n!}{(n-k)!} \sum_{j=0}^{k} \binom{k}{j} (-1)^{k-j} \Delta((u+1)^{<j>}, u^{<n-j>}). \tag{2.6}$$

The derivation of this formula is quite involved as it requires extending the domain of blossoms from affine space to linear space. A detailed discussion is provided by Ramshaw [Ram87].

We can see that computing the kth derivative of a polynomial at parameter value u is done by varying k arguments away from this value. If we keep all arguments fixed, all the information we can obtain is the value at u. Varying all arguments allows us to compute all derivatives at a point and we therefore know the polynomial completely.

2.2 Bézier Curves

Having introduced the blossoming notation, we now show how this concept helps us understand the theory underlying Bézier curves.

The *de Casteljau* algorithm [dFdC59] provides an intuitive way to construct Bézier curves by repeated *linear interpolation*[1]. We now illustrate the de Casteljau algorithm using the blossoming notation. This provides us with insight into properties of Bézier curves which will be useful in further developments.

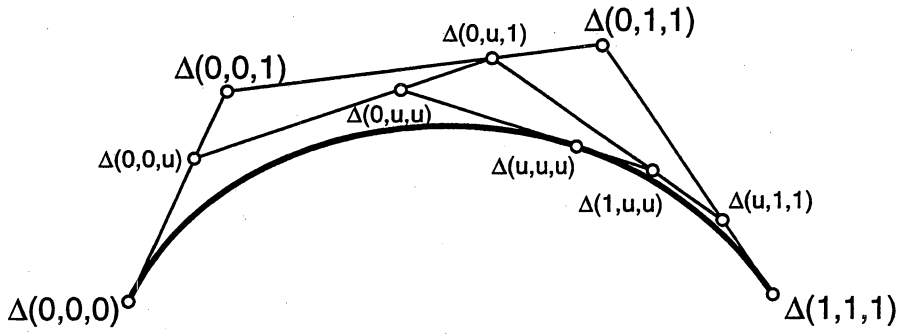


Figure 2.1: The De Casteljau algorithm for a cubic Bézier curve

Figure 2.1 shows a cubic (degree 3) Bézier curve $\mathbf{f}(u) = \Delta(u,u,u) : \mathcal{R} \rightarrow \mathcal{R}^d$, defined over the interval $[0,1]$. The Bézier points are labelled as blossom values $\Delta(0,0,0)$, $\Delta(0,0,1)$, $\Delta(0,1,1)$ and $\Delta(1,1,1)$. A curve point $\Delta(u,u,u)$ is now constructed as follows: By linearly interpolating between the Bézier

[1] As pointed out by Farin [Far93] (Chapter 2, page 20), the correct term is *affine interpolation*, however, we shall use *linear interpolation* to be consistent with most of the existing literature.

points we obtain the intermediate points $\Delta(0,0,u)$, $\Delta(0,u,1)$ and $\Delta(u,1,1)$. Repeated interpolation, permuting the arguments where necessary, yields the blossom values $\Delta(0,u,u)$ and $\Delta(1,u,u)$, from which we finally obtain the curve point $\Delta(u,u,u)$.

The blossom notation conveys a significant amount of geometrical information: Three points lie on the same line if their corresponding blossom arguments agree in all but one argument and their distance ratio is then given by the varying argument. For example, the points

$$\Delta(0,\boxed{0},1), \Delta(0,\boxed{t},1), \Delta(0,\boxed{1},1)$$

agree in all but one argument (boxed) and lie therefore on the same line. $\Delta(0,t,1)$ divides the line in the ratio $t:(1-t)$. This means that given two points whose blossoms agree in all but one argument, we can construct any other point on the same line by linear interpolation between them.

The de Casteljau algorithm provides a geometrically intuitive way of constructing Bézier curves by recursive linear interpolation between control points. We can use the blossom properties discussed in Section 2.1 to show that the intermediate blossom values actually correspond to the intermediate points of the de Casteljau algorithm.

Let us first express the parameter u as an affine combination of the interval endpoints 0 and 1

$$u = (1-u)\cdot 0 + u \cdot 1.$$

If we express the $n+1$ control points of a degree n Bézier curve defined over the interval $[0,1]$ in blossom notation as $\Delta(0^{<n-i>}, 1^{<i>})$, we can use the multiaffine invariance property of blossoms (Equation (2.3)) to recursively generate intermediate points

$$\begin{aligned}&\Delta(0^{<n-i-j>}, u^{<j>}, 1^{<i>}) = \\ &\quad (1-u)\,\Delta(0^{<n-i-j+1>}, u^{<j-1>}, 1^{<i>}) + u\,\Delta(0^{<n-i-j>}, u^{<j-1>}, 1^{<i+1>})\end{aligned}$$

and

$$\mathbf{f}(u) = \Delta(u^{<n>}).$$

Figure 2.2 shows a schematic representation of the recursive generation of blossoms for a cubic Bézier curve.

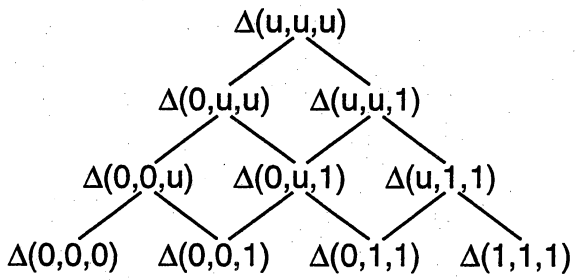


Figure 2.2: Recursive generation of intermediate points for the cubic case

By repeatedly applying Equation (2.3) we can obtain a non-recursive expression for Bézier curve points $\mathbf{f}(u)$:

$$\begin{aligned}
\mathbf{f}(u) &= \Delta(u,\ldots,u) \\
&= \Delta(u^{<n-1>}, (1-u)\cdot 0 + u\cdot 1) \\
&= (1-u)\,\Delta(u^{<n-1>}, 0) + u\,\Delta(u^{<n-1>}, 1) \\
&= (1-u)^2\Delta(u^{<n-2>}, 0^{<2>}) + 2(1-u)\,u\,\Delta(u^{<n-2>}, 0, 1) \\
&\quad +u^2\Delta(u^{<n-2>}, 1^{<2>}) \\
&= \sum_{i=0}^{n} B_i^n(u)\,\Delta(0^{<n-i>}, 1^{<i>}) \qquad (2.7)
\end{aligned}$$

where B_i^n are the *Bernstein polynomials* with

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{(n-i)}.$$

If we compare this with the Bernstein-Bézier form given in standard graphics texts such as Farin [Far93]

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{b}_i B_i^n(u),$$

we can see that the Bézier points $\mathbf{b}_i$ are the blossom values

$$\mathbf{b}_i = \Delta(0^{<n-i>}, 1^{<i>}).$$

2.2.1 Extending the Domain

The above derivation of $\mathbf{f}(u)$ in blossom form is not restricted to the domain interval $[0, 1]$, rather, $\mathbf{f}(u)$ can be defined over any interval $[r, s]$ by expressing u as an affine combination of r and s:

$$u = \frac{s-u}{s-r} r + \frac{u-r}{s-r} s \tag{2.8}$$

Equation (2.7) then becomes

$$\mathbf{f}(u) = \sum_{i=0}^{n} B_i^n(u) \Delta(r^{<n-i>}, s^{<i>}) \tag{2.9}$$

with

$$B_i^n(u) = \binom{n}{i} \left(\frac{u-r}{s-r}\right)^i \left(\frac{s-u}{s-r}\right)^{n-i}.$$

The Bézier points $\mathbf{b}_i$ can now be found as blossom values

$$\mathbf{b}_i = \Delta(r^{<n-i>}, s^{<i>}). \tag{2.10}$$

Thus, for a degree 3 curve defined over the interval $[r, s]$, the Bézier points $\mathbf{b}_0$, $\mathbf{b}_1$, $\mathbf{b}_2$ and $\mathbf{b}_3$ can be expressed as the blossom values $\Delta(r, r, r)$, $\Delta(r, r, s)$, $\Delta(r, s, s)$ and $\Delta(s, s, s)$.

The blossoming notation enables us to conveniently introduce standard operations on Bézier curves such as computing derivatives, subdivision, degree elevation and general composition.

2.2.2 Derivatives

We can use Equation (2.6) to determine the derivative of a Bézier curve. Recall that

$$f^{(k)}(u) = \frac{n!}{(n-k)!} \sum_{j=0}^{k} \binom{k}{j} (-1)^{k-j} \Delta(u+1^{<j>}, u^{<n-j>}).$$

We can employ the multivariate invariance property and obtain[2]

$$\begin{aligned}
\frac{df(u)}{du} &= n\left(\Delta(u^{<n-1>}, u+1) - \Delta(u^{<n-1>}, u)\right) \\
&= n\left(\Delta(u^{<n-1>}, 1)\right) \\
&= n\left(\Delta(u^{<n-1>}, \frac{s-r}{s-r})\right) \\
&= \frac{n}{s-r}\left(\Delta(u^{<n-1>}, s) - \Delta(u^{<n-1>}, r)\right) \\
&= \frac{n}{s-r}\left(\sum_{i=0}^{n-1} \Delta(r^{<n-i-1>}, s^{<i+1>})\, B_i^{n-1}(u)\right. \\
&\quad \left. - \sum_{i=0}^{n-1} \Delta(r^{<n-i>}, s^{<i>})\, B_i^{n-1}(u)\right) \\
&= \frac{n}{s-r}\sum_{i=0}^{n-1}\left((\Delta(r^{<n-i-1>}, s^{<i+1>}) - \Delta(r^{<n-i>}, s^{<i>})\right) B_i^{n-1}(u) \\
&= \frac{n}{s-r}\sum_{i=0}^{n-1}(\mathbf{b}_{i+1} - \mathbf{b}_i)\, B_i^{n-1}(u),
\end{aligned}$$

so the derivative of a degree n Bézier curve is a Bézier curve[3] of degree $n-1$.

2.2.3 Subdivision

A Bézier curve is generally defined over some interval $[r, s]$, but any piece of it is also defined over some subinterval $[t, u]$. The process of finding the Bézier points associated with this subinterval is called *subdivision*.

Figure 2.3 shows a quadratic Bézier curve defined over the parameter interval $[r, s]$. Suppose we want to subdivide the curve and find the Bézier points for the subinterval $[t, u]$. According to Equation (2.9), the Bézier points of a curve associated with an arbitrary interval $[r, s]$ are the blossom values

$$\Delta(r^{<n-i>}, s^{<i>}).$$

Thus the Bézier points for the curve over the interval $[t, u]$ are the blossom values

$$\Delta(t^{<n-i>}, u^{<i>}).$$

[2]This is a linear combination in linear space rather than in affine space, which is why the coefficients sum up to 0 rather than 1.

[3]Note that the derivative curve is obtained by repeated linear interpolation of differences of points, therefore its range is in linear space rather than affine space.

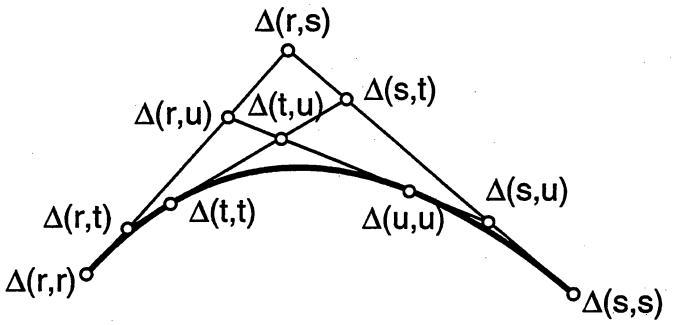


Figure 2.3: Subdivision of a quadratic Bézier curve

In the case of the quadratic curve in Figure 2.3, the Bézier points associated with interval $[r, s]$ are $\Delta(r, r)$, $\Delta(r, s)$ and $\Delta(s, s)$, whereas the Bézier points for interval $[t, u]$ are $\Delta(t, t)$, $\Delta(t, u)$ and $\Delta(u, u)$.

2.2.4 Degree Elevation

In some situations it is desirable to increase the degree of a Bézier curve to add flexibility. This is usually done by adding another Bézier point such that the shape of the curve remains unchanged.

Let $\Delta^n(u_1, \ldots, u_n)$ be the blossom of a degree n Bézier curve. Then the degree $n+1$ blossom $\Delta^{n+1}(u_1, \ldots, u_{n+1})$ that describes the same curve is given by

$$\Delta^{n+1}(u_1, \ldots, u_{n+1}) = \frac{1}{n+1} \sum_{i=1}^{n+1} \Delta^n(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_{n+1}),$$

which can be verified by noting that the right-hand side is multiaffine, symmetric and produces $\mathbf{f}(u)$ on the diagonal (Bartels [Bar91]).

Figure 2.4 shows the old and new control points of a quadratic Bézier curve. The new Bézier points are given by

$$\begin{aligned}
&\Delta^{n+1}(r^{<n-j+1>}, s^{<j>}) \\
&= \frac{1}{n+1} \left(\sum_{i=1}^{j} \Delta^n(r^{<n-j+1>}, s^{<j-1>}) + \sum_{i=j+1}^{n+1} \Delta^n(r^{<n-j>}, s^{<j>}) \right) \\
&= \frac{1}{n+1} \left(j\, \Delta^n(r^{<n-j+1>}, s^{<j-1>}) + (n+1-j)\, \Delta^n(r^{<n-j>}, s^{<j>}) \right).
\end{aligned}$$

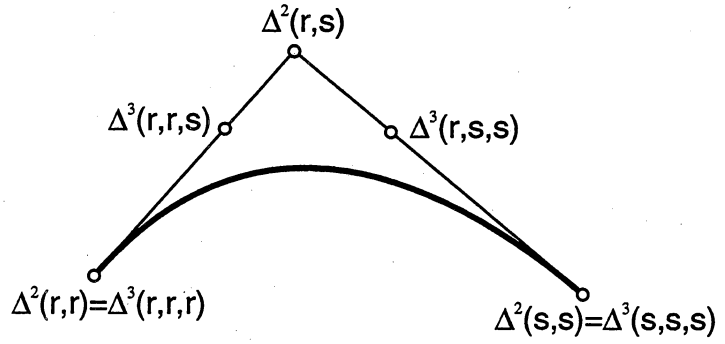


Figure 2.4: Degree elevation of a quadratic Bézier curve

Example 2 *For a quadratic Bézier curve as in Figure 2.4, the new Bézier points can be found as blossom values*

$$\Delta^3(r^{<3-j>}, s^{<j>}) = \frac{1}{3}\left(j\,\Delta^2(r^{<3-j>}, s^{<j-1>}) + (3-j)\,\Delta^2(r^{<2-j>}, s^{<j>})\right).$$

2.2.5 General Composition

Evaluation and subdivision can be seen as special cases of the general functional composition problem. The composition problem consists of finding the composed curve $\tilde{\mathbf{f}} = \mathbf{f} \circ g$ of a Bézier curve $\mathbf{f}$ and a polynomial function g.

The problem can be stated as follows:
Given a Bézier curve

$$\mathbf{f}(t) : \mathcal{R} \to \mathcal{R}^d$$

and a polynomial function

$$g(u) : \mathcal{R} \to \mathcal{R},$$

find the corresponding reparametrised Bézier curve[4]

$$\tilde{\mathbf{f}}(u) = \mathbf{f}(g(u)) : \mathcal{R} \to \mathcal{R}^d.$$

Let us express the degree m Bézier curve, defined over the interval $[0, 1]$ as

$$\begin{aligned} \mathbf{f}(t) &= \sum_{i=0}^{m} \mathbf{b}_i B_i^m(t) \\ &= \sum_{i=0}^{m} \Delta(0^{<m-i>}, 1^{<i>}) B_i^m(t) \end{aligned}$$

[4]Note that if the range of $g(u)$ is only a part of the domain of $\mathbf{f}(t)$, then the reparametrised curve $\tilde{\mathbf{f}}(u)$ will only consist of a part of $\mathbf{f}(t)$.

and the degree k polynomial, also defined over $[0,1]$, as

$$g(u) = \sum_{p=0}^{k} a_p B_p^k(u).$$

A solution to this problem is given by deRose [DeR88]: For any $s \in 0,\ldots,m$,

$$\tilde{\mathbf{f}}(u) = \mathbf{f}(g(u)) = \sum_{i=0}^{m-s} B_i^{m-s}(g(u)) \sum_{r=0}^{ks} \mathbf{b}_{i,r}^s B_r^{ks}(u) \tag{2.11}$$

with $\mathbf{b}_{i,r}^s$, $i = 0,\ldots,m-s$, $r = 0,\ldots,ks$ defined as

$$\mathbf{b}_{i,r}^s = \sum_{\substack{i_1,\ldots,i_s \in \{0,\ldots,k\} \\ i_1+\cdots+i_s = r}} \mathcal{C}_r(i_1,\ldots,i_s)\Delta(0^{<m-s-i>}, a_{i_1},\ldots,a_{i_s}, 1^{<i>}),$$

and $\mathcal{C}_r(i_1,\ldots,i_s)$ being combinatorial constants defined as

$$\mathcal{C}_r(i_1,\ldots,i_s) = \frac{\binom{k}{i_1}\binom{k}{i_2}\cdots\binom{k}{i_s}}{\binom{ks}{r}}. \tag{2.12}$$

The parameter s defines how much of the curve is parametrised in u and how much in $g(u)$. $s = 0$ means the whole curve is parametrised in $g(u)$, whereas with $s = m$ the curve is parametrised in u. We will now look at Equation (2.11) with $s = 0$ and $s = m$.

With $s = 0$, Equation (2.11) becomes

$$\begin{aligned}
\tilde{\mathbf{f}}(u) &= \sum_{i=0}^{m} B_i^m(g(u)) \sum_{r=0}^{0} \mathbf{b}_{i,r}^0 B_r^0(u) \\
&= \sum_{i=0}^{m} \Delta(0^{<m-i>}, 1^{<i>}) B_i^m(g(u)) \\
&= \mathbf{f}(g(u)).
\end{aligned}$$

The more interesting case occurs if we set $s = m$. Now the whole curve is parametrised in u. Equation (2.11) becomes

$$\begin{aligned}
\tilde{\mathbf{f}}(u) &= \sum_{i=0}^{0} B_i^0(g(u)) \sum_{r=0}^{km} \mathbf{b}_{i,r}^m B_r^{km}(u) \\
&= \sum_{r=0}^{km} \mathbf{b}_{0,r}^m B_r^{km}(u) \\
&= \sum_{r=0}^{km} \tilde{\mathbf{b}}_r B_r^{km}(u),
\end{aligned} \tag{2.13}$$

where

$$\tilde{\mathbf{b}}_r = \mathbf{b}_{0,r}^m = \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,k\} \\ i_1+\cdots+i_m = r}} C_r(i_1,\ldots,i_m)\Delta(a_{i_1},\ldots,a_{i_m}) \tag{2.14}$$

are the Bézier points for the reparametrised curve $\tilde{\mathbf{f}}(u)$.

We can use this to find the composed Bézier function $\tilde{\mathbf{f}}(u)$ of two Bézier functions $\mathbf{f}(t)$, $g(u)$ with $\tilde{\mathbf{f}}(u) = \mathbf{f}(g(u))$. The composition of two functions of degree m and k is of degree mk. With Equation (2.14), the new Bézier points $\tilde{\mathbf{b}}_r$ can be found using the following two-step algorithm:

1. Using repeated linear interpolation, generate all blossoms

$$\Delta(a_{i_1},\ldots,a_{i_m}) \quad \text{with } i_1,\ldots,i_m \in \{0,\ldots,k\}.$$

2. Construct the new Bézier points $\tilde{\mathbf{b}}_r$ as convex combinations of all blossoms $\Delta(a_{i_1},\ldots,a_{i_m})$ whose argument indices sum up to r.

Example 3 *Problem: Given a quadratic Bézier curve defined over the interval* $[0,1]$

$$\mathbf{f}(t) = \sum_{i=0}^{2} \Delta(0^{<2-i>}, 1^{<i>}) B_i^2(t),$$

and a polynomial $g(u)$ *with Bernstein coefficients* a_0, a_1, a_2,

$$g(u) = \sum_{p=0}^{2} a_p B_p^2(u), \quad u \in [0,1],$$

find the Bézier points of the reparametrised curve

$$\mathbf{f}(g(u)) = \tilde{\mathbf{f}}(u) = \sum_{r=0}^{4} \tilde{\mathbf{b}}_r B_r^m(u).$$

Solution: The new Bézier points $\tilde{\mathbf{b}}_r = \mathbf{b}_{0,r}^2$ *can be found as convex combinations of all blossoms* $\Delta(a_{i_1}, a_{i_2})$ *where* $i_1 + i_2 = r$ *(Equation (2.13)):*

$$\tilde{\mathbf{b}}_r = \sum_{\substack{i_1,i_2 \in \{0,1,2\} \\ i_1+i_2 = r}} C_r(i_1,i_2)\Delta(a_{i_1},a_{i_2}),$$

where $C_r(i_1,i_2)$ *are the combinatorial constants given in Equation (2.12).*

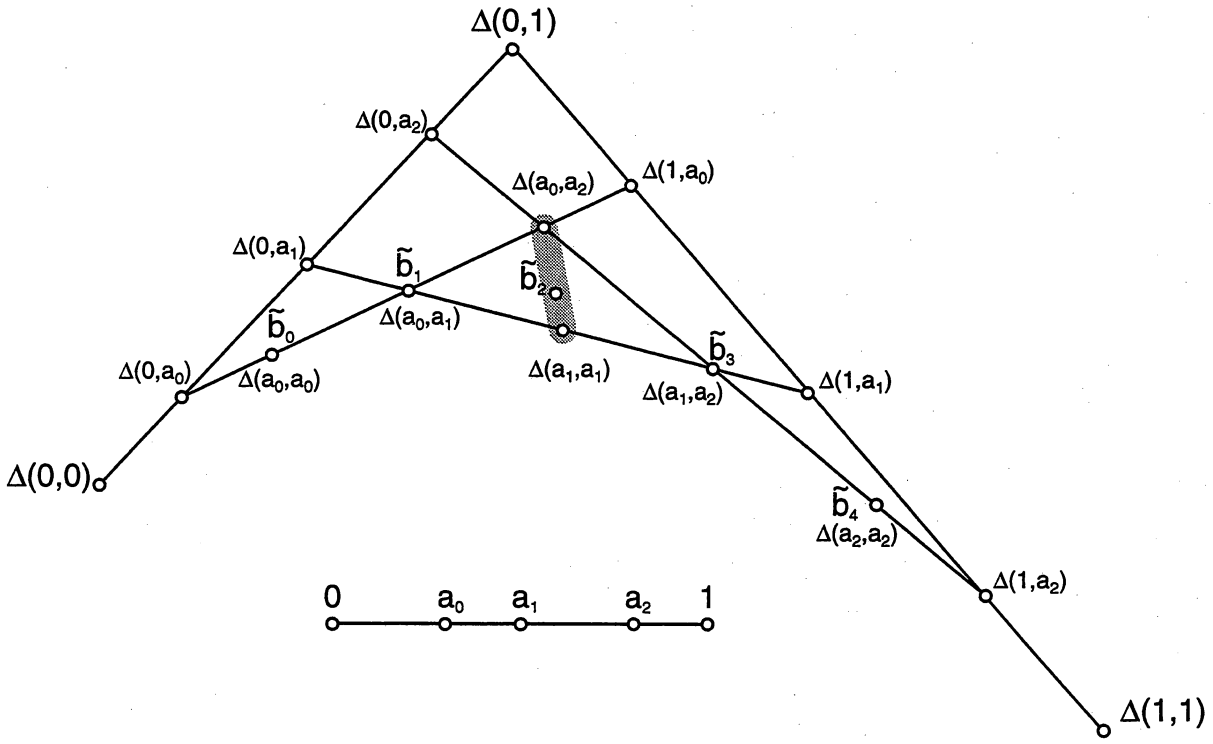


Figure 2.5: Composition of a quadratic Bézier curve and a quadratic polynomial

The first step consists of using repeated linear interpolation to generate all blossoms $\Delta(a_{i_1}, a_{i_2})$ *with* $i_1, i_2 \in \{0, 1, 2\}$. *Owing to symmetry, this yields blossoms* $\Delta(a_0, a_0), \Delta(a_0, a_1), \Delta(a_1, a_1), \Delta(a_1, a_2), \Delta(a_2, a_2)$ *(Figure 2.5).*

As the second step, we generate the new Bézier points $\tilde{\mathbf{b}}_r$ *as convex combinations of these blossoms:*

$$\begin{aligned}
\tilde{\mathbf{b}}_0 &= 1\Delta(a_0, a_0) \\
\tilde{\mathbf{b}}_1 &= \frac{1}{2}\Delta(a_0, a_1) + \frac{1}{2}\Delta(a_1, a_0) \\
&= \Delta(a_0, a_1) \\
\tilde{\mathbf{b}}_2 &= \frac{1}{6}\Delta(a_0, a_2) + \frac{2}{3}\Delta(a_1, a_1) + \frac{1}{6}\Delta(a_2, a_0) \\
&= \frac{1}{3}\Delta(a_0, a_2) + \frac{2}{3}\Delta(a_1, a_1) \\
\tilde{\mathbf{b}}_3 &= \frac{1}{2}\Delta(a_1, a_2) + \frac{1}{2}\Delta(a_2, a_1) \\
&= \Delta(a_1, a_2) \\
\tilde{\mathbf{b}}_4 &= 1\Delta(a_2, a_2)
\end{aligned}$$

Figure 2.5 shows the reparametrised curve with its new control points $\tilde{\mathbf{b}}_r$.

We can now see why evaluation and subdivision can be considered special cases of the general composition problem. Evaluation corresponds to composition with a constant function, whereas subdivision can be viewed as composition with a linear function.

Composition of $\mathbf{f}(t)$ with a constant function $g(u) = a_0$ is equivalent to evaluating $\mathbf{f}(t)$ at a_0. With $k = r = 0$ and $s = m$, Equation (2.12) becomes

$$\begin{aligned}
\tilde{\mathbf{f}}(u) &= \sum_r \mathbf{b}_{0,r}^m B_r^{km}(u) \\
&= \mathbf{b}_{0,0}^m
\end{aligned}$$

Subdivision of $\mathbf{f}(t)$ corresponds to composition with a linear function $g(u)$ with Bernstein coefficients a_0, a_1. With $k = 1$ and $r = 0, \ldots, m$ we obtain

$$\begin{aligned}
\tilde{\mathbf{f}}(u) &= \sum_r \mathbf{b}_{0,r}^m B_r^m(u) \\
&= \sum_r \Delta(a_0^{<m-r>}, a_1^{<r>}) B_r^m(u)
\end{aligned}$$

2.3 B-splines

The blossoming principle not only simplifies the notation of Bézier curves, it is also extremely useful for the derivation of B-spline theory. Again, we do not provide a complete introduction to B-splines; for a detailed treatment refer to Farin [Far93].

Given a non decreasing knot vector

$$T = \{t_0, \ldots, t_{L+2n-2}\} \quad \text{with } t_i \in \mathcal{R}$$

and

$$t_i \leq t_{i+1} \quad \text{for } i = 0, \ldots, L + 2n - 3.$$

u can then be expressed as an affine combination of any two elements t_i, t_j in the knot vector T with $t_i \neq t_j$:

$$u = \frac{t_j - u}{t_j - t_i} t_i + \frac{u - t_i}{t_j - t_i} t_j$$

Let $\mathbf{f}(u) = \Delta(u^{<n>})$ be a degree n polynomial curve with $t_i \leq u < t_{i+1}$. We can use the multiaffine invariance property of blossoms to generate intermediate points:

$$\begin{aligned} &\Delta(\underbrace{t_{i-(n-l-j-1)}, \ldots, t_i}_{n-l-j}, u^{<l>}, \underbrace{t_{i+1}, \ldots, t_{i+j}}_{j}) \qquad (2.15) \\ &= \frac{t_{i+j+1} - u}{t_{i+j+1} - t_{i-(n-l-j)}} \Delta(\underbrace{t_{i-(n-l-j)}, \ldots, t_i}_{n-l-j+1}, u^{<l-1>}, \underbrace{t_{i+1}, \ldots, t_{i+j}}_{j}) \\ &\quad + \frac{u - t_{i-(n-l-j)}}{t_{i+j+1} - t_{i-(n-l-j)}} \Delta(\underbrace{t_{i-(n-l-j-1)}, \ldots, t_i}_{n-l-j}, u^{<l-1>}, \underbrace{t_{i+1}, \ldots, t_{i+j+1}}_{j+1}) \end{aligned}$$

Figure 2.6 shows a schematic representation of the recursive generation of blossom values for a cubic B-spline ($n = 3$). It can be seen that the value of $\mathbf{f}(u)$ with $t_i \leq u < t_{i+1}$ is in fact only dependent on $n + 1$ blossoms

$$\Delta(t_{i-(n-j-1)}, \ldots, t_{i+j}), \quad j = 0, \ldots, n. \qquad (2.16)$$

We have shown that the shape of a Bézier curve is determined by its Bézier polygon, the shape of a B-spline curve is determined by its knot vector and

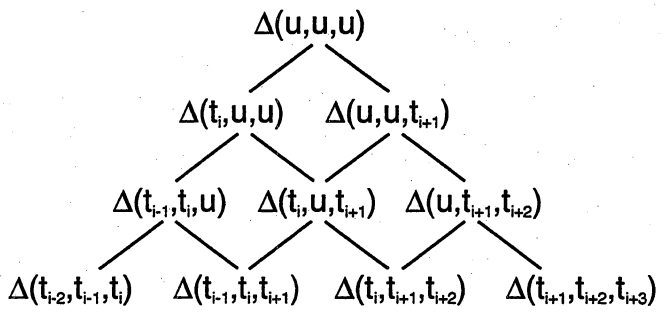


Figure 2.6: Recursive Generation of intermediate points for a cubic B-spline curve

its B-spline polygon. The vertices of the B-spline control polygon, known as *de Boor points* in standard text books, are given by

$$\mathbf{d}_i = \Delta(t_i, \ldots, t_{i+n-1})$$

Using these points, we can also express a B-spline curve as

$$\mathbf{f}(u) = \sum_{i=0}^{L+n-1} \mathbf{d}_i N_i^n,$$

where N_i^n are defined by the *Cox-de Boor recurrence relationship* (de Boor [dB72]).

$$N_i^n = \frac{u - t_i}{t_{i+n} - t_i} N_i^{n-1} + \frac{t_{i+n+1} - u}{t_{i+n+1} - t_i} N_{i+1}^{n-1}$$

and

$$N_i^n = \begin{cases} 1 & \text{if } t_i \leq u < t_{i+1} \\ 0 & \text{else} \end{cases}$$

Figure 2.7 shows a cubic B-spline curve with knot vector $\{0,0,0,1,2,4,5,5,5\}$ and de Boor points $\{0,0,0\}$, $\{0,0,1\}$, $\{0,1,2\}$, $\{1,2,4\}$, $\{2,4,5\}$, $\{4,5,5\}$, $\{5,5,5\}$. It can be seen that the curve actually consists of four Bézier curves over the parameter ranges $[0,1]$, $[1,2]$, $[2,4]$, $[4,5]$. The first Bézier curve, for instance, is defined over the interval $[0,1]$ and has control points $\Delta(0,0,0)$, $\Delta(0,0,1)$, $\Delta(0,1,1)$, $\Delta(1,1,1)$. This allows us to subdivide the B-spline curve into Bézier curves, which is useful for example if we want to render the B-spline and have dedicated hardware to render Bézier curves efficiently.

In general, a Bézier curve that defines a segment of a B-spline curve in the interval $[t_i, t_{i+1}]$, has control points

$$\mathbf{b}_i = \Delta(t_i^{<n-i>}, t_{i+1}^{<i>}).$$

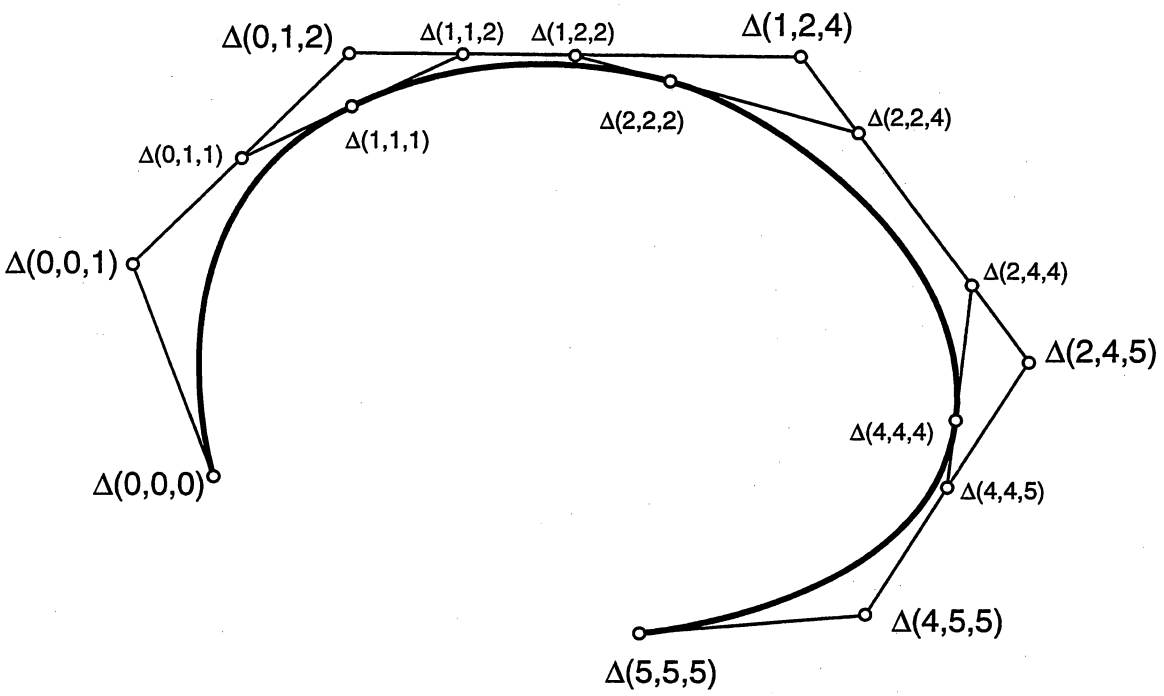


Figure 2.7: The de Boor algorithm for a cubic B-spline curve with knot vector $\{0, 0, 0, 1, 2, 4, 5, 5, 5\}$

2.3.1 Knot Insertion

Inserting a new knot into B-splines adds a new control vertex and can be used to increase the flexibility of a curve much like degree elevation is used for Bézier curves (section 2.2.4).

Let us suppose we have a knot vector

$$U = \{t_0, \ldots, t_i, t_{i+1}, \ldots, t_{L+2n-2}\}$$

and we would like to insert a new knot $\hat{u}$ with $t_i < \hat{u} < t_{i+1}$. What are the new de Boor points?

Once again employing the multiaffine invariance property of blossoms, we can use Equation (2.15) to determine the new de Boor points:

$$\begin{aligned} &\Delta(\underbrace{t_{i-(n-j-2)}, \ldots, t_i}_{n-j-1}, \hat{u}, \underbrace{t_{i+1}, \ldots, t_{i+j}}_{j}) \qquad (2.17) \\ &= \frac{t_{i+j+1} - \hat{u}}{t_{i+j+1} - t_{i-(n-j-1)}} \Delta(t_{i-(n-j-1)}, \ldots, t_{i+j}) \\ &\quad + \frac{\hat{u} - t_{i-(n-j-1)}}{t_{i+j+1} - t_{i-(n-j-1)}} \Delta(t_{i-(n-j-2)}, \ldots, t_{i+j+1}) \end{aligned}$$

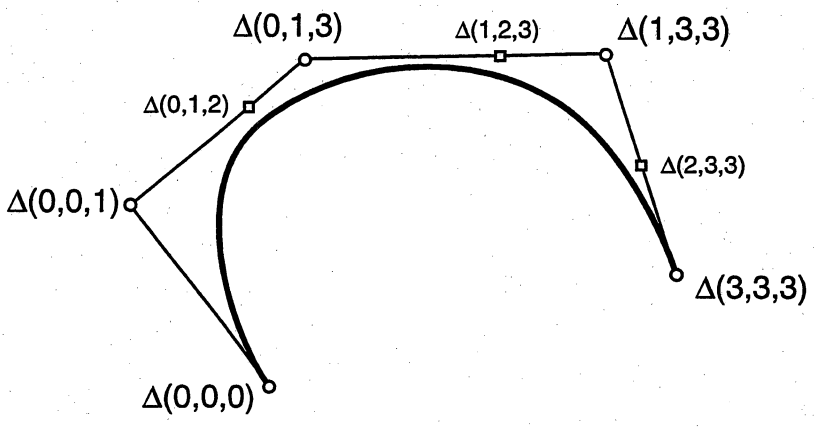


Figure 2.8: Knot insertion into a cubic B-spline curve. The original knot vector is $\{0,0,0,1,3,3,3\}$, we insert knot 2 and obtain the new knot vector $\{0,0,0,1,2,3,3,3\}$

Example 4 *Given a cubic (degree 3) B-spline with knot vector* $\{0,0,0,1,3,3,3\}$ *and de Boor points* $\Delta(0,0,0)$,$\Delta(0,0,1)$,$\Delta(0,1,3)$,$\Delta(1,3,3)$, $\Delta(3,3,3)$ *(Figure 2.8) . We wish to insert a new knot* 2. *What are the new de Boor points?*

The new de Boor points, given by Equation (2.17) with $n = 3$, *are* $\Delta(0,1,2)$, $\Delta(1,2,3)$ *and* $\Delta(2,3,3)$ *and can be obtained by linear interpolation of existing de Boor points:*

$$\begin{aligned} j=0: \quad \Delta(0,1,2) &= \tfrac{3-2}{3-0}\Delta(0,0,1) + \tfrac{2-0}{3-0}\Delta(0,1,3) \\ j=1: \quad \Delta(1,2,3) &= \tfrac{3-2}{3-0}\Delta(0,1,3) + \tfrac{2-0}{3-0}\Delta(1,3,3) \\ j=2: \quad \Delta(2,3,3) &= \tfrac{3-2}{3-1}\Delta(1,3,3) + \tfrac{2-1}{3-1}\Delta(3,3,3) \end{aligned}$$

2.3.2 Continuity

We have shown that B-splines are actually piecewise continuous Bézier curves. Two degree n B-spline segments $\Delta_1()$ and $\Delta_2()$, defined over parameter ranges $[t_0, t_1]$ and $[t_1, t_2]$ respectively, join with C^k continuity at t_1 if

$$\Delta_1(u_1, \ldots, u_k, t_1^{<n-k>}) = \Delta_2(u_1, \ldots, u_k, t_1^{<n-k>}),$$

that is if they agree in all blossoms that include at least $(n-k)$ copies of t_1. Evaluating the kth derivative involves, according to Equation (2.6), varying at most k arguments, so if $\Delta_1()$ and $\Delta_2()$ agree in all these arguments, they must be C^k continuous.

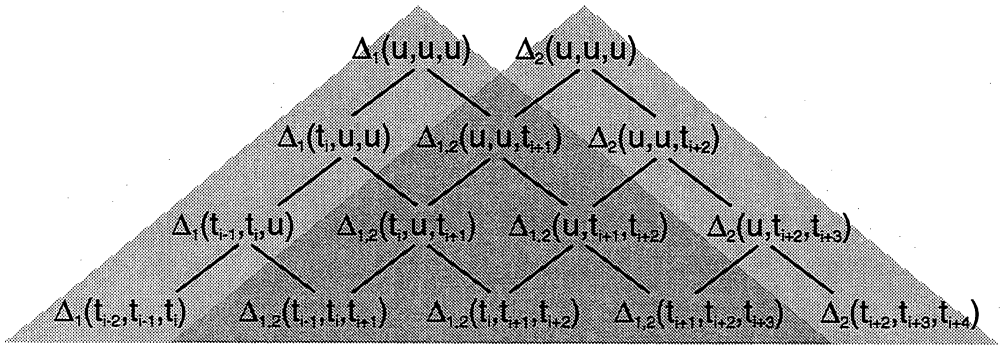


Figure 2.9: De Boor points for two adjacent cubic B-Spline segments

Figure 2.9 shows the overlapping schemes for the generation of two adjacent cubic B-spline curves $\Delta_1()$ and $\Delta_2()$, defined over the interval $[t_i, t_{i+1}]$ and $[t_{i+1}, t_{i+2}]$, respectively. If we use this scheme to calculate the curve at

parameter value t_{i+1}, we see that

$$\begin{aligned}\Delta_1(t_{i+1}, t_{i+1}, t_{i+1}) &= \Delta_2(t_{i+1}, t_{i+1}, t_{i+1}) \\ \Delta_1(u_1, t_{i+1}, t_{i+1}) &= \Delta_2(u_1, t_{i+1}, t_{i+1}) \\ \Delta_1(u_1, u_2, t_{i+1}) &= \Delta_2(u_1, u_2, t_{i+1}).\end{aligned}$$

The curves agree in all blossoms that contain at least 1 copy of t_{i+1} and thus meet with C^2 continuity.

Generally, two degree n spline segments meet with at least C^{n-r} continuity at a knot with multiplicity r.

2.4 Surfaces

Let us now extend the Bézier and B-spline ideas to surfaces. Surface patches are generally either triangular or rectangular. The most natural generalisation of Bézier curves are Bézier triangles (de Casteljau [dFdC59]), but rectangular shaped tensor product Bézier patches have also gained wide popularity. For a comprehensive survey and classification of different types of surface patches, see Barnhill [Bar85]. Our discussion starts with Bézier triangles, as they are the most natural extension of Bézier curves to surfaces.

2.4.1 Bézier Triangles

Mathematically, a polynomial surface is a polynomial function with a two-dimensional domain, so we have to extend our blossoming notation to multivariate functions. Let $\mathbf{f}(\mathbf{u})$ be a be a polynomial function $\mathcal{R}^2 \rightarrow \mathcal{R}^d$ of degree n. The corresponding blossom $\Delta(\mathbf{u}_1, \ldots, \mathbf{u}_n) : (\mathcal{R}^2)^n \rightarrow \mathcal{R}^d$ has the following properties:

- It produces $\mathbf{f}$ on the *diagonal*:

$$\mathbf{f}(\mathbf{u}) = \Delta(\mathbf{u}, \ldots, \mathbf{u})$$

- It is *symmetric* for any ordering of variables:

$$\Delta(\mathbf{u}_1, \ldots, \mathbf{u}_n) = \Delta(\mathbf{u}_{i_1}, \ldots, \mathbf{u}_{i_n})$$

- It is *multiaffine*:

$$\Delta(\mathbf{u}_1, \ldots, \sum_i \alpha_i \mathbf{u}_{j_i}, \ldots, \mathbf{u}_n) = \sum_i \alpha_i \Delta(\mathbf{u}_1, \ldots, \mathbf{u}_{j_i}, \ldots, \mathbf{u}_n)$$

  for all $j = 1, \ldots, n$.

In our discussion of Bézier curves, we expressed the parameter $\mathbf{u}$ as an affine combination of the endpoints of the interval $[0, 1]$. For surfaces, we will express $\mathbf{u}$ as an affine combination of a reference frame $\mathbf{e1}, \mathbf{e2}, \mathbf{e3}$, with $\mathbf{e1} = (1, 0, 0)$, $\mathbf{e2} = (0, 1, 0)$, $\mathbf{e3} = (0, 0, 1)$. These three vertices define a plane and each point $\mathbf{u}$ in the plane can be expressed uniquely as an affine combination of these points:

$$\mathbf{u} = u_0\,\mathbf{e1} + u_1\,\mathbf{e2} + u_2\,\mathbf{e3}, \qquad u_0 + u_1 + u_2 = 1.$$

The development of Bézier patches is strictly analogous to Bézier curves. Let $\mathbf{i}$ be a multiindex with

$$\mathbf{i} \equiv (i_0, i_1, i_2).$$

We can use the multivariate invariance property to construct intermediate blossoms by repeated linear interpolation:

$$\begin{aligned}
&\Delta(\mathbf{u}^{<n-|\mathbf{i}|>}, \mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>}) = \\
&\quad u_0\,\Delta(\mathbf{u}^{<n-|\mathbf{i}|-1>}, \mathbf{e1}^{<i_0+1>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>}) \\
&\quad + u_1\,\Delta(\mathbf{u}^{<n-|\mathbf{i}|-1>}, \mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1+1>}, \mathbf{e3}^{<i_2>}) \\
&\quad + u_2\,\Delta(\mathbf{u}^{<n-|\mathbf{i}|-1>}, \mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2+1>})
\end{aligned}$$

and

$$\mathbf{f}(u) = \Delta(\mathbf{u}^{<n>}).$$

Figure 2.10 shows the domain triangle $\mathbf{e1}, \mathbf{e2}, \mathbf{e3}$ and the intermediate points for the evaluation of a point on the surface of a Bézier triangle.

As in the Bézier curve treatment, we can now use this to derive a non recursive expression for Bézier patches:

$$\begin{aligned}
\mathbf{f}(\mathbf{u}) &= \Delta(\mathbf{u}^{<n>}) \\
&= u_0\,\Delta(\mathbf{u}^{<n-1>}, \mathbf{e1}^{<1>}) + u_1\,\Delta(\mathbf{u}^{<n-1>}, \mathbf{e2}^{<1>}) + u_2\,\Delta(\mathbf{u}^{<n-1>}, \mathbf{e3}^{<1>}) \\
&= \sum_{|\mathbf{i}|=n} B^n_{\mathbf{i}}(\mathbf{u})\,\Delta(\mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>}) \qquad (2.18)
\end{aligned}$$

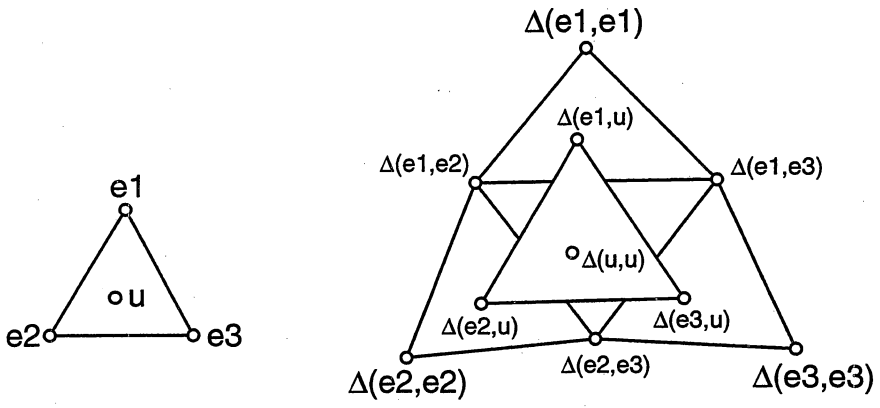


Figure 2.10: The de Casteljau algorithm for a quadratic Bézier triangle

where $B_{\mathbf{i}}^n(\mathbf{u})$ are *multivariate Bernstein polynomials* with

$$B_{\mathbf{i}}^n(\mathbf{u}) = \binom{n}{\mathbf{i}} \mathbf{u}^{\mathbf{i}}.$$

The multinomial coefficients are defined as

$$\binom{n}{\mathbf{i}} \equiv \frac{n!}{i_0! \cdots, i_n!}$$

and

$$\mathbf{u}^{\mathbf{i}} \equiv u_0^{i_0} \cdots u_n^{i_n}.$$

The standard expression for Bézier triangles given in the literature (Barnhill [Bar85]) is

$$\mathbf{f}(\mathbf{u}) = \sum_{|\mathbf{i}|=n} \mathbf{b}_{\mathbf{i}} B_{\mathbf{i}}^n(\mathbf{u}).$$

If we compare this with the expression in Equation (2.18), we can see that the Bézier points can actually be expressed in blossoming notation as

$$\mathbf{b}_{\mathbf{i}} = \Delta(\mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>}).$$

Derivatives

When discussing surfaces, it is appropriate to define a *directional derivative.* We define the direction as the difference of two points $\mathbf{r}$ and $\mathbf{s}$ in the domain

of the surface. With Equation (2.6), the directional derivative along a vector $\mathbf{d} = \mathbf{s} - \mathbf{r}$ is

$$\begin{aligned} \frac{\partial \mathbf{f}(u)}{\partial \mathbf{d}} &= n\Delta(\mathbf{u}^{<n-1>}, \mathbf{d}) \\ &= \sum_{|\mathbf{i}|=n-1} \Delta(\mathbf{d}, \mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>})\, B_{\mathbf{i}}^{n-1}(\mathbf{u}) \end{aligned} \tag{2.19}$$

Composition

The composition problem for Bézier triangles consists of finding the composed triangle $\tilde{\mathbf{f}} = \mathbf{f} \circ \mathbf{g}$, where $\mathbf{f}$ and $\mathbf{g}$ are Bézier triangles. We can extend the composition algorithm for Bézier curves, given in Section 2.2.5, to handle the composition of Bézier triangles:

Let $\mathbf{f}(\mathbf{t}) : \mathcal{R}^2 \to \mathcal{R}^d$ be a Bézier triangle of degree m and $\mathbf{g}(\mathbf{u}) : \mathcal{R}^2 \to \mathcal{R}^2$ be a Bézier triangle of degree k,

$$\begin{aligned} \mathbf{f}(\mathbf{t}) &= \sum_{|\mathbf{i}|=m} \mathbf{b}_{\mathbf{i}} B_{\mathbf{i}}^{m}(\mathbf{t}), \\ \mathbf{g}(\mathbf{u}) &= \sum_{|\mathbf{p}|=k} \mathbf{a}_{\mathbf{p}} B_{\mathbf{p}}^{k}(\mathbf{u}). \end{aligned}$$

We are looking for the control points of the composed Bézier simplex

$$\tilde{\mathbf{f}}(\mathbf{u}) = \mathbf{f}(\mathbf{g}(\mathbf{u})) = \sum_{|\mathbf{r}|=km} \tilde{\mathbf{b}}_{\mathbf{r}} B_{\mathbf{r}}^{km}(\mathbf{u})$$

DeRose [DeR88] derives an extension of the composition formula for Bézier curves (Equation (2.11)):
For any $s \in 0, \ldots, m$,

$$\tilde{\mathbf{f}}(\mathbf{u}) = \mathbf{f}(\mathbf{g}(\mathbf{u})) = \sum_{|\mathbf{i}|=m-s} B_{\mathbf{i}}^{m-s}(\mathbf{g}(\mathbf{u})) \sum_{|\mathbf{r}|=ks} \mathbf{b}_{\mathbf{i},\mathbf{r}}^{s} B_{\mathbf{r}}^{ks}(\mathbf{u}) \tag{2.20}$$

with $\mathbf{b}_{\mathbf{i},\mathbf{r}}^{s}$, $|\mathbf{i}| = m - s$, $|\mathbf{r}| = ks$ defined as

$$\mathbf{b}_{\mathbf{i},\mathbf{r}}^{s} = \sum_{\substack{|\mathbf{i}_1| = \cdots = |\mathbf{i}_s| = k \\ \mathbf{i}_1 + \cdots + \mathbf{i}_s = \mathbf{r}}} \mathcal{C}_{\mathbf{r}}(\mathbf{i}_1, \ldots, \mathbf{i}_s) \Delta(\mathbf{a}_{\mathbf{i}_1}, \ldots, \mathbf{a}_{\mathbf{i}_s}, \mathbf{e1}^{<i_0>}, \mathbf{e2}^{<i_1>}, \mathbf{e3}^{<i_2>}),$$

and $\mathcal{C}_{\mathbf{r}}(\mathbf{i}_1, \ldots, \mathbf{i}_s)$ being combinatorial constants defined as

$$\mathcal{C}_{\mathbf{r}}(\mathbf{i}_1, \ldots, \mathbf{i}_s) = \frac{\binom{k}{\mathbf{i}_1}\binom{k}{\mathbf{i}_2}\cdots\binom{k}{\mathbf{i}_s}}{\binom{ks}{\mathbf{r}}}. \tag{2.21}$$

Again, the parameter s defines to what degree the curve is parametrised in $\mathbf{u}$ and to what degree in $\mathbf{g(u)}$. We are particularly interested in the case $s = m$, because then the whole curve is parametrised in u. Equation (2.20) becomes

$$\begin{aligned} \tilde{\mathbf{f}}(\mathbf{u}) &= \sum_{|\mathbf{i}|=0} B_{\mathbf{i}}^{0}(\mathbf{g(u)}) \sum_{|\mathbf{r}|=km} \mathbf{b}_{\mathbf{i},\mathbf{r}}^{m} B_{\mathbf{r}}^{km}(\mathbf{u}) \\ &= \sum_{|\mathbf{r}|=km} \mathbf{b}_{\mathbf{0},\mathbf{r}}^{m} B_{\mathbf{r}}^{km}(\mathbf{u}) \\ &= \sum_{|\mathbf{r}|=km} \tilde{\mathbf{b}}_{\mathbf{r}} B_{\mathbf{r}}^{km}(\mathbf{u}), \end{aligned} \tag{2.22}$$

where

$$\tilde{\mathbf{b}}_{\mathbf{r}} = \mathbf{b}_{\mathbf{0},\mathbf{r}}^{m} = \sum_{\substack{|\mathbf{i}_1| = \cdots = |\mathbf{i}_m| = k \\ \mathbf{i}_1 + \cdots + \mathbf{i}_m = \mathbf{r}}} C_{\mathbf{r}}(\mathbf{i}_1, \ldots, \mathbf{i}_m) \Delta(\mathbf{a}_{\mathbf{i}_1}, \ldots, \mathbf{a}_{\mathbf{i}_m}) \tag{2.23}$$

are the Bézier points for the reparametrised curve $\tilde{\mathbf{f}}(\mathbf{u})$.

We can use this to find the composed Bézier simplex $\tilde{\mathbf{f}}(\mathbf{u})$ of two Bézier functions $\mathbf{f(t)}$, $\mathbf{g(u)}$ with $\tilde{\mathbf{f}}(\mathbf{u}) = \mathbf{f}(\mathbf{g(u)})$:

The new Bézier points $\tilde{\mathbf{b}}_{\mathbf{r}}$ can be found using the following two-step algorithm:

1. Using repeated linear interpolation, generate all blossoms

$$\Delta(\mathbf{a}_{\mathbf{i}_1}, \ldots, \mathbf{a}_{\mathbf{i}_m}) \quad \text{with } |\mathbf{i}_1| = \cdots = |\mathbf{i}_m| = k.$$

2. Construct the new Bézier points $\tilde{\mathbf{b}}_{\mathbf{r}}$ as convex combinations of all blossoms $\Delta(\mathbf{a}_{\mathbf{i}_1}, \ldots, \mathbf{a}_{\mathbf{i}_m})$

   - whose argument indices sum up to $\mathbf{r}$ and
   - all argument index metrices equal k.

2.4.2 Tensor Product Bézier Patches

One other popular variant of Bézier based surfaces is the tensor product patch. A useful way of looking at a Bézier patch is to see it as a Bézier curve that sweeps through space, because each of its control points changes its position along another Bézier curve in space.

We can make this explanation more precise. Given a Bézier curve of degree m

$$\mathbf{f}(u) = \sum_{i=0}^{m} \mathbf{b}_i B_i^m(u),$$

we can let each $\mathbf{b}_i$ trace out a different Bézier curve of degree n

$$\mathbf{b}_i = \mathbf{b}_i(v) = \sum_{j=0}^{n} \mathbf{b}_{i,j} B_j^n(v).$$

If we combine these two equations, we obtain the tensor product Bézier patch $\mathbf{f}(u,v) : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^d$ as

$$\mathbf{f}(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{b}_{i,j} B_i^m(u) B_j^n(v).$$

The corresponding blossom $\Delta(u_1, \ldots, u_m; v_1, \ldots, v_n) : \mathcal{R}^m \times \mathcal{R}^n \to \mathcal{R}^d$ is obtained by polarising each variable separately and has the following properties:

- It produces $\mathbf{f}(u,v)$ on the *diagonal*:

$$\mathbf{f}(u,v) = \Delta(u, \ldots, u; v, \ldots, v) \tag{2.24}$$

- It is *symmetric* for any ordering of variables with each parameter group:

$$\Delta(u_1, \ldots, u_m; v_1, \ldots, v_n) = \Delta(u_{i_1}, \ldots, u_{i_m}; v_{j_1}, \ldots, v_{j_n}) \tag{2.25}$$

- It is *multiaffine* in each variable u, v separately.

The Bézier points of a degree (m,n) tensor product Bézier patch over the interval $[p,q] \times [r,s]$ can be expressed in Blossom notation as

$$\mathbf{b}_{i,j} = \Delta(p^{<m-i>}, q^{<i>}; r^{<n-j>}, s^{<j>}).$$

Figure 2.11 shows the control point net for a bicubic (3×3) Bézier patch. Note that the patch edges are actually Bézier curves of degree m and n, respectively. In fact, any isoparametric curve $v = \text{const}$ is a Bézier curve of degree m, whereas isoparametric curves with $u = \text{const}$ are Bézier curves of degree n. We will see in the next chapter that generally non isoparametric straight lines on the patch are of degree $m + n$.

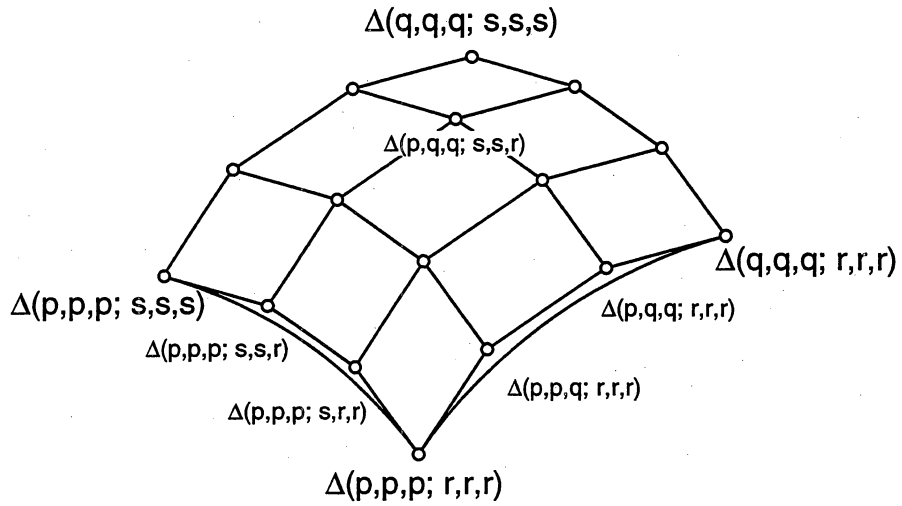


Figure 2.11: The control point net for a bicubic tensor product Bézier patch

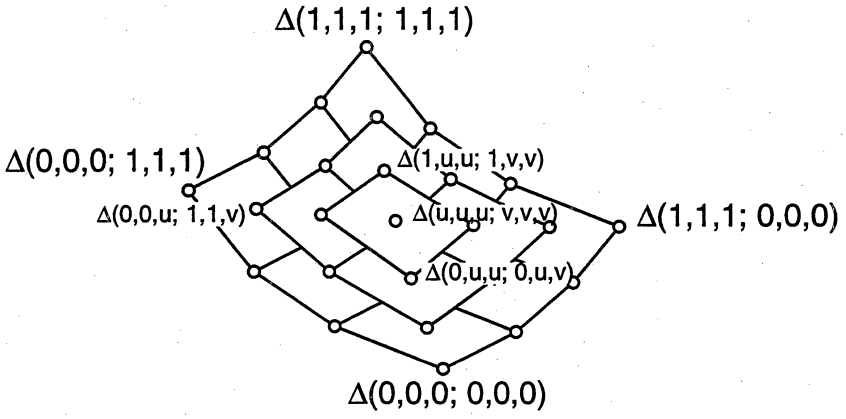


Figure 2.12: Direct evaluation of a point (u, v) on the surface of a bicubic tensor product Bézier patch over the interval $[0, 1] \times [0, 1]$

A point on the surface of a tensor product Bézier surface $\mathbf{f}(u,v)$ can be expressed in blossom notation as

$$\mathbf{f}(u,v) = \Delta(u,\ldots,u;v,\ldots,v).$$

There are two ways of evaluating a point on a Bézier patch, which we shall call *two step evaluation* and *direct evaluation*.

The two step evaluation utilises the fact that tensor product patches can be thought of as curves of curves. Starting from the control points

$$\mathbf{b}_{i,j} = \Delta(0^{<m-i>},1^{<i>};0^{<n-j>},1^{<j>}),$$

it first uses linear interpolation to generate the Bézier points of an isoparametric line with $u = const$

$$\mathbf{b}_i = \Delta(0^{<m-i>},1^{<i>};v^{<n>}),$$

and subsequently linearly interpolates between these intermediate points with parameter value u to obtain the surface point

$$\mathbf{f}(u,v) = \Delta(u^{<m>};v^{<n>}).$$

The direct evaluation approach is shown in Figure 2.12. It uses bilinear interpolation to directly compute blossom values

$$\Delta(0^{<m-i-k>},u^k,1^{<i>};0^{<n-j-k>},v^k,1^{<j>})$$

with $k = 0,\ldots,\min(m,n)$. If the surface is of different degree in u and v ($m \neq n$), surface points can not be obtained using only bilinear interpolation. In this case, the final steps are performed using linear interpolation (see Farin [Far93], Chapter 16, for more details, although not in blossoming notation, on evaluating patches of different degrees in u and v).

Derivatives

The computation of the partial derivative of tensor product patches along their parameters u and v is a straightforward extension of the derivative of a Bézier curve. We derive the formula for the u-partial:

$$\begin{aligned}\frac{\partial \mathbf{f}(u,v)}{\partial u} &= \sum_{j=0}^{n}\left[\frac{\partial}{\partial u}\sum_{i=0}^{m}\mathbf{b}_{i,j}B_i^m(u)\right]B_j^n(v)\\ &= m\sum_{j=0}^{n}\sum_{i=0}^{m-1}(\mathbf{b}_{i+1,j}-\mathbf{b}_{i,j})\,B_i^{m-1}(u)B_j^n(v) \qquad (2.26)\end{aligned}$$

2.4.3 Tensor Product B-spline Patches

Just as Bézier patches, a tensor product B-spline patch can be defined by two B-spline curves as

$$\mathbf{f}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{d}_{i,j} N_i^m(u) N_j^n(v).$$

A de Boor point $\mathbf{d}_{i,j}$ of a degree (m,n) B-spline can be expressed in blossom notation as

$$\mathbf{d}_{i,j} = \Delta(p^{<m-i>}, q^{<i>}; r^{<n-j>}, s^{<j>}).$$

Just as B-spline curves can be thought of as a collection of Bézier segments with some degree of continuity across joints, B-spline patches are actually a regular array of Bézier patches with some degree of continuity across common edges.

The Bézier points of an individual Bézier patch defined over the interval $[s_i, s_{i+1}] \times [t_j, t_{j+1}]$ can be expressed in blossoming notation as

$$\mathbf{b}_{i,j} = \Delta(s_i^{<m-i>}, s_{i+1}^{<i>}; t_j^{<n-j>}, t_{j+1}^{<j>})$$

2.5 Higher Dimensional Bézier Patches

The concept of Bézier triangles and tensor product patches can easily be extended to higher degrees. Although difficult to visualise, the three dimensional objects can perform a mapping $\mathcal{R}^3 \rightarrow \mathcal{R}^3$ and are thus useful in defining deformations of objects in space. We will briefly introduce them here because they constitute the deformation basis for Free Form Deformation and will also be used in our discussions on composition algorithms in Chapter 3.

2.5.1 Bézier Simplexes

The generalisation of Bézier triangles are Bézier simplexes. The three dimensional simplex is called a Bézier tetrahedron. A general Bézier simplex of degree n can be expressed as

$$\mathbf{f}(\mathbf{u}) = \sum_{|\mathbf{i}|=n} \mathbf{b}_\mathbf{i} B_\mathbf{i}^n(\mathbf{u}),$$

or, in blossoming notation

$$\mathbf{f(u)} = \sum_{|\mathbf{i}|=n} \Delta(\mathbf{e1}^{i_0}, \mathbf{e2}^{i_1}, \ldots, \mathbf{e(n+1)}^{i_n}) B_{\mathbf{i}}^{n}(\mathbf{u}).$$

The labelling scheme for a quadratic simplex, a Bézier tetrahedron, is shown in Figure 2.13.

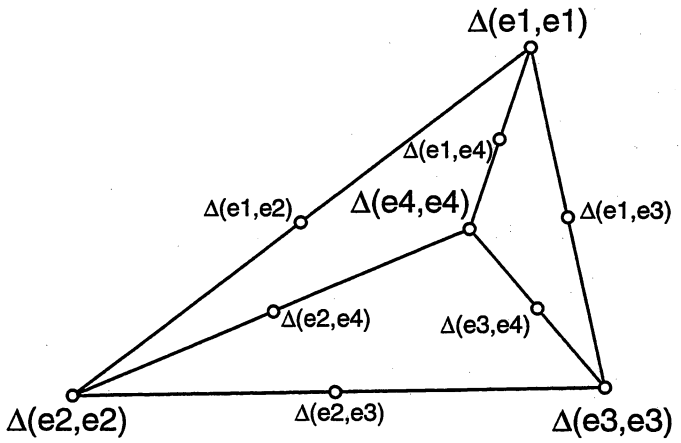


Figure 2.13: Control points of a Bézier tetrahedron

2.5.2 Tensor Product Hypersurfaces

Tensor product surfaces can be extended to tensor product hypersurfaces. A Bézier volume, the three dimensional version of a tensor product hypersurface, is a mapping $f(u,v,w) : \mathcal{R} \times \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^d$

$$\mathbf{f}(u,v,w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{b}_{i,j,k} B_i^m(u) B_j^n(v) B_k^p(w).$$

If the Bézier curves are of degree m, n, p, the corresponding blossom is

$$\Delta(u_1, \ldots, u_m; v_1, \ldots, v_n; w_1, \ldots, w_p) : \mathcal{R}^m \times \mathcal{R}^n \times \mathcal{R}^p \rightarrow \mathcal{R}^d.$$

In Figure 2.14 we show the blossom labelling scheme for the control points of a triquadratic $(2,2,2)$ Bézier volume over the interval $[0,1] \times [0,1] \times [0,1]$.

By keeping one or two parameters constant, one can easily verify that the volume edges map into Bézier curves, whereas the sides map into Bézier surfaces. A general straight line in parameter space, however, maps into a degree $[m+n+p]$ Bézier curve.

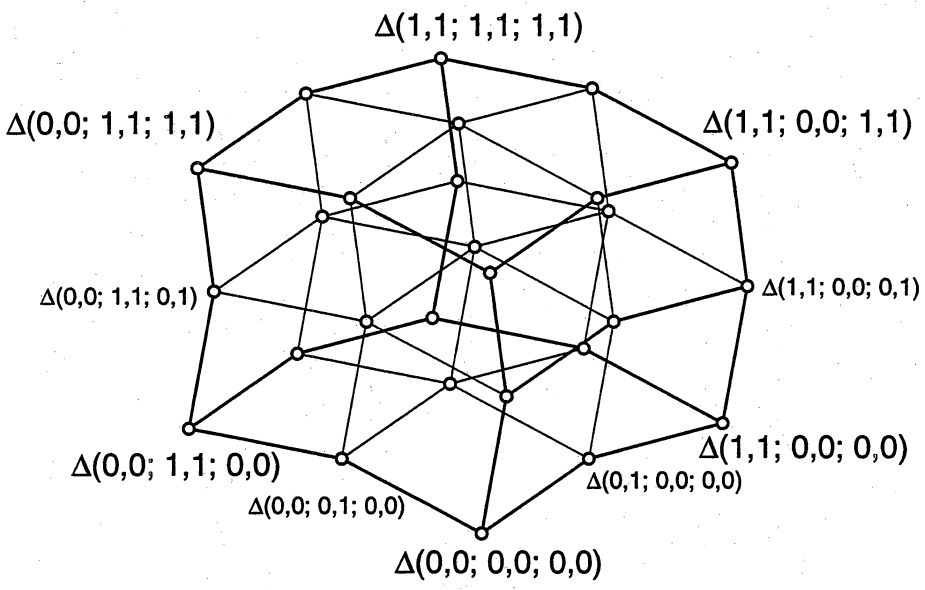


Figure 2.14: The control point net for a triquadratic tensor product Bézier Volume

2.6 Summary

We have described Bézier curves using the blossoming principle and we have shown how blossoming facilitates the introduction of standard operations such as computation of derivatives, subdivision and composition with polynomial functions. We have then shown two ways of extending the Bézier curve concept to higher dimensions (i.e. surfaces and volumes): Bézier simplices and tensor product patches.

Free Form Deformation uses tensor product Bézier volumes to deform curves and surfaces. Finding analytical expressions for the deformed objects is essentially a functional composition problem. In the next chapter, we show how to use blossoming to find analytic expressions for the composed (deformed) curves and surfaces.

Chapter 3

Composition Algorithms

3.1 Introduction

Let us take a closer look at the mapping between parameter space and world space. In the previous section we stated that for a degree (m, n) tensor product Bézier patch $f(u, v) : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^d$, a general straight line $\mathbf{g}(t) : \mathcal{R} \to \mathcal{R} \times \mathcal{R}$ in parameter space gets mapped into a degree $m + n$ Bézier curve on the surface of the biparametric patch (Figure 3.1). The curve on the patch can be expressed as the composed map $\tilde{\mathbf{g}} : \mathcal{R} \to \mathcal{R}^d = \mathbf{f} \circ \mathbf{g}$

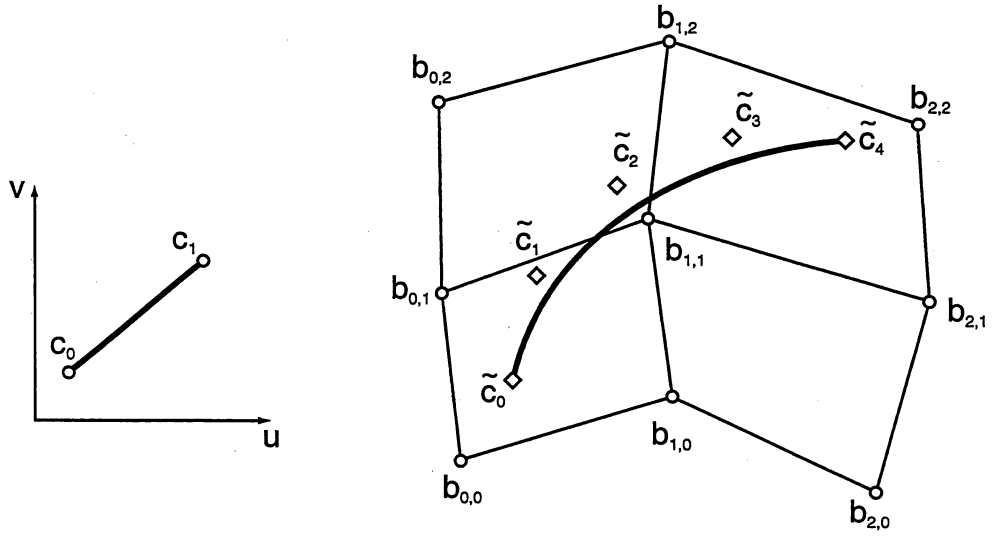


Figure 3.1: Mapping of a straight line in parameter space into world space

Moving the control vertices of the bicubic patch thus enables us to indirectly change the shape of the composed curve $\tilde{\mathbf{g}}$. This is the basis of the Free Form Deformation principle.

Rather then raising the degree of the curve and manipulating the curve directly, we embed a curve in a patch and indirectly manipulate its shape by changing the shape of the embedding solid. The advantage of this indirect manipulation scheme is that it

- can be applied to a number of embedded curves at once
- is independent of the representation of the embedded curve.

We can therefore change a whole object composed of curves in one step, without having to change each of its defining curves individually. Obviously, this principle can be extended to three dimensions to deform curves or surfaces in trivariate hyperpatches.

In this section, we will investigate the deformation of Bézier curves and triangles with tensor product surfaces and volumes. Using the knowledge of functional composition of Bézier curves derived in the previous chapter, we shall develop algorithms to determine analytic expressions $\tilde{\mathbf{g}}$ for curves and surfaces that have undergone indirect deformation.

The first part of this chapter is devoted to the deformation of curves. We investigate the deformation of Bézier curves embedded in tensor product Bézier patches and volumes. In the second part we examine the deformation of surfaces, we therefore extend the curve algorithms to cope with the deformation of Bézier triangles on surface patches and in volumes.

3.2 Composition of a Bézier Patch and a Bézier Curve

We first investigate the mapping of a Bézier curve $\mathbf{g}(t) : \mathcal{R} \rightarrow \mathcal{R} \times \mathcal{R}$ onto a tensor product Bézier patch $\mathbf{f}(u, v) : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^d$. We want to express the resulting curve as a univariate Bézier curve $\tilde{\mathbf{g}}(u) : \mathcal{R} \rightarrow \mathcal{R}^d$ and determine the control points of this new curve.

The problem can be stated as a functional composition problem: Given a bivariate function $\mathbf{f}(u, v)$ and a univariate function $\mathbf{g}(t)$, what is the resulting, composed function

$$\tilde{\mathbf{g}}(t) = \mathbf{f}(\mathbf{g}(t)).$$

3.2.1 Monomial Form Composition Algorithm

We can simplify the problem by first converting the functions into their *monomial* (power basis) form.

For every Bézier function in Bernstein-Bézier form,

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{b}_i B_i^n(u),$$

there is an equivalent power basis form with

$$\mathbf{f}(u) = \sum_{i=0}^{n} \mathbf{a}_i u^i.$$

The choice of the representation depends on the application. The Bernstein-Bézier form with its control points $\mathbf{b}_i$ is geometrically more intuitive, whereas the monomial form can be computationally more convenient[1]. Bernstein bases and power bases are related by

$$u^i = \sum_{j=i}^{n} \frac{\binom{j}{i}}{\binom{n}{i}} B_j^n(u) \tag{3.1}$$

and

$$B_i^n(u) = \sum_{j=i}^{n} (-1)^{j-i} \binom{n}{j} \binom{j}{i} u^j. \tag{3.2}$$

Let $\mathbf{f}(u,v) : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^d$ be a degree (m,n) tensor product Bézier patch in monomial form

$$\mathbf{f}(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{a}_{i,j} u^i v^j$$

and $\mathbf{g}(t) : \mathcal{R} \to \mathcal{R}^2$ be a degree p Bézier curve

$$\mathbf{g}(t) = \sum_{k=0}^{p} \mathbf{c}_k t^k$$

with its projections onto the u, v axes

$$\begin{aligned} g_u(t) &= \sum_{k=0}^{p} c_{u_k} t^k \\ g_v(t) &= \sum_{k=0}^{p} c_{v_k} t^k. \end{aligned}$$

[1] It has been shown by Farouki [Far91] however that the monomial form is numerically less stable, which represents a significant problem for the implementation on finite precision computers.

The composed function $\tilde{\mathbf{g}}(t)$ then becomes

$$\begin{aligned}\tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t)) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{a}_{i,j} \left[g_u(t)\right]^i \left[g_v(t)\right]^j \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{a}_{i,j} \left[\sum_{k=0}^{p} c_{u_k} t^k\right]^i \left[\sum_{k=0}^{p} c_{v_k} t^k\right]^j\end{aligned}$$

Expanding the expression and collecting the coefficients for each power e of t as $\tilde{\mathbf{b}}_e$ finally yields

$$\tilde{\mathbf{g}}(t) = \sum_{e=0}^{p(m+n)} \tilde{\mathbf{a}}_e t^e, \tag{3.3}$$

which is the power form representation of a degree $(p(m+n))$ Bézier curve.

We can now retransform the power basis representation into the Bernstein-Bézier representation and thus obtain the Bézier points of the composed function $\tilde{\mathbf{g}}(t)$.

Example 5 *Given a tensor product biquadratic patch*

$$\mathbf{f}(u, v) = \sum_{i=0}^{2}\sum_{j=0}^{2} \mathbf{a}_{i,j} u^i v^j$$

and a linear curve

$$\mathbf{g}(t) = \sum_{k=0}^{1} \mathbf{c}_k t^k,$$

with its projections onto the u, v axes

$$\begin{aligned} g_u(t) &= \sum_{k=0}^{1} c_{u_k} t^k \\ g_v(t) &= \sum_{k=0}^{1} c_{v_k} t^k. \end{aligned}$$

what are the coefficients of the composed function $\tilde{\mathbf{g}}(t) = \mathbf{f}(g_u(t), g_v(t))$?

$$
\begin{aligned}
\tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t)) \\
&= \sum_{i=0}^{2}\sum_{j=0}^{2} \mathbf{a}_{i,j} \left[g_u(t)\right]^i \left[g_v(t)\right]^j \\
&= \sum_{i=0}^{2}\sum_{j=0}^{2} \mathbf{a}_{i,j} \left[\sum_{k=0}^{1} c_{u_k} t^k\right]^i \left[\sum_{k=0}^{1} c_{v_k} t^k\right]^j \\
&= \mathbf{a}_{2,2} c_{u_1}^2 c_{v_1}^2 t^4 \\
&\quad + \left(\mathbf{a}_{2,1} c_{u_1}^2 c_{v_1} + \mathbf{a}_{1,2} c_{u_1} c_{v_1}^2 + 2\mathbf{a}_{2,2} c_{u_0} c_{u_1} c_{v_1}^2 + 2\mathbf{a}_{2,2} c_{u_1}^2 c_{v_0} c_{v_1}\right) t^3 \\
&\quad + \Big(\mathbf{a}_{1,2} c_{u_0} c_{v_1}^2 + 2\mathbf{a}_{1,2} c_{u_1} c_{v_0} c_{v_1} + \mathbf{a}_{2,0} c_{u_1}^2 + \mathbf{a}_{0,2} c_{v_1}^2 \\
&\qquad + 2\mathbf{a}_{2,1} c_{u_0} c_{u_1} c_{v_1} + \mathbf{a}_{2,1} c_{u_1}^2 c_{v_0} + \mathbf{a}_{1,1} c_{u_1} c_{v_1} + \mathbf{a}_{2,2} c_{u_0}^2 c_{v_1}^2 \\
&\qquad + 4\mathbf{a}_{2,2} c_{u_0} c_{u_1} c_{v_0} c_{v_1} + \mathbf{a}_{2,2} c_{u_1}^2 c_{v_0}^2\Big) t^2 \\
&\quad + \Big(\mathbf{a}_{0,1} c_{v_1} + 2\mathbf{a}_{1,2} c_{u_0} c_{v_0} c_{v_1} + \mathbf{a}_{1,2} c_{u_1} c_{v_0}^2 + \mathbf{a}_{1,0} c_{u_1} + 2\mathbf{a}_{2,2} c_{u_0}^2 c_{v_0} c_{v_1} \\
&\qquad + 2\mathbf{a}_{2,2} c_{u_0} c_{u_1} c_{v_0}^2 + 2\mathbf{a}_{0,2} c_{v_0} c_{v_1} + \mathbf{a}_{2,1} c_{u_0}^2 c_{v_1} + 2\mathbf{a}_{2,1} c_{u_0} c_{u_1} c_{v_0} \\
&\qquad + \mathbf{a}_{1,1} c_{u_0} c_{v_1} + \mathbf{a}_{1,1} c_{u_1} c_{v_0} + 2\mathbf{a}_{2,0} c_{u_0} c_{u_1}\Big) t \\
&\quad + \Big(\mathbf{a}_{0,0} + \mathbf{a}_{0,1} c_{v_0} + \mathbf{a}_{2,2} c_{u_0}^2 c_{v_0}^2 + \mathbf{a}_{2,1} c_{u_0}^2 c_{v_0} + \mathbf{a}_{0,2} c_{v_0}^2 + \mathbf{a}_{1,0} c_{u_0} \\
&\quad + \mathbf{a}_{2,0} c_{u_0}^2 + \mathbf{a}_{1,1} c_{u_0} c_{v_0} + \mathbf{a}_{1,2} c_{u_0} c_{v_0}^2\Big) \\
&= \sum_{e=0}^{4} \tilde{\mathbf{a}}_e t^e
\end{aligned}
$$

Following the same procedure, we can see that a degree p curve in parameter space gets mapped into a degree $p(k+m+n)$ curve by a trivariate tensor product volume of degree (k, m, n).

The monomial form algorithm allows us to calculate the Bézier points of the composed function, but the conversion from the geometrically intuitive Bernstein form to the monomial form and back is not only inconvenient but also limits the geometric insight we can achieve.

To obtain more insight into the geometric details during the composition process, we now develop a composition algorithm that uses only the Bernstein representation.

3.2.2 Blossoming Form Composition Algorithm

Let $\mathbf{f}(u,v) : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^d$ be a degree (m,n) tensor product Bézier patch in Bernstein form

$$\begin{aligned}\mathbf{f}(u,v) &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j} B_i^m(u) B_j^n(v) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \Delta(0^{<m-i>}, 1^{<i>}; 0^{<n-j>}, 1^{<j>}) B_i^m(u) B_j^n(v)\end{aligned}$$

and $\mathbf{g}(t) : \mathcal{R} \rightarrow \mathcal{R}^2$ be a degree p Bézier curve

$$\mathbf{g}(t) = \sum_{k=0}^{p} \mathbf{c}_k B_k^p(t),$$

with its projections onto the u, v axes

$$\begin{aligned}g_u(t) &= \sum_{k=0}^{p} c_{u_k} B_k^p(t) \\ g_v(t) &= \sum_{k=0}^{p} c_{v_k} B_k^p(t).\end{aligned}$$

The composed function $\tilde{\mathbf{g}}(t)$ then becomes

$$\begin{aligned}\tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t)) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j}\, B_i^m(g_u(t))\, B_j^n(g_v(t)) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j}\, B_i^m(\sum_{k=0}^{p} c_{u_k} B_k^p(t))\, B_j^n(\sum_{k=0}^{p} c_{v_k} B_k^p(t)) \qquad (3.4)\end{aligned}$$

Our first objective is to eliminate the two summations over k. Let us recall that Bézier patches can be thought of as curves of curves and rearrange equation (3.4) accordingly:

$$\tilde{\mathbf{g}}(t) = \sum_{i=0}^{m} B_i^m(\sum_{k=0}^{p} c_{u_k} B_k^p(t)) \sum_{j=0}^{n} B_j^n(\sum_{k=0}^{p} c_{v_k} B_k^p(t))\, \mathbf{b}_{i,j}.$$

In Section 2.2.5 we showed how to determine the control points of a

composed function $\tilde{\mathbf{g}}(u) = \mathbf{f}(g(u))$. We can apply this twice and obtain:

$$\begin{aligned}\tilde{\mathbf{g}}(t) &= \sum_{i=0}^{m} B_i^m(\sum_{k=0}^{p} c_{u_k} B_k^p(t)) \sum_{f=0}^{np} B_f^{np}(t)\,\tilde{\mathbf{b}}_{i,f} \\ &= \sum_{f=0}^{np} B_f^{np}(t) \sum_{i=0}^{m} B_i^m(\sum_{k=0}^{p} c_{u_k} B_k^p(t))\,\tilde{\mathbf{b}}_{i,f} \\ &= \sum_{f=0}^{np} B_f^{np}(t) \sum_{e=0}^{mp} B_e^{mp}(t)\,\tilde{\tilde{\mathbf{b}}}_{e,f} \qquad (3.5)\end{aligned}$$

We can see that the composed Bézier patch is of degree (np, mp). How can we determine the new Bézier points $\tilde{\tilde{\mathbf{b}}}_{e,f}$?

If we adapt Section 2.2.5, Equation (2.14), we can generate the Bézier points $\tilde{\mathbf{b}}_{i,f}$ of the composed function as

$$\begin{aligned}\tilde{\mathbf{b}}_{i,f} &= \tilde{\Delta}(0^{<m-i>}, 1^{<i>}; 0^{<np-f>}, 1^{<f>}) \\ &= \sum_{\substack{j_1,\ldots,j_n \in \{0,\ldots,p\} \\ j_1+\cdots+j_n = f}} C_f(j_1,\ldots,j_n)\,\Delta(0^{<m-i>}, 1^{<i>}; c_{v_{j_1}},\ldots,c_{v_{j_n}}) \qquad (3.6)\end{aligned}$$

with

$$C_f(j_1,\ldots,j_n) = \frac{\binom{p}{j_1}\binom{p}{j_2}\cdots\binom{p}{j_n}}{\binom{np}{f}}.$$

The Bézier points $\tilde{\tilde{\mathbf{b}}}_{e,f}$ can then be found as

$$\begin{aligned}\tilde{\tilde{\mathbf{b}}}_{e,f} &= \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,p\} \\ i_1+\cdots+i_m = e}} C_e(i_1,\ldots,i_m)\,\tilde{\Delta}(c_{u_{i_1}},\ldots,c_{u_{i_m}}; 0^{<np-f>}, 1^{<f>}) \\ &= \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,p\} \\ i_1+\cdots+i_m = e}} \sum_{\substack{j_1,\ldots,j_n \in \{0,\ldots,p\} \\ j_1+\cdots+j_n = f}} C_e(i_1,\ldots,i_m)\,C_f(j_1,\ldots,j_n) \\ &\quad \Delta(c_{u_{i_1}},\ldots,c_{u_{i_m}}; c_{v_{j_1}},\ldots,c_{v_{j_n}})\end{aligned}$$

with

$$C_e(i_1,\ldots,i_m) = \frac{\binom{p}{i_1}\binom{p}{i_2}\cdots\binom{p}{i_m}}{\binom{mp}{e}}.$$

We can now rewrite Equation (3.5) as

$$\begin{aligned}\tilde{\mathbf{g}}(t) &= \sum_{e=0}^{mp}\sum_{f=0}^{np} B_e^{mp}(t)\, B_f^{np}(t)\, \tilde{\tilde{\mathbf{b}}}_{e,f} \\ &= \sum_{e=0}^{mp}\sum_{f=0}^{np} \frac{\binom{pm}{e}\binom{pn}{f}}{\binom{p(m+n)}{e+f}} B_{e+f}^{p(m+n)}(t)\, \tilde{\tilde{\mathbf{b}}}_{e,f} \\ &= \sum_{g=0}^{p(m+n)} \tilde{\mathbf{c}}_g B_g^{p(m+n)}(t)\end{aligned}$$

with

$$\tilde{\mathbf{c}}_g = \sum_{e+f=g} \frac{\binom{pm}{e}\binom{pn}{f}}{\binom{p(m+n)}{g}}\, \tilde{\tilde{\mathbf{b}}}_{e,f}$$

If we define a new combinatorial constant

$$\mathcal{C}_g(e,f) = \frac{\binom{pm}{e}\binom{pn}{f}}{\binom{p(m+n)}{g}},$$

we can finally express the new reparametrised curve in Bézier form:

$$\tilde{\mathbf{g}}(t) = \sum_{g=0}^{p(m+n)} \tilde{\mathbf{c}}_g B_g^{p(m+n)}(t), \tag{3.7}$$

with

$$\begin{aligned}\tilde{\mathbf{c}}_g = \sum_{e+f=g} & \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,p\} \\ i_1+\cdots+i_m=e}} \sum_{\substack{j_1,\ldots,j_n \in \{0,\ldots,p\} \\ j_1+\cdots+j_n=f}} \\ & \mathcal{C}_g(e,f)\, \mathcal{C}_e(i_1,\ldots,i_m)\, \mathcal{C}_f(j_1,\ldots,j_n) \\ & \Delta(c_{u_{i_1}},\ldots,c_{u_{im}}; c_{v_{j_1}},\ldots,c_{v_{j_n}})\end{aligned}$$

Although the resulting formulae seem quite complex, we will now show that the blossoming form algorithm provides us with considerably more geometric insight than the monomial form algorithm.

Example 6 *We will use the same composition problem as for the monomial form algorithm (Example 5), so we are looking for the composed function* $\tilde{\mathbf{g}}(t) = \mathbf{f}(g_u(t), g_v(t))$ *of the biquadratic Bézier patch*

$$\mathbf{f}(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2} \mathbf{b}_{i,j} B_i^2(u) B_j^2(v)$$

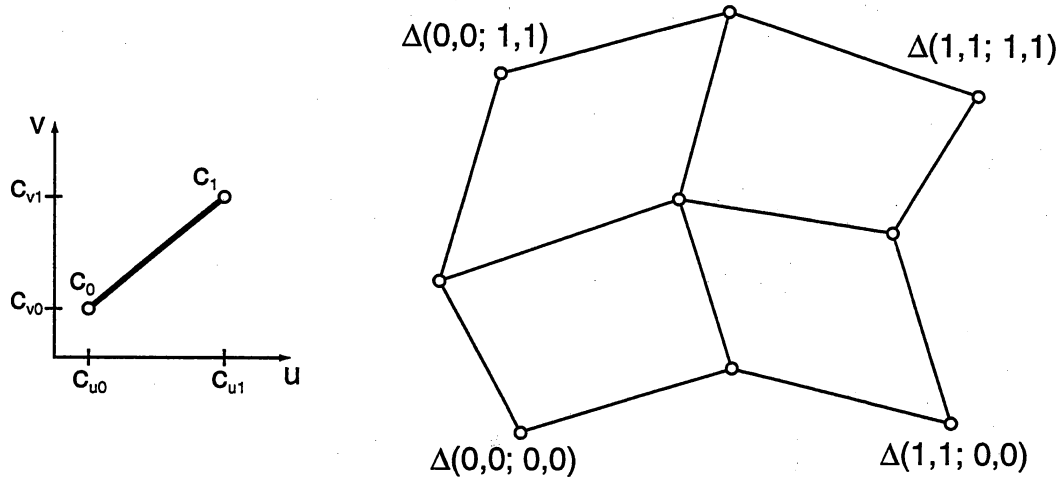


Figure 3.2: Linear Bézier curve and biquadratic Bézier patch

and a linear Bézier curve

$$\mathbf{g}(t) = \sum_{k=0}^{1} \mathbf{c}_k B_k^1(t),$$

with its projections onto the u, v axes

$$\begin{aligned} g_u(t) &= \sum_{k=0}^{1} c_{u_k} B_k^1(t) \\ g_v(t) &= \sum_{k=0}^{1} c_{v_k} B_k^1(t). \end{aligned}$$

Both the patch and the curve are shown in Figure 3.2. The composed function is

$$\begin{aligned} \tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t)) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j}\, B_i^2(\sum_{k=0}^{1} c_{u_k} B_k^1(t))\, B_j^2(\sum_{k=0}^{1} c_{v_k} B_k^1(t)) \\ &= \sum_{g=0}^{2} \tilde{\mathbf{c}}_g B_g^2(t) \end{aligned}$$

with

$$\begin{aligned} \tilde{\mathbf{c}}_g = \sum_{e+f=g} \sum_{\substack{i_1,i_2 \in \{0,1\} \\ i_1+i_2=e}} \sum_{\substack{j_1,j_2 \in \{0,1\} \\ j_1+j_2=f}} & \mathcal{C}_g(e,f)\, \mathcal{C}_e(i_1,i_2)\, \mathcal{C}_f(j_1,j_2) \\ & \Delta(c_{u_{i_1}}, c_{u_{i_2}}; c_{v_{j_1}}, c_{v_{j_2}}). \end{aligned}$$

Our first step consists of constructing all blossoms $\Delta(c_{u_{i_1}}, c_{u_{i_2}}; c_{v_{j_1}}, c_{v_{j_2}})$. *By thinking of a tensor product patch as curves of curves, we can do this in a two step process. Using repeated linear interpolation, we first construct the intermediate points* $\Delta(0^{2-i}, 1^i; c_{v_{j_1}}, c_{v_{j_2}})$ *(Figure 3.3).*

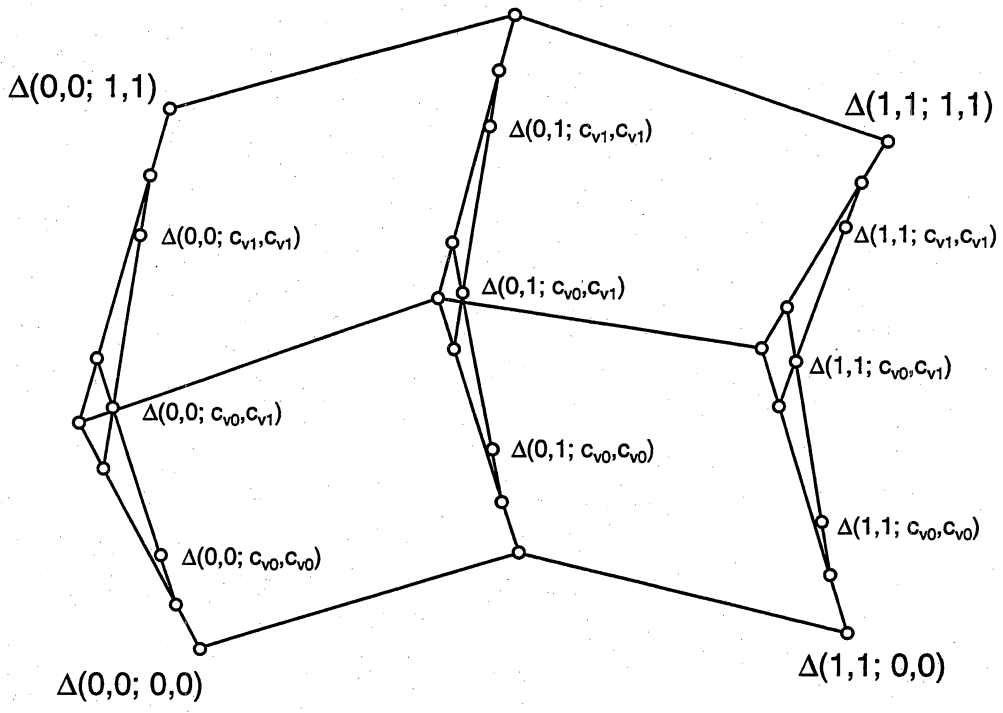


Figure 3.3: Reparametrisation of a biquadratic Bézier patch: intermediate Bézier points

In a second step, we use repeated linear interpolation between these intermediate points to construct the points $\Delta(c_{u_{i_1}}, c_{u_{i_2}}; c_{v_{j_1}}, c_{v_{j_2}})$. *This process is shown in Figure 3.4.*

Convex combinations between these points yield the Bézier points $\tilde{\mathbf{c}}_g$ *of the composed curve* $\tilde{\mathbf{g}}(t)$. *The combinatorial constants* $C_e(i_1, i_2)$, $C_f(j_1, j_2)$

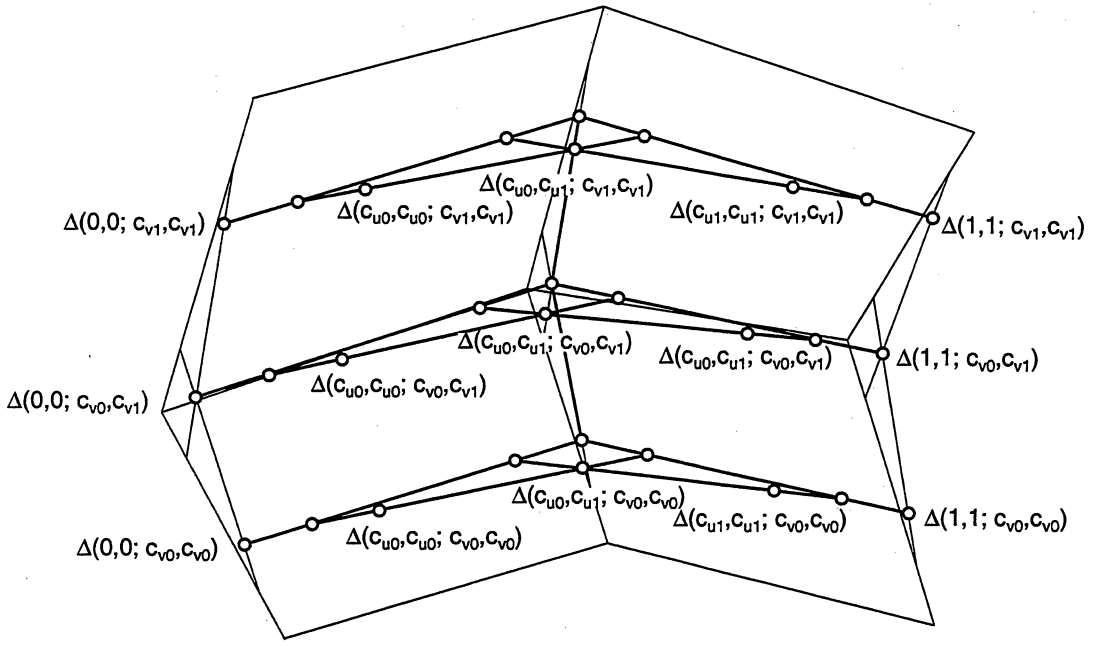


Figure 3.4: Reparametrisation of a biquadratic Bézier patch: intermediate Bézier points

and $\mathcal{C}_g(e,f)$ *are found to be*

$$\begin{aligned}
\mathcal{C}_e(0,0) &= \mathcal{C}_f(0,0) = 1 \\
\mathcal{C}_e(0,1) &= \mathcal{C}_f(0,1) = \frac{1}{2} \\
\mathcal{C}_e(1,1) &= \mathcal{C}_f(1,1) = 1 \\
\mathcal{C}_g(0,0) &= 1 \\
\mathcal{C}_g(0,1) &= \mathcal{C}_g(1,0) = \frac{1}{2} \\
\mathcal{C}_g(1,1) &= \frac{2}{3} \\
\mathcal{C}_g(0,2) &= \mathcal{C}_g(2,0) = \frac{1}{6} \\
\mathcal{C}_g(1,2) &= \mathcal{C}_g(2,1) = \frac{1}{2} \\
\mathcal{C}_g(2,2) &= 1
\end{aligned}$$

Thus the new Bézier points $\tilde{\mathbf{c}}_g$ *of the composed curve are*

$$
\begin{aligned}
\tilde{\mathbf{c}}_0 &= \Delta(c_{u_0}, c_{u_0}; c_{v_0}, c_{v_0}) \\
\tilde{\mathbf{c}}_1 &= \frac{1}{2}\Delta(c_{u_0}, c_{u_1}; c_{v_0}, c_{v_0}) + \frac{1}{2}\Delta(c_{u_0}, c_{u_0}; c_{v_0}, c_{v_1}) \\
\tilde{\mathbf{c}}_2 &= \frac{1}{6}\Delta(c_{u_1}, c_{u_1}; c_{v_0}, c_{v_0}) + \frac{2}{3}\Delta(c_{u_0}, c_{u_1}; c_{v_0}, c_{v_1}) + \frac{1}{6}\Delta(c_{u_0}, c_{u_0}; c_{v_1}, c_{v_1}) \\
\tilde{\mathbf{c}}_3 &= \frac{1}{2}\Delta(c_{u_0}, c_{u_1}; c_{v_1}, c_{v_1}) + \frac{1}{2}\Delta(c_{u_1}, c_{u_1}; c_{v_0}, c_{v_1}) \\
\tilde{\mathbf{c}}_4 &= \Delta(c_{u_1}, c_{u_1}; c_{v_1}, c_{v_1})
\end{aligned}
$$

The composition process and the new Bézier points of the composed curve are shown in Figure 3.5. Grey lines indicate convex combinations of points that yield the Bézier points of the composed curve.

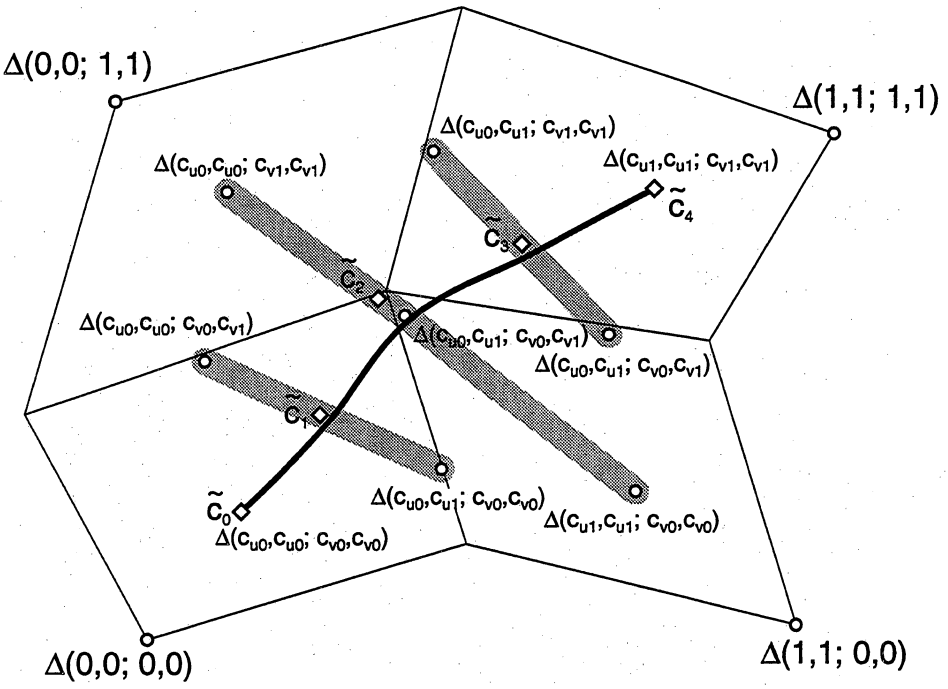


Figure 3.5: Reparametrisation of a biquadratic Bézier patch: final Bézier points

3.2.3 Extension to Higher Dimensions

Both the monomial form algorithm and the blossoming form algorithm can be easily extended to higher dimensions.

We first present the monomial form formula for the mapping of a Bézier curve into a Bézier volume. The derivation is strictly analogous to the two-dimensional case.

Let $\mathbf{f}(u,v,w) : \mathcal{R} \times \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^d$ be a degree (m,n,p) tensor product Bézier patch in monomial form

$$\mathbf{f}(u,v,w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{a}_{i,j,k} u^i v^j w^k$$

and $\mathbf{g}(t) : \mathcal{R} \rightarrow \mathcal{R}^3$ be a degree q Bézier curve

$$\mathbf{g}(t) = \sum_{e=0}^{q} \mathbf{c}_e t^e$$

with its projections onto the u, v, w axes

$$\begin{aligned} g_u(t) &= \sum_{e=0}^{q} c_{u_e} t^e \\ g_v(t) &= \sum_{e=0}^{q} c_{v_e} t^e \\ g_w(t) &= \sum_{e=0}^{q} c_{w_e} t^e. \end{aligned}$$

The composed function $\tilde{\mathbf{g}}(t)$ then becomes

$$\begin{aligned} \tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t), g_w(t)) \\ &= \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{a}_{i,j,k} \left[g_u(t)\right]^i \left[g_v(t)\right]^j \left[g_w(t)\right]^k \\ &= \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{a}_{i,j,k} \left[\sum_{e=0}^{q} c_{u_e} t^e\right]^i \left[\sum_{e=0}^{q} c_{v_e} t^e\right]^j \left[\sum_{e=0}^{q} c_{w_e} t^e\right]^k \end{aligned}$$

Expanding the expression and collecting the coefficients for each power of t as $\tilde{\mathbf{b}}_f$ finally yields

$$\tilde{\mathbf{g}}(t) = \sum_{f=0}^{q(m+n+p)} \tilde{\mathbf{a}}_f t^f,$$

which is the power form representation of a degree $(q(m+n+p))$ Bézier curve.

The derivation of the three-dimensional blossoming algorithm is exactly along the lines of the two-dimensional algorithm. We start off with a degree (m, n, p) tensor product Bézier volume in Bernstein form

$$\mathbf{f}(u, v, w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{b}_{i,j,k} B_i^m(u) B_j^n(v) B_k^p(w)$$

and a degree q Bézier curve

$$\mathbf{g}(t) = \sum_{e=0}^{q} \mathbf{c}_e B_e^q(t)$$

with its three projections $g_u(t)$, $g_v(t)$ and $g_w(t)$.

We can find the new Bézier points $\tilde{\tilde{\tilde{\mathbf{b}}}}_{f,g,h}$ as

$$\begin{aligned} \tilde{\tilde{\tilde{\mathbf{b}}}}_{f,g,h} \quad = \quad & \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,q\} \\ i_1+\cdots+i_m=f}} \sum_{\substack{j_1,\ldots,j_n \in \{0,\ldots,q\} \\ j_1+\cdots+j_n=g}} \sum_{\substack{k_1,\ldots,k_p \in \{0,\ldots,q\} \\ k_1+\cdots+k_p=h}} \\ & \mathcal{C}_f(i_1,\ldots,i_m)\, \mathcal{C}_g(j_1,\ldots,j_n)\, \mathcal{C}_h(k_1,\ldots,k_p) \\ & \Delta(c_{u_{i_1}},\ldots,c_{u_{i_m}}; c_{v_{j_1}},\ldots,c_{v_{j_n}}; c_{w_{k_1}},\ldots,c_{w_{k_p}}) \end{aligned}$$

with

$$\mathcal{C}_f(i_1,\ldots,i_m) = \frac{\binom{q}{i_1}\binom{q}{i_2}\cdots\binom{q}{i_m}}{\binom{mq}{f}}$$

and

$$\mathcal{C}_g(j_1,\ldots,j_n) = \frac{\binom{q}{j_1}\binom{q}{j_2}\cdots\binom{q}{j_n}}{\binom{nq}{g}}$$

and

$$\mathcal{C}_h(k_1,\ldots,k_p) = \frac{\binom{q}{k_1}\binom{q}{k_2}\cdots\binom{q}{k_p}}{\binom{pq}{h}}.$$

Now we can write the composed curve as

$$\begin{aligned} \tilde{\mathbf{g}}(t) \quad &= \quad \sum_{f=0}^{mq} \sum_{g=0}^{nq} \sum_{h=0}^{pq} B_f^{mq}(t)\, B_g^{nq}(t)\, B_h^{pq}(t)\, \tilde{\tilde{\tilde{\mathbf{b}}}}_{f,g,h} \\ &= \quad \sum_{f=0}^{mq} \sum_{g=0}^{nq} \sum_{h=0}^{pq} \frac{\binom{mq}{f}\binom{nq}{g}\binom{pq}{h}}{\binom{q(m+n+p)}{f+g+h}} B_{f+g+h}^{q(m+n+p)}(t)\, \tilde{\tilde{\tilde{\mathbf{b}}}}_{f,g,h} \\ &= \quad \sum_{d=0}^{q(m+n+p)} \tilde{\mathbf{c}}_d B_d^{q(m+n+p)}(t) \end{aligned} \tag{3.8}$$

with

$$\tilde{\mathbf{c}}_d = \sum_{f+g+h=d} \frac{\binom{mq}{f}\binom{nq}{g}\binom{pq}{h}}{\binom{q(m+n+p)}{d}} \tilde{\tilde{\tilde{\mathbf{b}}}}_{f,g,h}$$

Introducing a new combinatorial constant

$$C_d(f,g,h) = \frac{\binom{mq}{f}\binom{nq}{g}\binom{pq}{h}}{\binom{q(m+n+p)}{d}}$$

we can see that with Equation (3.8) we have finally obtained the Bernstein-Bézier representation of the new, composed curve:

$$\tilde{\mathbf{g}}(t) = \sum_{d=0}^{q(m+n+p)} \tilde{\mathbf{c}}_d B_d^{q(m+n+p)}(t)$$

with

$$\begin{aligned}\tilde{\mathbf{c}}_d \;=\; & \sum_{f+g+h=d} \; \sum_{\substack{i_1,\ldots,i_m \in \{0,\ldots,q\} \\ i_1+\cdots+i_m=f}} \; \sum_{\substack{j_1,\ldots,j_n \in \{0,\ldots,q\} \\ j_1+\cdots+j_n=g}} \; \sum_{\substack{k_1,\ldots,j_p \in \{0,\ldots,q\} \\ k_1+\cdots+j_p=h}} \\ & C_d(f,g,h)\, C_f(i_1,\ldots,i_m)\, C_g(j_1,\ldots,j_n)\, C_h(k_1,\ldots,k_p) \\ & \Delta(c_{u_{i_1}},\ldots,c_{u_{i_m}}; c_{v_{j_1}},\ldots,c_{v_{j_n}}; c_{w_{k_1}},\ldots,c_{w_{k_p}})\end{aligned}$$

3.3 Composition of Bézier Patch and Bézier Triangle

Considering the fact that Bézier triangles represent a generalisation of Bézier curves, it is not surprising that the composition algorithm with Bézier triangles can be developed along the same lines as the composition with Bézier curves.

Let $\mathbf{f}(u,v) : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^d$ be a degree (m,n) tensor product Bézier patch in Bernstein form

$$\mathbf{f}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j} B_i^m(u) B_j^n(v)$$

and $\mathbf{g}(\mathbf{t}) : \mathcal{R}^2 \to \mathcal{R}^2$ be a degree p Bézier simplex

$$\mathbf{g}(\mathbf{t}) = \sum_{|\mathbf{k}|=p} \mathbf{c}_\mathbf{k} B_\mathbf{k}^p(\mathbf{t})$$

with its projections onto the u, v axes

$$\begin{aligned} g_u(\mathbf{t}) &= \sum_{|\mathbf{k}|=p} c_{u_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t}) \\ g_v(\mathbf{t}) &= \sum_{|\mathbf{k}|=p} c_{v_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t}). \end{aligned}$$

The composed function $\tilde{\mathbf{g}}(\mathbf{t})$ becomes

$$\begin{aligned}
\tilde{\mathbf{g}}(\mathbf{t}) &= \mathbf{f}(g_u(\mathbf{t}), g_v(\mathbf{t})) \\
&= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j}\, B_i^m(g_u(\mathbf{t}))\, B_j^n(g_v(\mathbf{t})) \\
&= \sum_{i=0}^{m}\sum_{j=0}^{n} \mathbf{b}_{i,j}\, B_i^m(\sum_{|\mathbf{k}|=p} c_{u_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t}))\, B_j^n(\sum_{|\mathbf{k}|=p} c_{v_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t})) \qquad (3.9)
\end{aligned}$$

Again, we can rearrange Equation (3.9) and reparametrise twice:

$$\begin{aligned}
\tilde{\mathbf{g}}(\mathbf{t}) &= \sum_{i=0}^{m} B_i^m(\sum_{|\mathbf{k}|=p} c_{u_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t})) \sum_{j=0}^{n} B_j^n(\sum_{|\mathbf{k}|=p} c_{v_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t}))\, \mathbf{b}_{i,j} \qquad (3.10) \\
&= \sum_{i=0}^{m} B_i^m(\sum_{k=0}^{p} c_{u_\mathbf{k}} B_\mathbf{k}^p(\mathbf{t})) \sum_{|\mathbf{f}|=np} B_\mathbf{f}^{np}(\mathbf{t})\, \tilde{\mathbf{b}}_{i,\mathbf{f}} \\
&= \sum_{|\mathbf{f}|=np} B_\mathbf{f}^{np}(\mathbf{t}) \sum_{i=0}^{m} B_i^m(\sum_{k=0}^{p} c_{u_k} B_k^p(\mathbf{t}))\, \tilde{\mathbf{b}}_{i,\mathbf{f}} \\
&= \sum_{|\mathbf{f}|=np} B_\mathbf{f}^{np}(\mathbf{t}) \sum_{|\mathbf{e}|=mp} B_\mathbf{e}^{mp}(\mathbf{t})\, \tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}} \qquad (3.11)
\end{aligned}$$

We can generate the intermediate Bézier points $\tilde{\mathbf{b}}_{i,\mathbf{f}}$ of the composed function as

$$\begin{aligned}
\tilde{\mathbf{b}}_{i,\mathbf{f}} &= \tilde{\Delta}(0^{<m-i>}, 1^{<i>}; \mathbf{e1}^{<f_0>}, \mathbf{e2}^{<f_1>}, \mathbf{e3}^{<f_2>}) \\
&= \sum_{\substack{|\mathbf{j}_1| = \cdots = |\mathbf{j}_n| = p \\ \mathbf{j}_1 + \cdots + \mathbf{j}_n = \mathbf{f}}} \mathcal{C}_\mathbf{f}(\mathbf{j}_1, \ldots, \mathbf{j}_n) \\
&\quad \Delta(0^{<m-i>}, 1^{<m>}; c_{v_{\mathbf{j}_1}}, \ldots, c_{v_{\mathbf{j}_n}}) \qquad (3.12)
\end{aligned}$$

with

$$\mathcal{C}_\mathbf{f}(\mathbf{j}_1, \ldots, \mathbf{j}_n) = \frac{\binom{p}{\mathbf{j}_1}\binom{p}{\mathbf{j}_2}\cdots\binom{p}{\mathbf{j}_n}}{\binom{np}{\mathbf{f}}}.$$

The Bézier points $\tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}}$ can be found as

$$\begin{aligned}
\tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}} &= \sum_{\substack{|\mathbf{i}_1| = \cdots = |\mathbf{i}_m| = p \\ \mathbf{i}_1 + \cdots + \mathbf{i}_m = \mathbf{e}}} \mathcal{C}_\mathbf{e}(\mathbf{i}_1, \ldots, \mathbf{i}_m) \\
&\quad \tilde{\Delta}(c_{u_{\mathbf{i}_1}}, \ldots, c_{u_{\mathbf{i}_m}}; \mathbf{e1}^{<f_0>}, \mathbf{e2}^{<f_1>}, \mathbf{e3}^{<f_2>}) \\
&= \sum_{\substack{|\mathbf{i}_1| = \cdots = |\mathbf{i}_m| = p \\ \mathbf{i}_1 + \cdots + \mathbf{i}_m = \mathbf{e}}} \sum_{\substack{|\mathbf{j}_1| = \cdots = |\mathbf{j}_n| = p \\ \mathbf{j}_1 + \cdots + \mathbf{j}_n = \mathbf{f}}} \mathcal{C}_\mathbf{e}(\mathbf{i}_1, \ldots, \mathbf{i}_m)\, \mathcal{C}_\mathbf{f}(\mathbf{j}_1, \ldots, \mathbf{j}_n) \\
&\quad \Delta(c_{u_{\mathbf{i}_1}}, \ldots, c_{u_{\mathbf{i}_m}}; c_{v_{\mathbf{j}_1}}, \ldots, c_{v_{\mathbf{j}_n}})
\end{aligned}$$

with

$$\mathcal{C}_{\mathbf{e}}(\mathbf{i}_1,\ldots,\mathbf{i}_m) = \frac{\binom{p}{\mathbf{i}_1}\binom{p}{\mathbf{i}_2}\cdots\binom{p}{\mathbf{i}_m}}{\binom{mp}{\mathbf{e}}}.$$

we can now rewrite Equation (3.11) as

$$\begin{aligned}\tilde{\mathbf{g}}(\mathbf{t}) &= \sum_{|\mathbf{e}|=mp}\sum_{|\mathbf{f}|=np}\tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}}\,B_{\mathbf{e}}^{mp}(\mathbf{t})\,B_{\mathbf{f}}^{np}(\mathbf{t})\\ &= \sum_{|\mathbf{e}|=mp}\sum_{|\mathbf{f}|=np}\frac{\binom{pm}{\mathbf{e}}\binom{pn}{\mathbf{f}}}{\binom{p(m+n)}{\mathbf{e}+\mathbf{f}}}\,B_{\mathbf{e}+\mathbf{f}}^{p(m+n)}(\mathbf{t})\,\tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}}\\ &= \sum_{|\mathbf{g}|=p(m+n)}\tilde{\mathbf{c}}_{\mathbf{g}}\,B_{\mathbf{g}}^{p(m+n)}(\mathbf{t}) \qquad (3.13)\end{aligned}$$

with

$$\tilde{\mathbf{c}}_{\mathbf{g}} = \sum_{\mathbf{e}+\mathbf{f}=\mathbf{g}}\frac{\binom{pm}{\mathbf{e}}\binom{pn}{\mathbf{f}}}{\binom{p(m+n)}{\mathbf{g}}}\,\tilde{\tilde{\mathbf{b}}}_{\mathbf{e},\mathbf{f}}$$

Equation (3.13) is the Bernstein-Bézier representation of the composed Bézier triangle and $\tilde{\mathbf{c}}_{\mathbf{g}}$ are its new control points.

Defining new combinatorial constants

$$\mathcal{C}_{\mathbf{g}}(\mathbf{e},\mathbf{f}) = \frac{\binom{pm}{\mathbf{e}}\binom{pn}{\mathbf{f}}}{\binom{p(m+n)}{\mathbf{g}}},$$

we can express the new control points as

$$\begin{aligned}\tilde{\mathbf{c}}_{\mathbf{g}} = \sum_{\mathbf{e}+\mathbf{f}=\mathbf{g}} &\sum_{\substack{|\mathbf{i}_1|=\cdots=|\mathbf{i}_m|=p\\ \mathbf{i}_1+\cdots+\mathbf{i}_m=\mathbf{e}}}\ \sum_{\substack{|\mathbf{j}_1|=\cdots=|\mathbf{j}_n|=p\\ \mathbf{j}_1+\cdots+\mathbf{j}_n=\mathbf{f}}}\\ &\mathcal{C}_{\mathbf{g}}(\mathbf{e},\mathbf{f})\,\mathcal{C}_{\mathbf{e}}(\mathbf{i}_1,\ldots,\mathbf{i}_m)\,\mathcal{C}_{\mathbf{f}}(\mathbf{j}_1,\ldots,\mathbf{j}_n)\\ &\Delta(c_{u_{\mathbf{i}_1}},\ldots,c_{u_{\mathbf{i}_m}};c_{v_{\mathbf{j}_1}},\ldots,c_{v_{\mathbf{j}_n}}).\end{aligned}$$

Example 7 *We will now give a geometric interpretation of the algorithm by giving an example of the composition of a tensor product Bézier patch and a Bézier triangle. For the benefit of clarity, we will treat the composition of the two simplest surfaces, i.e. bilinear patch and linear triangle.*

We are looking for the composed function $\tilde{\mathbf{g}}(\mathbf{t}) = \mathbf{f}(g_u(\mathbf{t}), g_v(\mathbf{t}))$ *of the bilinear Bézier patch*

$$\mathbf{f}(u,v) = \sum_{i=0}^{2}\sum_{j=0}^{2}\mathbf{b}_{i,j}B_i^2(u)B_j^2(v)$$

and a linear Bézier triangle

$$\mathbf{g(t)} = \sum_{|\mathbf{k}|=1} \mathbf{c_k} B^1_{\mathbf{k}}(\mathbf{t}),$$

with its projections onto the u, v axes

$$\begin{aligned} g_u(\mathbf{t}) &= \sum_{|\mathbf{k}|=1} c_{u_k} B^1_{\mathbf{k}}(t) \\ g_v(\mathbf{t}) &= \sum_{|\mathbf{k}|=1} c_{v_k} B^1_{\mathbf{k}}(t). \end{aligned}$$

Both the patch and the curve are shown in Figure 3.6.

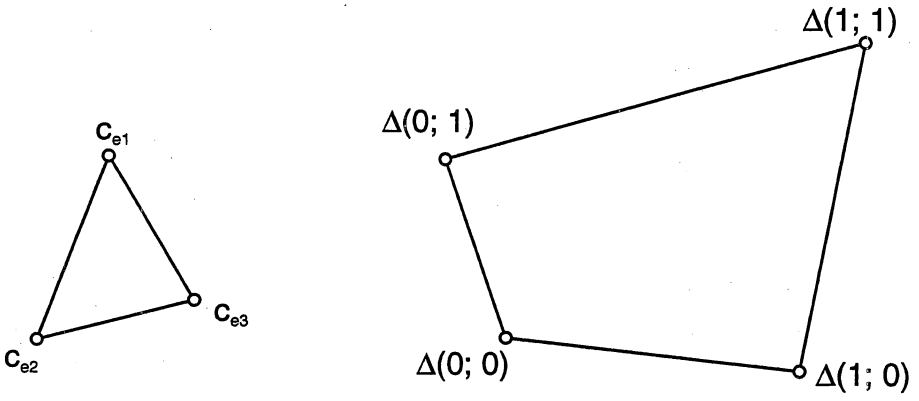


Figure 3.6: Mapping of a linear Bézier triangle onto a bilinear Bézier patch

By Equation (3.13), the composed function is

$$\begin{aligned} \tilde{\mathbf{g}}(t) &= \mathbf{f}(g_u(t), g_v(t)) \\ &= \sum_{|\mathbf{g}|=2} \tilde{\mathbf{c}}_{\mathbf{g}} B^2_{\mathbf{g}}(\mathbf{t}) \end{aligned}$$

with

$$\tilde{\mathbf{c}}_{\mathbf{g}} = \sum_{\mathbf{e}+\mathbf{f}=\mathbf{g}} \sum_{\substack{|\mathbf{i}_1|=1 \\ \mathbf{i}_1=\mathbf{e}}} \sum_{\substack{|\mathbf{j}_1|=1 \\ \mathbf{j}_1=\mathbf{f}}} \mathcal{C}_{\mathbf{g}}(\mathbf{e},\mathbf{f})\, \mathcal{C}_{\mathbf{e}}(\mathbf{i}_1)\, \mathcal{C}_{\mathbf{f}}(\mathbf{j}_1) \Delta(c_{u_{\mathbf{i}_1}}; c_{v_{\mathbf{j}_1}}).$$

Our first step is to construct all points with the blossoms

$$\Delta(c_{u_{\mathbf{i}_1}}; c_{v_{\mathbf{j}_1}}).$$

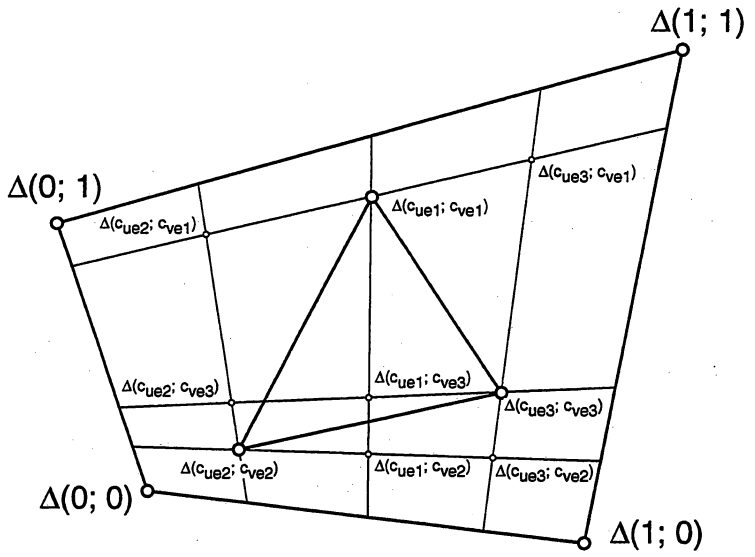


Figure 3.7: Composition algorithm: intermediate points

We obtain 9 points which are shown in Figure 3.7.

The control points $\tilde{\mathbf{c}}_{\mathbf{g}}$ *of the new, composed, Bézier triangle are now found as linear combinations of these blossoms. We obtain*

$$
\begin{aligned}
\tilde{\mathbf{c}}_{(200)} &= \tilde{\mathbf{c}}_{\mathbf{e1+e1}} = \Delta(c_{u_{\mathbf{e1}}}; c_{v_{\mathbf{e1}}}) \\
\tilde{\mathbf{c}}_{(020)} &= \tilde{\mathbf{c}}_{\mathbf{e2+e2}} = \Delta(c_{u_{\mathbf{e2}}}; c_{v_{\mathbf{e2}}}) \\
\tilde{\mathbf{c}}_{(002)} &= \tilde{\mathbf{c}}_{\mathbf{e3+e3}} = \Delta(c_{u_{\mathbf{e3}}}; c_{v_{\mathbf{e3}}}) \\
\tilde{\mathbf{c}}_{(110)} &= \tilde{\mathbf{c}}_{\mathbf{e1+e2}} = \frac{1}{2}\Delta(c_{u_{\mathbf{e1}}}; c_{v_{\mathbf{e2}}}) + \frac{1}{2}\Delta(c_{u_{\mathbf{e2}}}; c_{v_{\mathbf{e1}}}) \\
\tilde{\mathbf{c}}_{(011)} &= \tilde{\mathbf{c}}_{\mathbf{e2+e3}} = \frac{1}{2}\Delta(c_{u_{\mathbf{e2}}}; c_{v_{\mathbf{e3}}}) + \frac{1}{2}\Delta(c_{u_{\mathbf{e3}}}; c_{v_{\mathbf{e2}}}) \\
\tilde{\mathbf{c}}_{(101)} &= \tilde{\mathbf{c}}_{\mathbf{e1+e3}} = \frac{1}{2}\Delta(c_{u_{\mathbf{e1}}}; c_{v_{\mathbf{e3}}}) + \frac{1}{2}\Delta(c_{u_{\mathbf{e3}}}; c_{v_{\mathbf{e1}}})
\end{aligned}
$$

The composition process of the new Bézier triangle is shown in Figure 3.8. The grey lines indicate convex combination of points.

3.3.1 Extension to Higher Dimensions

The extension of the algorithm to the composition of a Bézier volume with a Bézier triangle is straightforward. For completeness, we give the expression for the composed triangle:

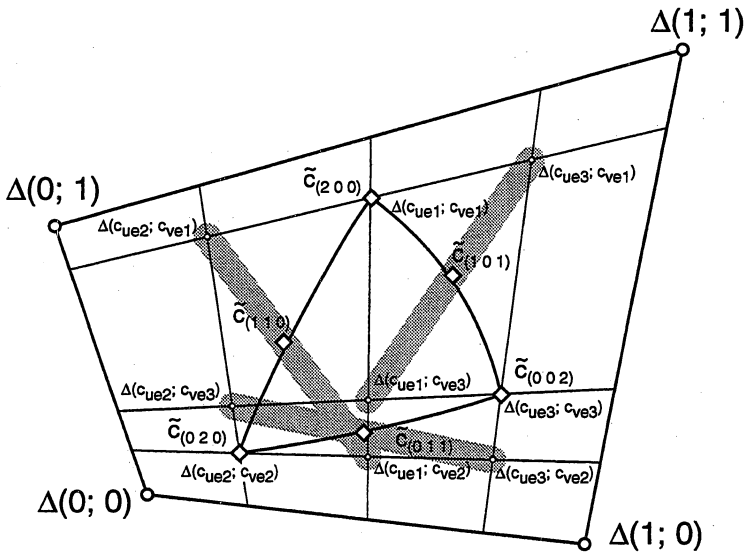


Figure 3.8: Composition algorithm: final points are obtained as convex combinations

$$\tilde{\mathbf{g}}(\mathbf{t}) = \sum_{|\mathbf{d}|=q(m+n+p)} \tilde{\mathbf{c}}_{\mathbf{d}} B_{\mathbf{d}}^{q(m+n+p)}(\mathbf{t})$$

with

$$\begin{aligned}\tilde{\mathbf{c}}_{\mathbf{d}} = \sum_{\mathbf{f}+\mathbf{g}+\mathbf{h}=\mathbf{d}} & \sum_{\substack{|\mathbf{i}_1|=\cdots=|\mathbf{i}_m|=q \\ \mathbf{i}_1+\cdots+\mathbf{i}_m=\mathbf{f}}} \sum_{\substack{|\mathbf{j}_1|=\ldots|\mathbf{j}_n|=q \\ \mathbf{j}_1+\cdots+\mathbf{j}_n=\mathbf{g}}} \sum_{\substack{|\mathbf{k}_1|+\cdots+|\mathbf{j}_p|=q \\ \mathbf{k}_1+\cdots+\mathbf{j}_p=\mathbf{h}}} \\ & \mathcal{C}_{\mathbf{d}}(\mathbf{f},\mathbf{g},\mathbf{h})\,\mathcal{C}_{\mathbf{f}}(\mathbf{i}_1,\ldots,\mathbf{i}_m)\,\mathcal{C}_{\mathbf{g}}(\mathbf{j}_1,\ldots,\mathbf{j}_n)\,\mathcal{C}_{\mathbf{h}}(\mathbf{k}_1,\ldots,\mathbf{k}_p) \\ & \Delta(c_{u_{\mathbf{i}_1}},\ldots,c_{u_{\mathbf{i}_m}};c_{v_{\mathbf{j}_1}},\ldots,c_{v_{\mathbf{j}_n}};c_{w_{\mathbf{k}_1}},\ldots,c_{w_{\mathbf{k}_p}})\end{aligned}$$

3.3.2 Composition of a Bézier Volume and a Bézier Patch

Let us finally look at the degree of a patch that results when we compose a Bézier volume and a Bézier patch. To obtain information on the degree of the resulting patch, we use the monomial form algorithm.

Let $\mathbf{f}(u,v,w) : \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}^d$ be a degree (m,n,p) tensor product Bézier volume in monomial form

$$\mathbf{f}(u,v,w) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p} \mathbf{a}_{i,j,k} u^i v^j w^k$$

and let $\mathbf{g}(t) : \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}^3$ be a degree (q, r) Bézier patch

$$\mathbf{g}(s,t) = \sum_{e=0}^{q} \sum_{f=0}^{r} \mathbf{c}_e \mathbf{c}_f \, s^e t^f$$

with projections onto the u, v, w axes $g_u(s,t)$, $g_v(s,t)$ and $g_w(s,t)$.

The composed function $\tilde{\mathbf{g}}(t)$ then becomes

$$\begin{aligned}
\tilde{\mathbf{g}}(s,t) &= \mathbf{f}(g_u(s,t), g_v(s,t), g_w(s,t)) \\
&= \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{a}_{i,j,k} \left[g_u(s,t)\right]^i \left[g_v(s,t)\right]^j \left[g_w(s,t)\right]^k \\
&= \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{a}_{i,j,k} \\
&\quad \left[\sum_{e=0}^{q} \sum_{f=0}^{r} c_{u_e} c_{u_f} s^e t^f\right]^i \left[\sum_{e=0}^{q} \sum_{f=0}^{r} c_{v_e} c_{v_f} s^e t^f\right]^j \left[\sum_{e=0}^{q} \sum_{f=0}^{r} c_{w_e} c_{w_f} s^e t^f\right]^k
\end{aligned}$$

Expanding the expression and collecting the coefficients for each power of s and t as $\tilde{\mathbf{b}}_{g,h}$ yields

$$\tilde{\mathbf{g}}(s,t) = \sum_{g=0}^{q(m+n+p)} \sum_{h=0}^{r(m+n+p)} \tilde{\mathbf{a}}_{g,h} s^g t^h, \tag{3.14}$$

which is the power form representation of a degree $[q(m+n+p), r(m+n+p)]$ Bézier patch.

3.4 Further Composition Problems

Let us briefly remark on some further possible extensions of our composition algorithms:

- deformation of B-spline curves
- deformation of tensor product B-spline surfaces
- deformation of curves by a B-spline solid
- deformation of surfaces by a B-spline solid
- rational Bézier curves and non-uniform rational B-splines (NURBS)

The deformation of B-spline curves can be achieved by considering them as a number of Bézier curves with a certain continuity between them. In Section 2.3, we have shown that the blossoming notation provides a convenient way of expressing the control points of the individual Bézier curves. The deformation of a B-spline curve is then reduced to the deformation of a number of Bézier curves. Let us suppose the B-spline is of degree q with single knot multiplicity k (C^{q-k} continuity) and the deformation solid is of degree (m, n, p). The composed Bézier curves will then be of degree $q(m+n+p)$. The composed Bézier curves will still join with C^{q-k} continuity, as the composition is not going to change the continuity. We can therefore join them into a B-spline with knot multiplicity $q(m+n+p)-(q-k)$.

The deformation of tensor product B-spline surfaces can be done using the same principle, if we consider them as a collection of Bézier patches with a certain degree of continuity between them. Again, we use the blossoming principle to find the individual Bézier patches, deform them, and rejoin the deformed patches.

The deformation with B-spline surfaces and volumes is more involved. Figure 3.9 (a) shows a curve in the parameter space of a B-spline patch, where t_{u0}, t_{u1}, t_{u2} and t_{v0}, t_{v1}, t_{v2} denote the knot lines. The composition can again be done by considering the B-spline patch as a number of Bézier patches. In order to determine which part of the curve falls into which Bézier patch, we have to determine the intersections of the curve with the knot line in which the Bézier patches join[2]. These intersection points constitute the new knots of our deformed B-spline curve. The deformed curve is determined by deforming each curve segment individually and re-joining them into a B-spline.

The problem is more complex for the deformation of Bézier patches. If the Bézier patch is aligned with the isoparametric lines of the B-spline patch (Figure 3.9 (b)), the new knots can be determined for each parameter separately. However, if the Bézier patch and the B-spline patch are not aligned, it is not clear how to determine the knot vector of the composed patch, as the intersection of the two parameters with the knot lines are no longer independent of each other.

Finally, let us consider the generalisation of our composition algorithms to rational curves (for an introduction to rational B-splines and Bézier curves, see [Pie91]). A rational Bézier curve in $\mathcal{R}^3$ can be viewed as a projection

[2]Note that a degree q curve can intersect a knot line up to q times.

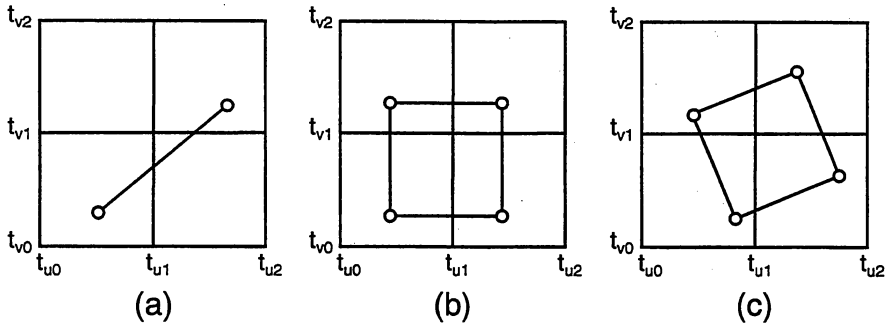


Figure 3.9: Deformation by B-spline patch: curve (a), aligned Bézier patch (b), general Bézier patch (c)

of a Bézier curve in $\mathcal{R}^4$ onto the hyperplane $w = 1$ (Farin [Far83]). Bézier patches and B-splines are extended the same way. To apply our composition algorithms to rational curves, we can therefore simply apply the composition algorithm in $\mathcal{R}^4$ space.

3.5 Summary

In this chapter, we have developed functional composition algorithms to find analytic expressions for curves and surfaces that have been embedded in tensor product Bézier surfaces and volumes. The functional composition problem is posed in terms of composing Bézier simplexes with tensor product Bézier hyperpatches. We chose Bézier curves to investigate the deformation of curves and Bézier triangles for the deformation of surfaces. The deformation basis can be defined in terms of Bézier simplexes (triangles or tetrahedra) or tensor product Bézier patches and volumes. We have used tensor product deformation bases for this discussion as they are most frequently used for Free Form Deformations.

Most of the composition problems were investigated using the blossoming principle. The extensive use of the blossoming notation has led to algorithms with geometrically meaningful interpretations.

We have concluded our chapter with a discussion of possible extensions of our composition algorithms to B-splines and rational Bézier simplexes and rational B-splines (NURBS).

In the next chapter, we shall investigate the relevance of these results to the Free Form Deformation method.

Chapter 4

Free Form Deformations

4.1 Introduction

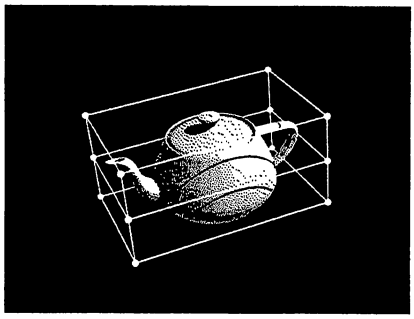
Functional composition algorithms are useful to understand and analyse the Free Form Deformation (FFD) principle. FFD is a versatile representation-independent technique for deforming geometric models in a free-form manner.

The original technique is due to Bézier [Béz78], although it gained popularity through the more graphics-oriented implementation by Sederberg and Parry [Par86].

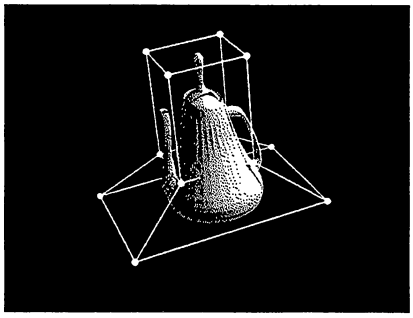
We begin this chapter with a general introduction to the Free Form Deformation algorithm. We then use the Sederberg-Parry implementation [SP86b], based on a trivariate tensor product deformation basis, to provide a step-by-step discussion of the algorithm and show how to evaluate deformed points, tangents and normals. Using the results on functional composition, derived in the previous chapter, we give expressions for edges and surfaces of deformed objects. The chapter closes with a review of extensions to the original Free Form Deformation Algorithm.

4.2 The FFD Algorithm

FFD can be thought of as embedding an object in a block of clear, pliable plastic. A deformation is applied to the plastic block and the embedded object is deformed together with the surrounding solid. Thus we have an indirect way of changing the shape of an object by deforming the surrounding solid.



(a)



(b)

Figure 4.1: Turning the teapot into a jug: Definition of the deformation solid (a), deformation of the solid results in deformation of the enclosed teapot (b)

Figure 4.1 shows the process of applying FFD to an object. First, a solid is constructed around the object to be deformed (a). The deformation solid is shown as a cube with bright edges. Then the deformation is applied by displacing the control points of the deformation solid, which indirectly changes the shape of the embedded object (b). The deformation solid in Figure 4.1 has twelve control points. Depending on the type of desired deformation, deformation solids with more control points can be defined.

Mathematically, the FFD deformation solid is defined as a lattice of control points of a trivariate parametric function $\mathbf{f}(u, v, w)$, which performs a mapping $\mathcal{R}^3 \rightarrow \mathcal{R}^3$. Assuming no singularities, each object point (x, y, z) in *world space* can be mapped to a point (u, v, w) in *parameter space*[1]. The

[1]Note that this is the parameter space of the deformation solid and not of the object. FFD does not require a parametrically defined object.

deformation is applied by displacing the control points, which leads to a new function $\tilde{\mathbf{f}}(u, v, w)$. Now each point (u, v, w) can be re-mapped using $\tilde{\mathbf{f}}$ to a point $(\tilde{x}, \tilde{y}, \tilde{z})$ in *deformed space*. FFD therefore consists of a two stage mapping, where each point in world space is first mapped to parameter space and then to deformed space:

$$(x, y, z) \xrightarrow{\mathbf{f}^{-1}} (u, v, w) \xrightarrow{\tilde{\mathbf{f}}} (\tilde{x}, \tilde{y}, \tilde{z}).$$

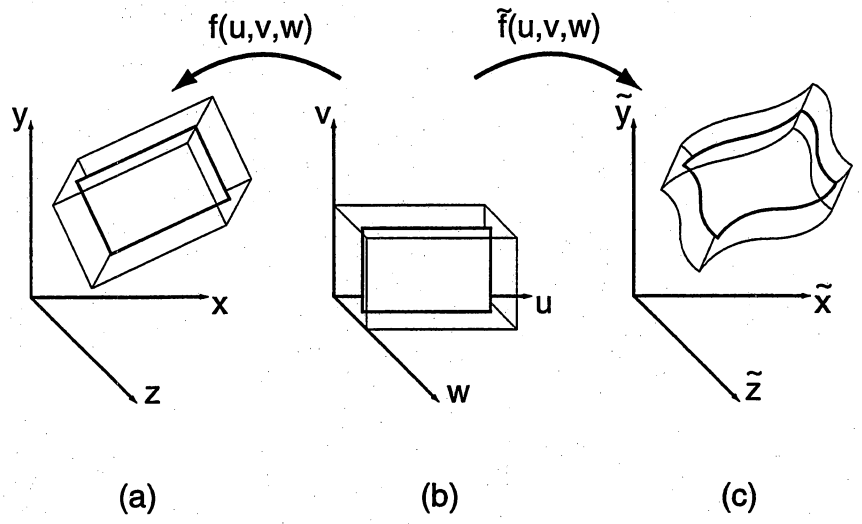


Figure 4.2: Surface in world space (x, y, z), parameter space (u, v, w) and deformed space $(\tilde{x}, \tilde{y}, \tilde{z})$

An implementation of FFD consists of four steps:

1. Defining a parametric solid that encloses the object or surface to be deformed. The solid consists of a lattice of control points and a corresponding set of parametric basis functions and performs a mapping $\mathcal{R}^3 \rightarrow \mathcal{R}^3$. Assuming no singularities, each point in world space (x, y, z) can be expressed as a set of local parametric coordinates (u, v, w) in the solid (Figure 4.2 (a)).

2. Determining the local coordinates (u, v, w) of all points (x, y, z) of the embedded object (Figure 4.2(b)) such that

$$\mathbf{f}(u, v, w) = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

This can either be done iteratively or, if it exists, by determining the inverse mapping $\mathbf{f}^{-1}(x, y, z)$.

3. Deforming the surrounding parametric solid by moving the control points from the original lattice positions, thus changing the basis functions. Let $\tilde{\mathbf{f}}(u, v, w)$ be the new parametric solid with displaced control points.

4. Determining the new points $(\tilde{x}, \tilde{y}, \tilde{z})$ of the embedded object (Figure 4.2 (c)). This is done by evaluating the modified basis functions

$$\tilde{\mathbf{f}}(u, v, w) = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{pmatrix}.$$

The choice of basis function determines what effect moving a control point has on the shape of the embedded object. Depending on the basis functions, moving a single control point can either deform the whole object (global deformation) or only parts of it (local deformation).

Next we use the Sederberg and Parry implementation to provide a more detailed discussion of FFD.

4.3 Deformation Using a Trivariate Bernstein Basis

The first step in defining an FFD is the choice of a deformation function $\mathbf{f}(u, v, w) : \mathcal{R}^3 \rightarrow \mathcal{R}^3$. The deformation function can be defined in terms of any polynomial or piecewise polynomial basis, such as a tensor product Bézier , B-spline or nonuniform rational B-spline (NURBS) basis or some non-tensor product volume such as a Bézier tetrahedron.

Both Bézier [Béz78] and Sederberg and Parry [SP86b] [Par86] use a tensor product trivariate Bernstein-Bézier polynomial basis

$$\mathbf{f}(u, v, w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{b}_{i,j,k} B_i^m(u) B_j^n(v) B_k^p(w).$$

The advantage of using a Bernstein basis is that there is a geometrically meaningful relationship between the displacement of a control point and the resulting deformation of the solid (cf. Section 2.5.2).

Having defined the deformation solid, our next step consists of finding the local coordinates of the object points. For a point with world coordinates (x, y, z), we want to find the local solid coordinates (u, v, w) such that

$$\mathbf{f}(u, v, w) = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

The solution to this problem depends on the trivariate basis used. While there is a closed form inversion of non-singular parametric curves and surfaces (Sederberg [SAG84]), there is generally no closed form inversion for trivariate hyperpatches (Sederberg [Sed83]), so it is not possible to express the local solid coordinates (u, v, w) as rational polynomial functions of the world space coordinates (x, y, z). Other functions that do not possess closed form inversions are piecewise polynomial bases such as B-splines and NURBS. Therefore, the inverse point problem usually involves general root finding, which is time consuming and prone to numerical errors.

The main advantage of using a Bézier basis is that it simplifies this inverse point problem. Let us for the moment assume that the embedded object has coordinates (x, y, z) in the range $[0, 1] \times [0, 1] \times [0, 1]$. Bézier curves have the useful *linear precision* property (Farin [Far93]):

$$f(u) = \sum_{i=0}^{m} b_i B_i^m(u) = u \tag{4.1}$$

if we set the control points $b_i = i/m$. This property extends to higher dimensions. For the trivariate case,

$$\mathbf{f}(u, v, w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{p} \mathbf{b}_{i,j,k}\, B_i^m(u) B_j^n(v) B_k^p(w) = \begin{pmatrix} u \\ v \\ w \end{pmatrix}, \tag{4.2}$$

if we arrange the control points in a parallelepipedical lattice

$$\mathbf{b}_{i,j,k} = \begin{pmatrix} i/m \\ j/n \\ k/p \end{pmatrix}. \tag{4.3}$$

We find that if we arrange our initial lattice points according to Equation (4.3), the local solid space coordinates (u, v, w) of a point are identical to its world space coordinates (x, y, z).

We can now alter the mapping function $\mathbf{f}$ by displacing the control points $\mathbf{b}_{i,j,k}$ from their original positions to new positions $\tilde{\mathbf{b}}_{i,j,k}$ and obtain the new mapping

$$\tilde{\mathbf{f}}(u,v,w) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p} \tilde{\mathbf{b}}_{i,j,k}\, B_i^m(u) B_j^n(v) B_k^p(w) = \begin{pmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{pmatrix}.$$

4.3.1 Object Representation

FFD performs a mapping $\mathcal{R}^3 \rightarrow \mathcal{R}^3$. It is independent of the underlying representation of the deformed object and thus can be applied to Constructive Solid Geometry (CSG) based solid objects as well as boundary representation (b-rep) objects. B-rep objects can be bounded by any analytic surface such as polygons, parametric surface patches or implicitly defined surfaces.

Although Bézier's original paper [Béz78] investigates the deformation of surfaces defined by tensor product Bézier patches and Parry [Par86] uses FFD to deform objects in a solid modelling environment, the majority of FFD applications (Sederberg and Parry [SP86b] [SP86a], Griessmair [GP89], Coquillart [Coq90] [CJ91] and Hsu [HHK92]) use polygonal objects. One reason for this concentration on polygonally bounded models is certainly their wide availability.

We will now examine the deformation process of b-rep models bounded by polygonal and Bézier patches.

4.4 Expressions for Deformed Objects

When we deform an object bounded by Bézier patches, the question arises how to compute surface points of the deformed model. One solution is to compute a surface point on the original patch, find its local solid coordinates and then transform it into deformed world space. However, it would be convenient to find a direct description of the deformed surface for two reasons. Firstly, it would give us a better idea of the type of surface we obtain through FFD. Secondly, we could render the deformed model directly from its analytic description which would provide a clean, well defined interface between modelling and rendering system.

We have shown that FFD is actually composed of two mapping processes: The first is the inverse point problem to map from the object's world coor-

dinates to the solid's parametric coordinates and the second is the actual deformation process that maps to deformed coordinates.

For a general trivariate polynomial basis there exists no closed form inversion to express the parametric solid coordinates (u, v, w) as functions of world coordinates (x, y, z) (and some affine transformation), so there will be no closed form expression for deformed solids either. We will therefore restrict our treatment to the special case of a deformation solid defined by a trivariate Bernstein basis with a parallelepipedical equally spaced lattice as in Equation (4.3). By Equation (4.2), the mapping

$$\mathbf{x} \xrightarrow{\mathbf{f}^{-1}} \mathbf{u} \xrightarrow{\tilde{\mathbf{f}}} \tilde{\mathbf{x}}.$$

can then be replaced by

$$\mathbf{x} \xrightarrow{\text{affine}} \mathbf{u} \xrightarrow{\tilde{\mathbf{f}}} \tilde{\mathbf{x}},$$

so the object's local coordinates (u, v, w) will now coincide with its world coordinates (x, y, z) and we only have to consider the deformation from parameter space to deformed space.

In the next two sections, we derive closed form expressions for objects that have been deformed by a trivariate Bernstein basis. We consider the deformation of curves and surfaces. Curves are represented in Bézier form, because the deformed curves can also be represented in Bézier form. We investigate the deformation of surfaces using Bézier triangles, because any planar polygonally bounded surface can be decomposed into a number of triangles. This decomposition process is the subject of Chapter 5.

4.4.1 Curves

We will first investigate the deformation of a parametric curve. A parametric curve of degree q, defined in local solid space, can be expressed in Bézier form as

$$\mathbf{g}(t) = \sum_{e=0}^{q} \mathbf{c}_e t^e = \begin{pmatrix} g_u(t) \\ g_v(t) \\ g_w(t) \end{pmatrix},$$

where $g_u(t)$,$g_v(t)$ and $g_w(t)$ are the projections of $\mathbf{g}(t)$ onto the u, v and w axes.

The deformation function is defined in terms of a trivariate Bernstein basis:

$$\mathbf{f}(u,v,w) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p} \mathbf{b}_{i,j,k}\, B_i^m(u) B_j^n(v) B_k^p(w) = \begin{pmatrix} u \\ v \\ w \end{pmatrix},$$

if the control points are initially arranged according to Equation (4.3).

By inserting the curve equation into the Bernstein basis and moving the control points to their new positions $\tilde{\mathbf{b}}_{i,j,k}$, we obtain an expression for the deformed curve:

$$\tilde{\mathbf{f}}(\mathbf{g}(t)) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p} \tilde{\mathbf{b}}_{i,j,k}\, B_i^m(g_u(t)) B_j^n(g_v(t)) B_k^p(g_w(t)).$$

Finding a direct expression for the deformed curve can now be posed as a functional composition problem of the form

$$\tilde{\mathbf{g}}(\mathbf{t}) = \tilde{\mathbf{f}}(\mathbf{g}(t)).$$

We can now apply the knowledge of functional composition of tensor product functions with polynomials that we derived in Chapter 3. The composed function is

$$\tilde{\mathbf{g}}(t) = \sum_{d=0}^{q(m+n+p)} \mathbf{a}_d B_d^{q(m+n+p)}(t).$$

A trivariate Bézier volume of degree (m,n,p) maps a curve of degree q in parameter space into a degree $(q(m+n+p))$ curve (Equation (3.8)). This means that a straight line would be deformed into a degree $(m+n+p)$ curve, where m,n,p are the degrees of the trivariate Bézier polynomial. To express the deformed curve in Bézier form, we can use the blossoming form algorithm discussed in Section 3.2.2 to find the Bézier points of the composed function.

4.4.2 Polygonal Surfaces

Any planar polygonal surface can be subdivided into a number of planar triangles, so we will use the triangle as the canonical primitive to investigate the effect that FFD has on surfaces.

We can express a triangle as

$$\mathbf{g}(\mathbf{t}) = \begin{pmatrix} g_u(\mathbf{t}) \\ g_v(\mathbf{t}) \\ g_w(\mathbf{t}) \end{pmatrix},$$

where $\mathbf{t} = (t_0, t_1, t_2)$ are the barycentric coordinates of a point inside the triangle.

Again using the trivariate Bernstein basis, we obtain the following expression for the deformed curve:

$$\tilde{\mathbf{f}}(\mathbf{g}(\mathbf{t})) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{p} \tilde{\mathbf{b}}_{i,j,k}\, B_i^m(g_u(\mathbf{t}))B_j^n(g_v(\mathbf{t}))B_k^p(g_w(\mathbf{t})).$$

Finding a direct expression for the deformed curve can now be posed as a functional composition problem of the form

$$\tilde{\mathbf{g}}(\mathbf{t}) = \tilde{\mathbf{f}}(\mathbf{g}(\mathbf{t})).$$

In Chapter 3, we showed that the composed function is of the form

$$\tilde{\mathbf{g}}(\mathbf{t}) = \sum_{|\mathbf{d}|=q(m+n+p)} \mathbf{a}_\mathbf{d} B_\mathbf{d}^{q(m+n+p)}(\mathbf{t}).$$

A planar triangle is a Bézier triangle of degree 1. Setting $q = 1$, we can see that a planar triangle gets mapped into a Bézier triangle of degree $(m + n + p)$.

4.5 Tangents and Normals

We have shown how to transform a point from world space via parameter space to deformed world space. Popular shading techniques such as Gouraud shading (Gouraud [Gou71]) or Phong shading (Bui-Tuong [BT75]) also require surface normals. If we want to render a deformed surface or obtain curvature information, it would therefore also be be useful to transform tangent and normal vectors into deformed space. Normal transformation is not mentioned in the existing FFD literature, although it has been used in a different context by Barr [Bar84].

Let $\mathbf{x} = (x, y, z)$ be the coordinates of a point in world space, $\mathbf{u} = (u, v, w)$ in parameter space and $\tilde{\mathbf{x}} = (\tilde{x}, \tilde{y}, \tilde{z})$ in deformed space. For a given transformation $\mathbf{f}(\mathbf{u}) = \mathbf{x}$, a tangent vector in parameter space $\mathbf{t}_\mathbf{u}$ can be transformed into a tangent vector in world space, $\mathbf{t}_\mathbf{x}$, by the *contravariant transformation* (Millman [MP77])

$$\mathbf{t}_\mathbf{x} = \mathbf{J}\mathbf{t}_\mathbf{u}, \tag{4.4}$$

where $\mathbf{J}$ is the *Jacobian matrix* given by

$$\mathbf{J} = \begin{pmatrix} \frac{\partial \mathbf{f}_x(\mathbf{u})}{\partial u} & \frac{\partial \mathbf{f}_x(\mathbf{u})}{\partial v} & \frac{\partial \mathbf{f}_x(\mathbf{u})}{\partial w} \\ \frac{\partial \mathbf{f}_y(\mathbf{u})}{\partial u} & \frac{\partial \mathbf{f}_y(\mathbf{u})}{\partial v} & \frac{\partial \mathbf{f}_y(\mathbf{u})}{\partial w} \\ \frac{\partial \mathbf{f}_z(\mathbf{u})}{\partial u} & \frac{\partial \mathbf{f}_z(\mathbf{u})}{\partial v} & \frac{\partial \mathbf{f}_z(\mathbf{u})}{\partial w} \end{pmatrix}$$

For the following discussion we assume that $\mathbf{J}$ is nonsingular, i.e. it has a unique inverse and its determinant is non-zero.

To retransform a tangent vector from world space back into parameter space, we can invert Equation (4.4) and obtain

$$\mathbf{t}_\mathbf{u} = \mathbf{J}^{-1}\mathbf{t}_\mathbf{x}. \tag{4.5}$$

Since the normal vector can be constructed from the cross product of any two tangent vectors at a point, we can also transform normal vectors to and from world space. Barr [Bar84] shows that a normal vector in parameter space $\mathbf{n}_\mathbf{u}$ can be transformed to world space by the *covariant transformation*

$$\mathbf{n}_\mathbf{x} = \det(\mathbf{J})\,\mathbf{J}^{-1T}\mathbf{n}_\mathbf{u}. \tag{4.6}$$

Since we are usually only interested in the direction of the normal vector, it is not necessary to compute the determinant.

Given a normal vector $\mathbf{n}_\mathbf{x}$ in world space, to find the corresponding vector $\mathbf{n}_\mathbf{u}$ in parameter space, we invert Equation (4.6) and obtain

$$\mathbf{n}_\mathbf{u} = (\det \mathbf{J})^{-1}\mathbf{J}^T\mathbf{n}_\mathbf{x}. \tag{4.7}$$

This gives us a mapping of normal vectors from and to world space.

FFD uses two functions, $\mathbf{f}(\mathbf{u})$ and $\tilde{\mathbf{f}}(\mathbf{u})$. $\mathbf{f}(\mathbf{u})$ transforms from parameter space to world space and $\tilde{\mathbf{f}}(\mathbf{u})$ transforms from parameter space to deformed space. To transform a point from world space to deformed space, we apply the mapping

$$\mathbf{x} \xrightarrow{\mathbf{f}^{-1}} \mathbf{u} \xrightarrow{\tilde{\mathbf{f}}} \tilde{\mathbf{x}}.$$

Equations (4.4) and (4.5) enable us to transform a tangent vector from world space to deformed space. Let $\mathbf{J}$ be the Jacobian of $\mathbf{f}(\mathbf{u})$ and $\tilde{\mathbf{J}}$ be the Jacobian of $\tilde{\mathbf{f}}(\mathbf{u})$. As $\mathbf{J}$ is assumed nonsingular, we can then apply a mapping

$$\mathbf{t}_\mathbf{x} \xrightarrow{\mathbf{J}^{-1}} \mathbf{t}_\mathbf{u} \xrightarrow{\tilde{\mathbf{J}}} \mathbf{t}_{\tilde{\mathbf{x}}}.$$

to transform a given surface tangent in world space, $\mathbf{t_x}$, to deformed world space:

$$\begin{aligned} \mathbf{t}_{\tilde{\mathbf{x}}} &= \mathbf{J}^{-1}\,\mathbf{t_u} \\ &= \mathbf{J}^{-1}\,\tilde{\mathbf{J}}\,\mathbf{t}_{\tilde{\mathbf{x}}} \end{aligned}$$

Similarly, we can use the mapping

$$\mathbf{n_x} \xrightarrow{(\det \mathbf{J})^{-1}\,\mathbf{J}^T} \mathbf{n_u} \xrightarrow{\det(\tilde{\mathbf{J}})\,\tilde{\mathbf{J}}^{-1T}} \mathbf{n}_{\tilde{\mathbf{x}}}.$$

to transform a given surface normal in world space, $\mathbf{n_x}$, to deformed world space:

$$\begin{aligned} \mathbf{n}_{\tilde{\mathbf{x}}} &= \det(\tilde{\mathbf{J}})\,\tilde{\mathbf{J}}^{-1T}\mathbf{n_u} \\ &= \det(\tilde{\mathbf{J}})\,\tilde{\mathbf{J}}^{-1T}(\det \mathbf{J})^{-1}\,\mathbf{J}^T\,\mathbf{n_x} \\ &= \det\tilde{\mathbf{J}}(\det \mathbf{J})^{-1}(\mathbf{J}\,\tilde{\mathbf{J}}^{-1})^T\mathbf{n_x}. \end{aligned}$$

Again, as we are usually only interested in the direction of the normal, it is not necessary to compute the two determinants.

Note that the transformation of normals does not require knowledge of the inverse $\mathbf{f}^{-1}(\mathbf{x}) = \mathbf{u}$ of our transformation function, it only requires that the determinants of $\mathbf{J}$ and $\tilde{\mathbf{J}}$ be non-zero. This makes this transformation also valid for B-spline and NURBS based transformations.

4.6 Extensions to FFD

Since the introduction of FFD to the graphics community, a number of extensions have been added. The main disadvantage of Sederberg's original version is that the control points have to be arranged as a regular, parallelepipedical lattice. Thus flexibility can only be added by increasing the number of control points, which also increases the degree of the polynomial basis. This not only makes evaluating the deformed points more expensive, it also increases the degree of the deformed surfaces. A cubic curve, mapped through a $10 \times 10 \times 10$ deformation lattice, will be of degree $3 \cdot (10 + 10 + 10) = 90$.

Coquillart introduces a technique called *Extended Free Form Deformation (EFFD)* [Coq90] [CJ91] that increases the flexibility of FFD by allowing arbitrary lattice geometry. The lattice is now composed of a number of tricubic

Bézier volumes, called "chunks". The lattice can be of any parallelepipedical or cylindrical shape and composite lattices can be formed by joining a number of lattices at their control points.

Each chunk corresponds to a $(3 \times 3 \times 3)$ deformation grid of Sederberg's original FFD, although the spacing of the internal lattice points does not have to be equal. Users are only allowed to manipulate the corner points of a chunk, while the application updates the other points to guarantee first order geometric continuity between the chunks. The determination of a point's local coordinates (u, v, w) is more complex, as the linear precision property (Equation (4.1)) does not hold. Thus, determining the local coordinates of embedded points (which Coquillart calls *freezing the lattice*) is done in two steps. First, the convex hull property of Bézier volumes is used to determine the chunk that contains the point. Then the point coordinates within the chunk are computed using Newton iteration. Coquillart reports no convergence problems. However, as the inverse-point problem has to be solved for each object point, the computational costs of EFFD are significantly higher.

FFD does not have to be expressed in terms of a Bézier basis. In fact, any set of parametric basis functions will do. Griessmair et al. have expressed FFD in terms of a trivariate B-spline basis [GP89]. Any trivariate B-spline volume can be split up into a number of trivariate Bézier volumes, with the degree of the Bézier volume being the degree of the spline basis functions. This reintroduces some form of local control. It also addresses the problem of the high degree of the resulting deformed object surfaces, as the degree of the deformation volumes no longer directly depends on the number of control points.

Although Griessmair was the first to use a B-spline basis in FFD, he did not address the inverse point problem. The inverse point problem, i.e. determining the local coordinates (u, v, w) of the points of the embedded object, needs special attention for B-splines as B-splines possess the linear precision property

$$f(u) = \sum_{i=0}^{m} d_i N_i^m(u) = u \tag{4.8}$$

only if we choose the control points according to

$$\mathbf{d}_i = \frac{1}{n} \left(t_i + \cdots + t_{i+n-1}\right)$$

where $[t_0, \ldots, t_{L+2n-2}]$ with $t_i \in \mathcal{R}$ is the knot vector.

If we choose all knots with multiplicity one, the image of the B-spline function does not fill the convex hull of the control lattice. We can change this by giving the outer control points a multiplicity of three, in which case the control lattice will no longer be evenly spaced.

If we insist on an evenly spaced lattice and a B-spline function whose image fills the convex hull of the control lattice, we have to abandon the linear precision property and find the parameter space coordinates (u, v, w) by general root finding, similar to the technique used in Coquillart's EFFD. This obviously significantly increases the computational expense of solving the inverse point problem.

The indirect deformation of objects through the displacement of control points is problematic in itself. The user needs to have a basic knowledge of Bézier curves or splines to understand the use of control points and the effect that moving a control point has on the shape of the object. It is difficult to predict the effect that a deformation will have on the object and this makes it difficult to achieve an exact shape. Finally, control points might be obstructed by the embedded object or a large number of control points might clutter the screen.

Hsu et al. [HHK92] have addressed this problem by developing a Free Form Deformation that allows the direct manipulation of object points rather then the control points of the surrounding solid. It is based on existing techniques for the direct manipulation of polygonal meshes (Parent [Par77]). The user interacts with the object by selecting an object point and moving it from its original position to a new position, called *target point*. Hsu developed an algorithm that moves the control points of the surrounding solid such that the resulting deformation will result in the selected object point getting moved to the target position. The problem is underdetermined as there are many control point displacements that will cause the same movement of a single object point. Hsu tries to find a least-square solution to the control point displacement. The resulting system is considerably more user-friendly than the original FFD as it keeps the underlying B-spline implementation details hidden from the user.

4.7 Summary

We have introduced Sederberg's Free Form Deformation (FFD) technique, which consists of indirectly deforming an object by embedding it in a flexible solid and then deforming this solid. FFD is independent of the underlying representation of the object and can thus be used to add free form flexibility to modelling schemes such as CSG that are based on a small number of predefined shapes.

One of the most popular applications of FFD is the deformation of polygonally bounded models; we have therefore provided a more detailed discussion of the deformation in the context of trivariate Bernstein Polynomials (Bézier hyperpatches) and polygonal surfaces and have shown how to evaluate deformed object points and normals. While it is generally not possible to find closed form expressions for deformed objects, we have shown that we can use the functional composition algorithms, derived in the previous chapter, to find expressions for deformed Bézier curves and triangles if they are deformed by trivariate Bernstein basis with a parallelepipedical lattice of control points. By inspection of the closed form expression of these deformed curves and triangles, we saw that FFD raises the degree of deformed surfaces significantly, which makes their evaluation expensive.

In the next chapter, we discuss ways to efficiently generate shaded images of deformed objects.

Chapter 5

Rendering of Deformed Models

5.1 Introduction

In this chapter, we investigate the rendering process of polygonally bounded objects that have undergone Free Form Deformation. FFD deformed models can be rendered using two main approaches:

- determining parametric expressions of the deformed objects that enable us to use existing rendering algorithms
- using a rendering algorithm that does not require such parametric surface expressions.

The first approach makes use of the composition algorithms discussed in Chapter 3. We have seen that for a certain class of deformation solids, Bézier volumes with parallelepipedical lattices of control points, we can express the deformed surfaces in closed form as an ensemble of Bézier patches. The rendering process of such curved surfaces is well investigated, see Coons [Coo67], Catmull [Cat74], Clark [Cla79], Lane and Carpenter [LC79] [LCWB80] and Lien et al. [LSP87] [SL87]. These approaches are based on two basic principles: iterative evaluation and adaptive subdivision. We start this chapter with a discussion of these two main approaches to rendering Bézier curves and surfaces.

Since a closed form expression of the deformed objects can in most cases not be obtained or is computationally very expensive, we then look at methods that do not require such closed form expressions. These methods replace

the object by a triangle mesh and render the deformed mesh. One advantage of such an approach is that it can also be applied to deformations by Bézier volumes with irregularly spaced grids or B-spline and NURBS volumes, where closed form expressions of the deformed objects usually do not exist. First, we investigate artefacts that can be introduced by this triangular approximation and then we discuss the two existing FFD-specific triangulation algorithms. Having identified their major problems, we then proceed to discuss Finite Element Method (FEM) meshing algorithms. Mesh generation is a central aspect of the Finite Element method and has received considerable attention. We evaluate several meshing algorithms with respect to Free Form Deformation applications and close the chapter with the identification of a meshing algorithm suitable for the FFD method.

5.2 Rendering of Deformed Polygons

In Chapter 3, we have shown that for a certain class of deformation solids, i.e. trivariate Bézier volumes defined by regularly spaced parallelepipedical lattices, it is possible to find the control points of Bézier curves and surfaces in deformed space. If we use these algorithms to obtain the parametric equation of a deformed patch in deformed space, then there are two main approaches to rendering of parametric curves and surfaces: *iterative evaluation* and *subdivision*. Iterative evaluation consists of rendering a surface directly from its analytical description (Catmull [Cat74]), whereas subdivision techniques recursively split a surface until the resulting segments are sufficiently flat and then these primitives are rendered using a fast primitive drawing routine.

5.2.1 Iterative Evaluation

Forward differencing (Coons [Coo67]) is a fast technique for evaluating polynomials at uniformly spaced intervals. It is based on the Gregory-Newton forward difference interpolation formula, which states that a degree n polynomial $\mathbf{f}(t)$ can be expressed at regular intervals $t_{j+1} = t_j + h$ as

$$\mathbf{f}(t) = \mathbf{f}_0 + \frac{1}{h}\Delta^1\mathbf{f}_0(t - t_0) + \cdots + \frac{1}{n!h^n}\Delta^n\mathbf{f}_0(t - t_0)\cdots(t - t_{n-1}),$$

with the differences[1] $\Delta^i \mathbf{f}_j$ given as

$$\Delta^i \mathbf{f}_j = \Delta^{i-1} \mathbf{f}_{j+1} - \Delta^{i-1} \mathbf{f}_j \tag{5.1}$$

and

$$\Delta^0 \mathbf{f}_j = \mathbf{f}(t_j).$$

If all the differences $\Delta^i \mathbf{f}_j$ at one point t_j are known, Equation (5.1) can be rearranged as

$$\Delta^i \mathbf{f}_{j+1} = \Delta^i \mathbf{f}_j + \Delta^{i+1} \mathbf{f}_j \tag{5.2}$$

and it is possible to reconstruct the polynomial at a point t_{j+1} just by additions. The advantage of the forward differencing algorithm is that every successive point on $\mathbf{f}$ can be evaluated using only n additions per dimension. After initialisation, the evaluation of an additional cubic curve point in $\mathcal{R}^3$ needs just 9 additions. For a tutorial on forward differencing, see Wallis [Wal90].

For rendering application, it is desirable to evaluate the polynomial once at every pixel location. This can be achieved by adapting the step size during the evaluation of the curve. The technique is called *adaptive forward differencing* and is discussed by Lien et al. [LSP87].

Forward differencing is also used to render biparametric patches. Patches are rendered as a collection of isoparametric curves, so they are evaluated by keeping one parameter constant and tracing the resulting curve. A discussion of the algorithm can be found in Foley et al. [FvDFH90].

Forward differencing can be implemented in hardware very efficiently and there are implementations for a number of cubic curves and bicubic surfaces such as Bézier patches. The direct application for rendering FFD surfaces is problematic, however, as they result in extremely high order patches for which hardware implementations usually do not exist and the high degree makes evaluation in software infeasibly slow.

A more general problem of forward differencing is that for large numbers of points the roundoff error introduced by the additions will significantly affect the accuracy and the results will increasingly deviate from the true values.

[1] note that Δ indicates here a difference and not a blossom

5.2.2 Recursive Subdivision

Another popular approach to displaying curves and curved surfaces is recursive subdivision (Lane and Carpenter [LC79]), a good introduction can be found in Watt and Watt [WW92]. The idea is to replace the curve or surface by a number of line segments or flat patches that represent a sufficiently close approximation of the curve or surface.

Obviously, we have to ensure that the generated mesh is as close to the original surface as possible. The finer we subdivide the surface, the more accurate is our resulting mesh. However, as we are generating more patches, we are also increasing our rendering costs. The advantage of recursive subdivision is that we can control the degree of accuracy with which we approximate the original surface, thus trading off accuracy for speed.

In Section 2.2.3 we have shown how to subdivide a Bézier curve and compute the new control points for the two resulting curves. The convex hull of the new control points will be closer to the original curve than the convex hull of the original points. If we repeat this process, we notice that the new control points move closer and closer to the original curve and provide a better approximation to the curve (Lane and Riesenfeld [LR80]) (Figure 5.1).

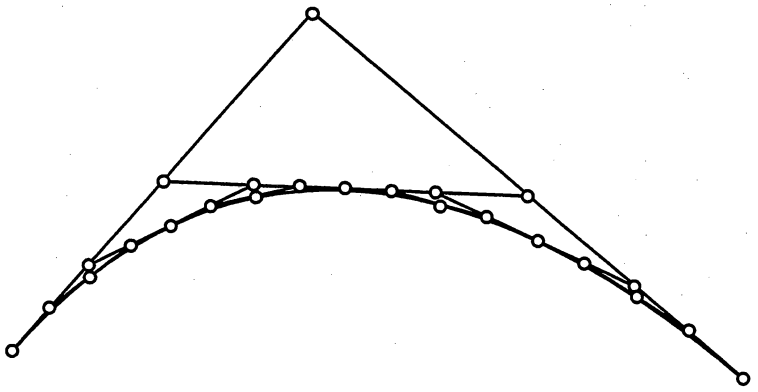


Figure 5.1: Approximation of a quadratic Bézier curve by repeated subdivision

We can use this to devise an algorithm to adaptively approximate a Bézier curve by straight line segments (Lane et al. [LCWB80]): Suppose the original curve is parametrised in u and defined over the interval $u = [0, 1]$. We now subdivide the curve about the midpoint $u = 0.5$ and compute the new

control points. The newly generated curves are again subdivided about their midpoint. This recursive subdivision process of a particular curve segment is terminated when the newly generated control points are sufficiently close to a straight line. Thus we can approximate a curve by repeatedly subdividing it and computing the new control points. The curve is then drawn by connecting the new control points by straight lines.

The subdivision technique can be easily extended to tensor product patches. The splitting process is then performed in each parameter independently. Subdivision of a patch along one parameter splits a patch into two, subsequent subdivision along the other parameter yields four subpatches. The recursive subdivision of a subpatch is terminated when the subdivided patch fulfills a predefined flatness criterion. Flatness could be determined by defining a plane through three corner points of a subpatch and finding the distance from the plane to the fourth corner. More sophisticated termination criteria would also take into account the projected area of the subpatch on the screen, as there is no use in subdividing a patch that only occupies one pixel on the screen.

5.3 FFD Specific Subdivision Methods

The previously discussed methods operate in the object's parameter space and thus assume we have a parametric representation of the deformed object. However, we have seen that obtaining this representation is only feasible for the most simple cases, i.e. deformation with a Bézier volume defined by a regularly spaced parallelepipedical lattice of control points. This is because for this kind of lattice there exists an analytic solution to the inverse point problem. For general lattice shapes, the inverse point problem (i.e. finding a point's parametric coordinates in deformation solid space) has to be solved iteratively, which makes it impossible to find a Bernstein-Bezier expression for the object in deformed space.

One solution to rendering these deformed objects is to substitute the parametric representation of the object in world space by a mesh, composed of triangular patches, apply the deformation to the mesh vertices and render the resulting deformed mesh.

A set $\Delta = \{T_i\}_{i=1}^{N}$ is called a triangulation of a domain Ω (Schumaker [Sch93b]) if:

- pairs of triangles intersect at most at a common vertex or a common edge.
- the union of the triangles $\{T_i\}_{i=1}^{N}$ is a connected set.

The triangle mesh represents an approximation to the original object, where the triangles lie as close to the object's surface as possible. Obviously, the higher the number of triangles, the closer we can approximate the curvature of the object. However, a large number of triangles also means a large number of points to map to deformed space and a large number of triangles to display, which increases the cost of rendering the deformed object.

It is therefore desirable to vary the size of the triangles, so small triangles can be used to closely approximate areas of high curvature and larger triangles can be used in flat regions of the object. This process is called *adaptive tessellation* or, for triangular elements, *adaptive triangulation* of an object. Adaptive triangulation has been applied extensively for rendering trimmed NURBS patches (see Abi-Ezzi and Shirman [AES91], Rockwood [RHD89], Sheng and Hirsch [SH92] and Piegl [PR95]). However, these methods are applicable to FFD if parametric expressions of the deformed surfaces have been determined.

5.3.1 Artefacts

Let us now identify two main artefacts that can be introduced when using the triangle mesh representation of an object.

T-vertices

T-vertices are vertices that lie on the edge of another triangle (Figure 5.2 (a)). After applying the deformation, this can result in a visible crack (shaded area in Figure 5.2 (b)). T-vertices can either already exist in an original polygonal model, or they can be introduced during the triangulation process.

Figure 5.3 (a) shows a t-vertex that has been introduced during the modelling process. It can be avoided by requiring the database to contain maximally connected coplanar faces, i.e. coplanar connected faces have to be represented by a single polygon (Figure 5.3 (b)). An algorithm to join coplanar connected faces is discussed by Baum [BMSW91]. However, joining all connected coplanar faces can lead to concave polygons or even concave polygons

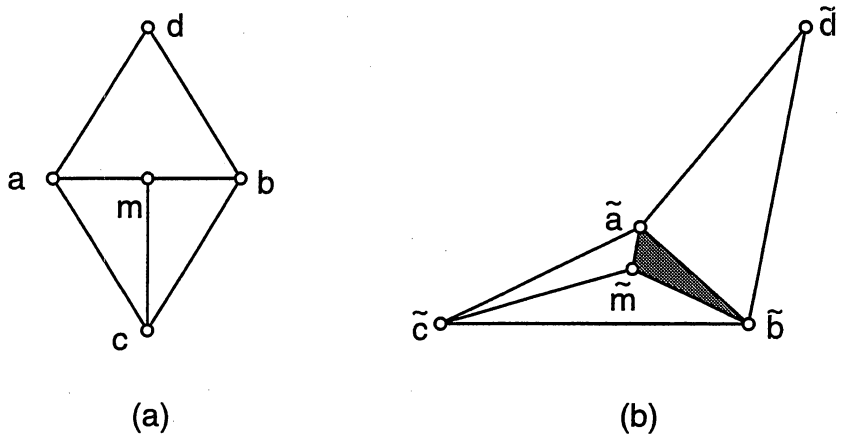


Figure 5.2: Crack caused by t-vertex

with holes, which puts additional demands on the subsequent triangulation routine.

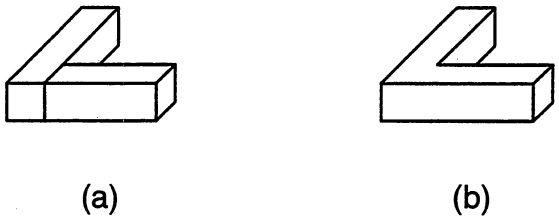


Figure 5.3: T-vertices (a) can be avoided by joining coplanar connected faces (b)

T-vertices can also be generated during the meshing process, if a triangle is subdivided along a shared edge and the other triangle sharing the same edge is not subdivided (Figure 5.2). This problem can be solved by always splitting both triangles along a shared edge, other solutions include the generation of filler polygons (Nydegger [Nyd72]) or forcing the shared edge to remain flat (Clark [Cla79]).

T-vertices are a topology feature. As the topology of the model remains unchanged during a deformation, the t-vertex criterion is independent of the space we triangulate in, i.e. if the triangulation of the undeformed model does not contain t-vertices, neither will the deformed model.

Slivers

Repeated subdivision of a triangle along the same edge can lead to triangles of long and thin shape with small interior angles, called *slivers* (Figure 5.4). Slivers are undesirable for several reasons: First, they causes artefacts when interpolated shading techniques such as Gouraud shading or Phong shading are employed (Foley et al. [FvDFH90]). Second, global illumination methods such as radiosity rely on well shaped triangles (Baum [BRW89]). A well shaped triangle is a triangle as equilateral as possible. A suitable measure to define the quality of shape of a triangle is its *aspect ratio* ρ, which is defined as the ratio of the radius of the inscribed circle to the radius of the circumscribed circle (Frey [Fre87]). The larger ρ, the better the shape of the triangle.

As we want to render models in deformed space, it is important to note that the aspect ratio is a geometric feature which changes as we move from world space to deformed space. Our intention is to obtain well shaped triangles in deformed space. Because of the distortion between world space and deformed space, well shaped triangles in world space do not have to lead to well shaped triangles in deformed space.

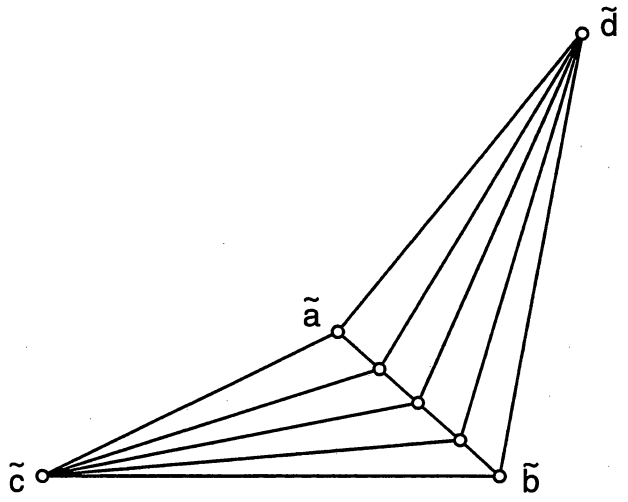


Figure 5.4: Generation of triangles of poor shape by repeated division of a triangle along the same edge

The main features of a good triangulation can be summarised as follows:

- approximates the shape of the object as closely as possible
- produces a minimum number of triangles
- produces well shaped triangles
- avoids cracks in the deformed object by not generating t-vertices.

Having identified desirable properties of a triangulation, we shall now examine two existing triangulation algorithms for FFD and see how they meet these criteria.

5.3.2 Parry's Mesh Generation

Parry [Par86] investigates FFD in a solid modelling environment and all his objects are composed of block, sphere and cylinder primitives. Each object has an initial, uniform triangulation assigned to it and the subsequent refinement is dependent on screen space and curvature.

The subdivision consists of splitting a triangle in half. In order to keep the triangles fairly equilateral, each triangle has one edge assigned to it along which to subdivide next. Parry calls this edge the *long side*, although it doesn't have to be the longest edge in the triangle. Initially, an arbitrary edge of each triangle is labelled long side. After the subdivision, the two sides that remained undivided become the long sides of the two newly generated triangles.

In order to prevent t-vertices, we also have to subdivide the triangle that shares the edge that we have just subdivided. However, triangles are only allowed to be split along their long side. So if the shared edge happens to be also the long side of the adjacent triangle, we can split the adjacent triangle and the subdivision process of this triangle stops. If they have different edges, a recursive subdivision process starts that subdivides adjacent triangles until two triangles with a shared long side are found.

The recursive subdivision process is shown in Figure 5.5. Triangle A's long side (indicated by a dotted line) is to be subdivided (Figure 5.5 (a)). Since A and its neighbour B do not share their long side, we have to progress to C to find a pair of triangles with a common long side(Figure 5.5 (a)). We now split B and C along their common long side (Figure 5.5 (b)), which means

that A and B now share a common long side and can be split (Figure 5.5 (c)).

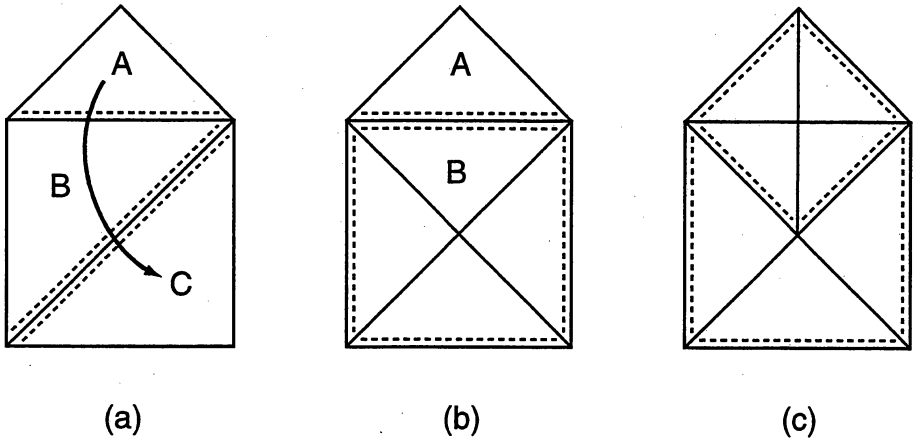


Figure 5.5: Mesh refinement with Parry's algorithm: long sides are shown as dotted lines

The previous algorithm shows how to subdivide a triangle and avoid artefacts such as t-vertices and poorly shaped triangles. Parry uses two criteria to decide which triangles to subdivide: the screen space size of a triangle and its curvature. A triangle is subdivided if its screen size exceeds a certain measure or if the angle between its normal and the normal of an adjacent triangle is greater than some pre-defined value.

5.3.3 Griessmair's Algorithm

Griessmair and Purgathofer[GP89] develop an alternative subdivision method that is based on individual edges rather than on triangles. All edges are stored in a heap structure according to some measure of quality. The lowest quality edge is taken off the heap, split, and the resulting new edges are stored in the heap. Figure 5.6 shows an edge (**ab**) with the two adjacent triangles in world space and the corresponding edge $\tilde{\mathbf{a}}\tilde{\mathbf{b}}$ in deformed space. It is important to recall, however, that a straight line in world space maps into a Bézier or B-spline curve in deformed space, so the displayed edges are only approximations to the actual edges in deformed space. This is the reason why the deformed point $\tilde{\mathbf{m}}$ generally will not be mapped onto the midpoint between $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$.

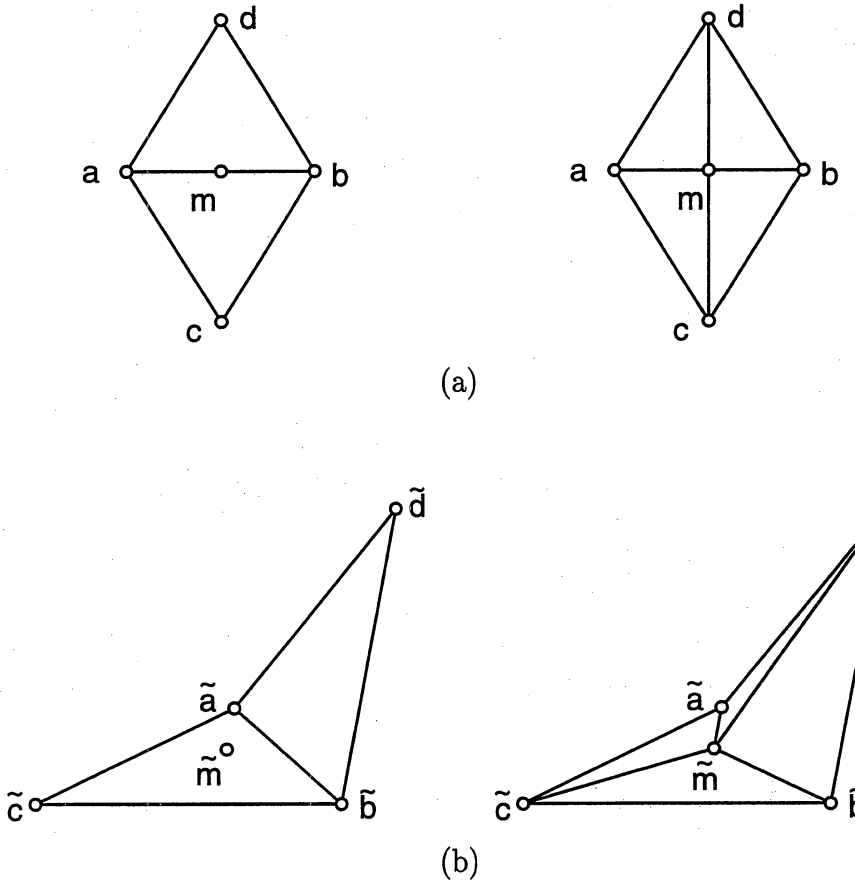


Figure 5.6: Edge splitting in undeformed (a) and deformed (b) space

Suppose edge **ab** is the lowest quality edge. It is taken off the heap, split along its midpoint **m**, and the four resulting new edges **am**, **bm**, **cm** and **dm** are stored on the heap. The strength of this method lies in the fact that the heap operations storing, deleting and retrieving can be performed very efficiently.

To assess the quality of an edge, one could therefore measure the distance between its deformed midpoint $\tilde{\mathbf{m}}$ and the midpoint on the straight line between the two deformed endpoints $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$, they will generally not coincide. While this measure actually works well in practice (Sederberg [Sed93]), Griessmair shows some cases where this proves too conservative and leads to unnecessary subdivisions (one example is the case when the midpoint $\tilde{\mathbf{m}}$ lies close to the plane of the two triangles $\tilde{\mathbf{a}}\tilde{\mathbf{c}}\tilde{\mathbf{b}}$ and $\tilde{\mathbf{a}}\tilde{\mathbf{b}}\tilde{\mathbf{d}}$. The improved measure of edge quality therefore consists of the sum of distances between the midpoint $\tilde{\mathbf{m}}$ and planes of the two triangles $\tilde{\mathbf{a}}\tilde{\mathbf{c}}\tilde{\mathbf{b}}$ and $\tilde{\mathbf{a}}\tilde{\mathbf{b}}\tilde{\mathbf{d}}$ (Figure 5.7).

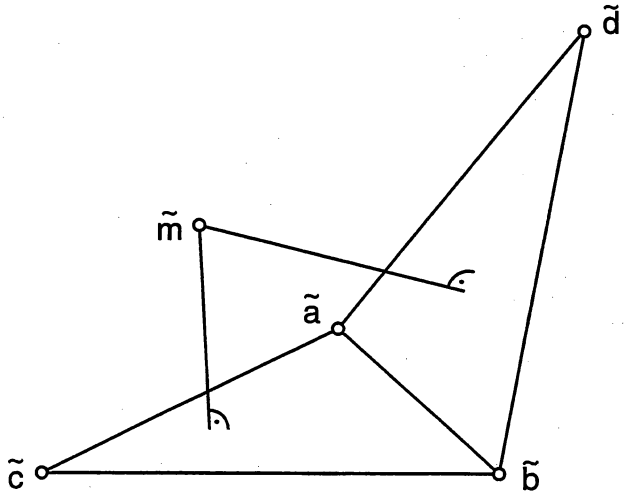


Figure 5.7: Measure of edge quality

While Griessmair develops a measure of the quality of edges, his algorithm does not address the problem of graded meshes or equilateral triangles. Although the generation of t-vertices is avoided, repeated subdivision along the same edge of a triangle can lead to new triangles of poor aspect ratio.

Both Parry's and Griessmair's algorithms require the existence of some initial triangulation. Since Parry's algorithm is designed for a solid modelling environment with a limited number of modelling primitives (block, sphere, cylinder), he can predefine an initial triangulation for each primitive and

stores it with the primitive. In a deformation system for general polygonally bounded objects, finding the initial triangulation of the objects is not obvious, especially for concave domains of complex shape.

Triangulation problems of surfaces and objects are frequently encountered in finite element computations. We will therefore discuss different finite element meshing algorithms and assess their suitability for FFD applications.

5.4 Finite Element Meshing

The problem of domain discretisation, also referred to as *meshing*, is an important part of finite element analysis and has received considerable attention. A variety of different approaches have been investigated and surveys can be found in George [Geo91], Shepard [She88] and Ho-Le [HL88]. Cohen and Wallace [CW93] discuss mesh generation in the context of radiosity methods.

Most algorithms were initially developed for the discretisation of two-dimensional planar domains and were only later extended to handle three-dimensional input. Two dimensional input consists of planes and usually produces triangular or quadrilateral input, whereas three dimensional input consists of solids and produces tetrahedral or sometimes brick elements. Although for most two-dimensional mesh generation algorithms an extension to three dimensions is possible, it is difficult to ensure the generation of well shaped elements.

A first classification of mesh generation algorithms can be made based on the element type they produce. Most of the current automatic mesh generators produce simplex meshes, that is triangular elements in two dimensions and tetrahedral elements in three dimensions. While these elements are known to be problematic in certain finite element areas such as stress analysis, they are useful for most computer graphics applications.

Meshes can be either *structured* or *unstructured*. A structured mesh has a predetermined topology, i.e. each vertex is connected to a predefined number of neighbours. A rectangular grid is an example of a structured mesh. In an unstructured mesh, each node can be connected to a different number of neighbours, but the type of the generated elements (triangles, quadrilaterals) is usually predefined (Figure 5.8). Some mesh generators produce unstructured meshes of mixed element type.

Mesh generation algorithms can be classified according to a variety of

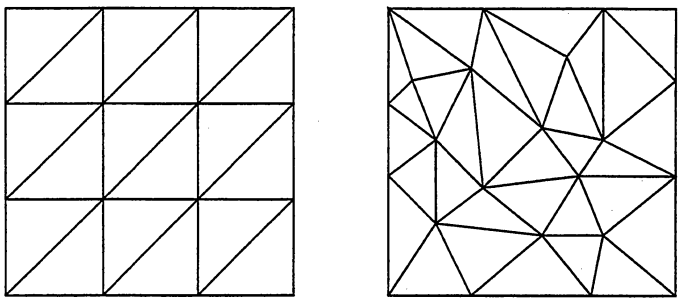


Figure 5.8: Structured (a) and unstructured (b) mesh

different criteria. For this discussion, we will organise meshing algorithms according to the order in which topology (node connectivity) and geometry (node position) are determined:

- topology first
- nodes first
- nodes and topology together

Most meshing algorithms decide on either the node placement or the topology of a generated element first. We will classify algorithms as topology first or nodes first if the decision on node placement and topology is done in two distinct phases of the algorithm.

5.4.1 Topology First

Most early mesh generators were based on the mesh template approach and produced structured meshes (Zienkiewicz and Phillips [ZP71]). These meshes are constructed by mapping a template, such as a rectangular grid, onto the polygon and subdividing the polygon accordingly (Figure 5.9). The topology of the resulting mesh is determined by the template.

The mapping between template and polygon is the crucial step in the mapped meshing process. Only if the mapping can be done without much distortion of the template, will a well shaped mesh be generated. The quality of the generated mesh can be improved by moving the generated nodes in a

postprocessing step. The most popular technique is *Laplacian smoothing* and iteratively moves each internal node into the centroid of the polygon consisting of its connected neighbour vertices. However, the resulting improvement is limited by the fact that only the position of existing nodes can be changed but no nodes can be added or deleted.

Since the distortion introduced by the mapping process becomes more severe with increasing complexity of the polygon shape, mapped meshing approaches are only suitable for relatively simple polygon shapes. Complex polygons have to be decomposed into mappable, less complex regions. This can either be done by hand or by automatic decomposition methods which will be discussed later, in Section 5.4.3.

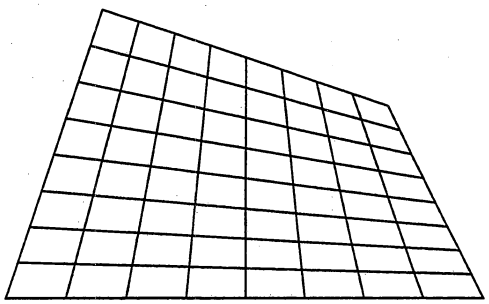


Figure 5.9: Mapped Mesh

5.4.2 Nodes First Meshing

In this approach, meshes are generated in two distinct steps:

1. Node generation. Nodes are placed inside or on the surface of the object. The initial placement of the nodes largely influences the quality of the generated mesh. Varying the number of the generated nodes allows the user to make the mesh finer or coarser in different areas.

2. Element generation. Once a set of nodes has been generated, they are connected to form elements that are usually triangular or tetrahedral. Probably the most popular method to connect a set of generated points that works both in two and three dimensions is the *Delaunay Triangulation.*

Since Delaunay Triangulation has been employed by a number of finite element mesh generation schemes, we will discuss it first and then proceed with the discussion of mesh generation schemes that make use of it.

Delaunay Triangulation

Delaunay triangulation is one of the best studied algorithms in Computational Geometry (Shamos [Sha78], Preparata and Shamos [PS85]. The basic property that makes Delaunay triangulation so appealing to finite element methods is that it maximises the smallest interior angle over all triangulations (Edelsbrunner [Ede87]), that is, the resulting triangulation is as equilateral as possible.

A convenient way of generating the Delaunay triangulation of a set of points P in a plane is the construction of the *Voronoi Diagram.* A Voronoi diagram is the subdivision of a polygon into polygonally bounded regions V_i, each associated with a point $\mathbf{p}_i \in P$, such that every point in its region is closer to $\mathbf{p}_i$ than to any other other point $\mathbf{p}$ in P:

$$V(\mathbf{p}_i) = \{\mathbf{x} : |\mathbf{p}_i - \mathbf{x}| \leq |\mathbf{p}_j - \mathbf{x}|, \forall j \neq i\}$$

The Delaunay triangulation is the straight line dual of the Voronoi diagram, hence it can be constructed by connecting pairs of points that share a Voronoi boundary. Figure 5.10 shows a Voronoi diagram and the corresponding Delaunay triangulation.

The popularity of the Delaunay triangulation within the computational geometry community has led to a number of very efficient construction algorithms. A popular, easy to implement algorithm by Bowyer [Bow81] and a similar version by Watson [Wat81] both use an incremental approach and run in $O(n^2)$ time, with n being the number of nodes. An $O(n \log n)$ sweepline algorithm is introduced by Fortune [For87].

One serious limitation for the application of Delaunay triangulation for finite element mesh generation is that it generates a tessellation of the convex hull of the polygon vertices. For a concave polygon, the convex hull of the polygon is different from the actual polygon boundary, which means that some of the actual polygon boundary edges might not be included in the tessellation. The problem can be solved by rejecting elements that are not within the boundary. Proper placement of nodes both inside and outside the domain ensures that the element boundaries are actually part of the

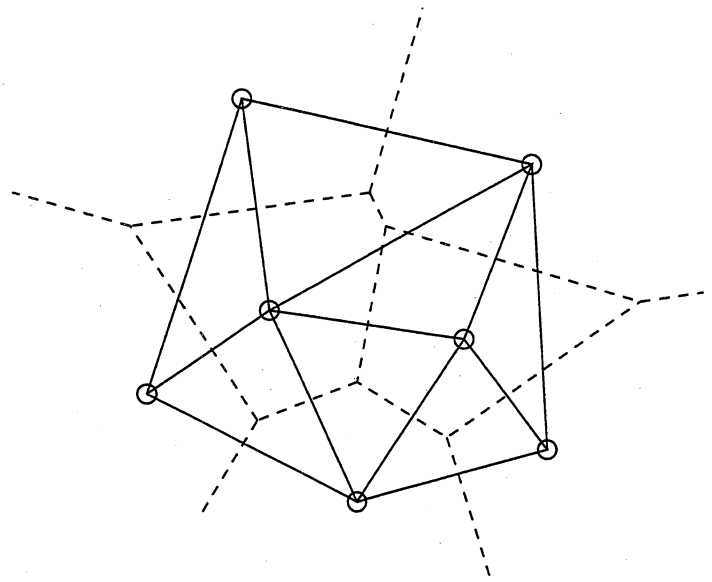


Figure 5.10: Voronoi diagram (dashed) and Delaunay triangulation

tessellation, i.e. no elements cross the domain boundary (Cavendish et al. [CFF85]).

Another approach to meshing concave regions is the *Constrained Delaunay triangulation* (de Floriani et al. [dFFP85]), which provides a triangulation as close as possible to the Delaunay triangulation given that certain edges (i.e. the polygon boundary) must be included in the generated triangulation. The constrained Delaunay triangulation has the advantage that arbitrary edges can be included in the final triangulation, this makes meshing of regions with holes possible. Details on how to construct a constrained Delaunay triangulation can be found in Chew [Che89].

Node placement

Delaunay Triangulation was first used for finite element mesh generation by Cavendish et al. [CFF85]. They employ a semi-automatic node placement strategy described in Cavendish [Cav74] which requires the user to specify zones in the object with a desired node density assigned to them. Using some random function, the algorithm automatically fills these zones with nodes of the specified density.

Frey [Fre87] develops an extension to this algorithm that not only allows to generate an *initial mesh* but also to *selectively refine* the mesh such that it satisfies some measure of accuracy.

Conceptually, the algorithm consists of the following steps:

1. Discretise the domain boundaries by inserting nodes into the edges.
2. Generate a Delaunay triangulation of only these boundary nodes (Figure 5.11 (a)).
3. Identify a triangle that is too large (Figure 5.11 (b)).
4. Insert a new node into the interior of the too large triangle (Figure 5.11 (c)).
5. Update the Delaunay triangulation by inserting the new node (Figure 5.11 (d)).
6. Repeat steps 2–5 until no large triangles are left.

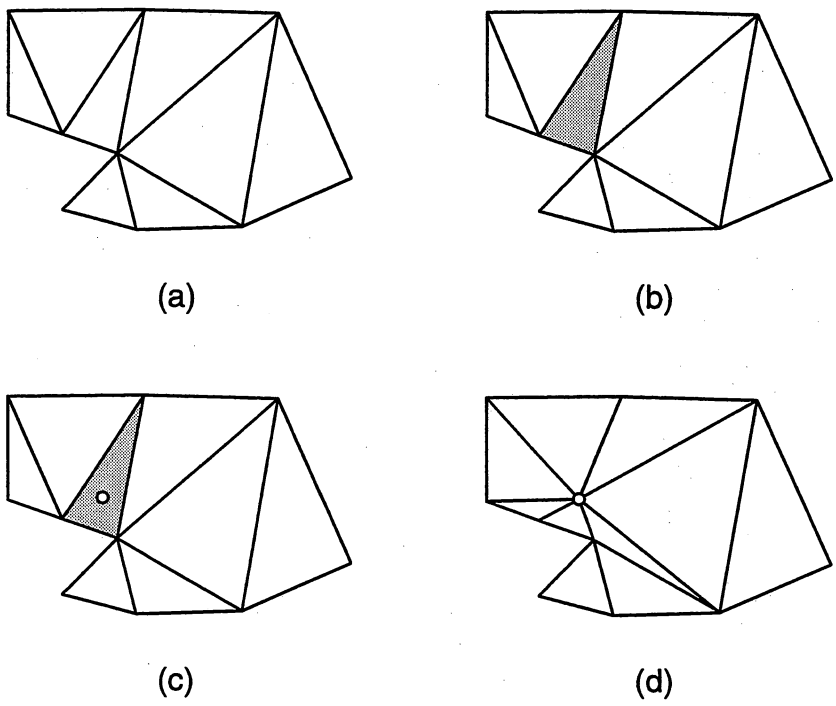


Figure 5.11: Mesh generation by selective refinement: Boundary triangulation (a), identification of a too large triangle (b), placement of new node (c), update of triangulation (d)

There are different measures to decide which triangles are considered too large and where inside a triangle to place new nodes, see Frey [Fre87] for

details. An incremental implementation of the Delaunay triangulation such as Watson's algorithm [Wat81] is most suitable for this meshing scheme, as it takes advantage of the fact that the interior nodes are being inserted one at a time.

5.4.3 Nodes and Topology Together

We will now discuss methods that perform the node placement and the composition of elements in one step. Popular algorithms that follow this approach are:

- removal of individual subdomains
- advancing front
- spatial decomposition and subdomain meshing

Removal of Individual Subdomains

Meshing by subdomain removal repeatedly splits off pieces of the domain until the whole remaining domain consists of one acceptable piece. Most of the subdomain removal algorithms split off pieces that will be directly used as elements (Wördenweber [Wör81]), but there are also schemes available that use the subdomain removal to decompose a complex region into simpler parts and then use a different technique (such as mapped meshing) to mesh these parts (Joe and Simpson [JS86]). Care must be taken to join these individually meshed regions properly and not to generate t-vertices.

The subdomain removal is usually done in terms of high-level operators that are used to split off vertices, edges or whole faces. Wördenweber [Wör81] reports a $O(n^2)$ performance for his algorithm. However, the algorithms usually rely on a set of complex rules to decide which vertices or edges to split off next, which makes a robust implementation of the algorithm difficult to achieve.

Figure 5.12 shows a mesh generated by direct element removal (adapted from Wördenweber [Wör81]). The mesh in Figure 5.13 was generated by removal of individual subdomains and subsequent mapped meshing.

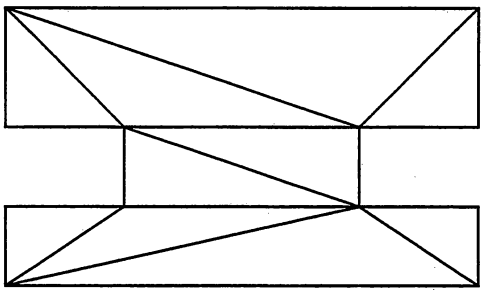


Figure 5.12: Mesh generated by removal of individual elements

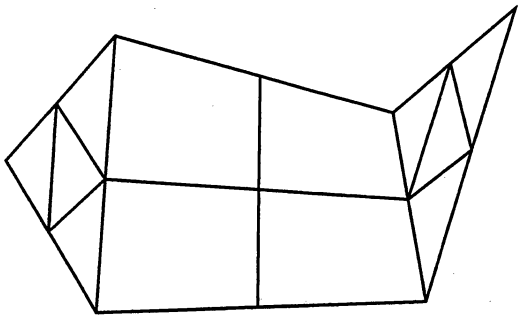


Figure 5.13: Subdomain removal and subsequent mapped meshing

Advancing Front

The advancing front technique is similar to subdomain removal algorithms. It starts off from the domain boundary and splits off finished elements one at a time until the entire domain is converted into elements (Lo [Lo85], Peraire [PVMZ88]). The current front consists of the boundary of the unmeshed domain and advances into the interior of the domain as new elements are formed. Different algorithms exist to decide which element of the current front to process next and how to determine the desired shape of new elements.

Advancing front methods are capable of meshing domains of arbitrary shape, including concave polygons with holes. However, a careful design is necessary to ensure that the algorithm terminates for concave domains with small interior angles.

Spatial Decomposition and Subdomain Meshing

Spatial decomposition followed by subdomain meshing is a two-step approach. The first step consists of a spatial subdivision of the initial domain

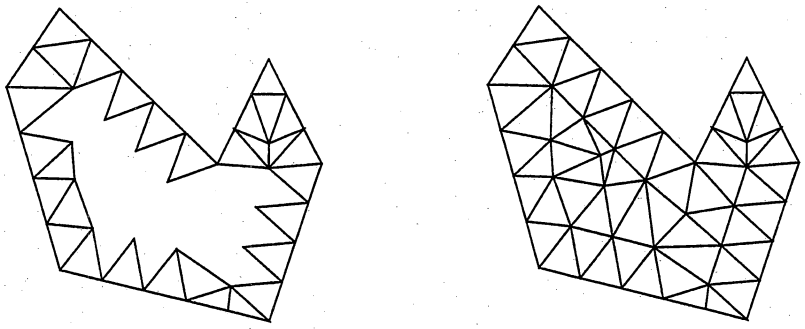


Figure 5.14: Advancing front: propagation of the front (a) and final mesh (b)

into simple cells and in the second step these cells are meshed individually.

The initial decomposition is usually done by a quadtree in two dimensional domains and by an octree in three dimensional domains. Quadtree based meshing was introduced by Yerry and Shepard [YS83], an octree version for three dimensional domains can be found in Yerry and Shepard [YS84] and a description of an improved quadtree version can be found in Baehmann et al. [BWS+87].

For a two dimensional domain, the mesh is generated using the following steps: The domain (a non self-penetrating polygon of convex or concave shape, possibly with holes) is subdivided into the quadrants of a quadtree with the quadrant size defining the mesh density. This subdivision is done in terms of polygon edges, outside boundaries are oriented counterclockwise and inside boundaries (holes) are oriented clockwise. Each edge of the polygon is recursively subdivided until each fragment fits into a quadrant of the tree and the mesh density is sufficient (Figure 5.15 (a)). In order to ensure a certain homogeneity of the mesh, each quadrant is allowed to be only one tree level different from its adjacent quadrants. Each quadrant is then classified as either interior, exterior or containing a boundary. Exterior quadrants are simply discarded, while interior quadrants are meshed using templates and boundary quadrants are meshed using previously discussed meshing algorithms such as subdomain removal (Figure 5.15 (b)) or, more recently,

Delaunay triangulation (Schroeder and Shepard [SS90]). Finally, Laplacian smoothing can be employed to improve the mesh quality.

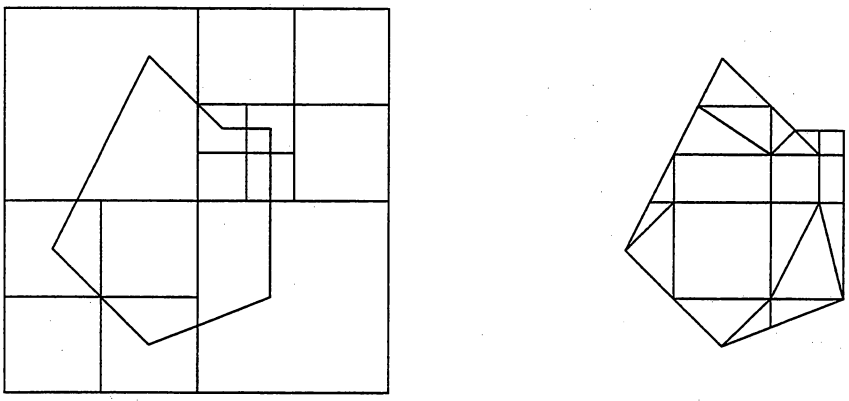


Figure 5.15: Spatial decomposition by quad tree (a) and resulting mesh before smoothing (b)

5.5 Application of FEM Meshing to FFD

The above finite element meshing methods decompose a domain in $\mathcal{R}^2$. In FFD of polygonal objects, our objective is the discretisation of planar polygons bounding a three dimensional object in $\mathcal{R}^3$. To discretise such an object, we can either use an algorithm capable of decomposing the surface of the whole object or we can discretise each surface patch separately. As each bounding surface of the undeformed object is planar, we can transform each patch into a plane and then use a planar mesh generation algorithm to decompose each patch individually. The decomposed surface is then retransformed into $\mathcal{R}^3$. Of course, reducing the 3D meshing problem to a number of 2D patch meshing problems simplifies the decomposition algorithm significantly.

However, if we mesh each surface individually in $\mathcal{R}^2$, care must be taken that after retransformation into $\mathcal{R}^3$ the individually meshed surfaces join properly and no t-vertices are generated along shared edges. Achieving this continuity property across shared edges is more difficult with some meshing algorithms than with others. Subdomain removal algorithms obtain it automatically by trying to split whole edges, and if no edges will be subdivided, no t-vertices can be created. For Delaunay triangulation and advanc-

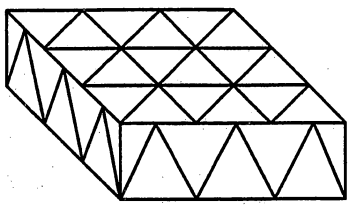


Figure 5.16: Generation of t-vertices introduced by individual meshing of surfaces

ing front algorithms, the problem can easily be avoided by subdividing the edges first and meshing the interior afterwards. Ensuring the same shared vertices across edges is more difficult for spatial decomposition algorithms and can probably only be done by moving vertices along shared edges in a postprocess.

5.5.1 Comparison of the Algorithms

Other properties that are important when identifying a finite element meshing method suitable for FFD include

- aspect ratio of the generated triangles,
- control over the shape of the generated triangles
- control over mesh density
- time efficiency.

We will now discuss how these properties are met by different FEM meshing methods.

Aspect Ratio

Let us first look at the quality of the original, undeformed mesh. Given an existing set of nodes, Delaunay triangulation provides by definition the mesh with triangles as equilateral as possible. The problem lies in an optimal initial node placement and the fact that the Delaunay triangulation is only defined for convex domains. Advancing front algorithms usually generate well shaped triangles, although the quality of the mesh is largely influenced by the implementation. The mesh quality of spatial decomposition algorithms

is excellent for interior elements, but the generation of well shaped elements close to the domain boundary tends to be difficult. Subdomain removal algorithms depend on the topology of the domain, so high quality meshes are difficult to guarantee. In summary, all meshing algorithms except subdomain removal lead to meshes of reasonable quality in the undeformed domain.

Mesh Size and Grading

A property that provides an important criterion for the selection of a meshing algorithm is the control over mesh size and grading. We have to be able to influence the triangle size as we want to adapt the size to the current curvature. Moreover, as the mesh generation takes place in undeformed space, it is desirable to have some influence on the shape of the generated triangles such that the triangulation in deformed space meets certain criteria. It is important to realise that a triangulation which is as equilateral as possible in undeformed space is not always desirable, the requirements of FFD and finite element methods are different here. Firstly, an equilateral triangulation of the undeformed model does not necessarily result in a equilateral triangulation of the deformed model, because the deformation might result in some stretching of the original domain. Secondly, although triangles with extremely poor aspect ratio (slivers) introduce rendering artefacts and are thus unwanted, for FFD a certain stretching of triangles along a direction of low curvature can be desirable.

Only the advancing front algorithm allows some form of control over the shape of the generated triangles during the generation process. This is achieved because triangles and nodes are generated together on a per-element basis, so each element can be stretched into some desirable direction.

The ability to control the size of the generated elements is of crucial importance to the application of any mesh generator in FFD. Spatial decomposition, advancing front and Delaunay triangulation are capable of producing triangles of varying size and thus generate graded meshes. The element size of mapped mesh generator meshes is determined by the mesh template, whereas for subdomain removal algorithms the topology of the domain defines the size of the generated triangles.

Time Efficiency

The mapped mesh generator shows the best time efficiency and runs in $O(n)$ time, with n being the number of triangles generated. Delaunay triangulations and spatial decompositions can be achieved in $O(n \log n)$ time. Advancing front is essentially a rule based algorithm and its execution time depends largely on the types of rules employed. It usually involves nearest node or closest edge searches, which can be supported by appropriate data structures, so an execution time of $O(n \log n)$ seems realistic (Lohner [Löh88]). Subdomain removal algorithms have with $O(n^2)$ the lowest time efficiency.

Discussion

Table 5.5.1 shows a summary of the comparison of the different meshing algorithms.

Approach	Aspect ratio	Density control	Shape control	Time efficiency
Mesh template	good (simple domain)	no	no	$O(n)$
Subdomain removal	poor	no	no	$O(n^2)$
Spatial decomposition	internal excellent	yes	no	$O(n \log n)$
Delaunay triangulation	excellent	yes	no	$O(n \log n)$
Advancing front	good	yes	yes	$O(n \log n)$

Table 5.1: Comparison of the different meshing approaches

Mesh template algorithms are fast, but their inability to cope with complex domain shapes and their lack of mesh density and shape control make them unsuitable for FFD. Subdomain removal algorithms do not score high in any area, in particular do not allow any control over mesh density and shape. Spatial decomposition algorithms are significantly better suited, they pro-

duce well shaped meshes, especially in the domain interior. Their mesh quality can be further enhanced by a Laplacian smoothing postprocess. Furthermore, they allow control over the element size and produce graded meshes. Their major shortcoming is that it is not clear how to ensure mesh compatibility across shared edges between two domains. Delaunay triangulation produces a triangulation as equilateral as possible which makes it extremely popular both in the finite element and computer graphics community. This popularity means that a number of efficient algorithms have been developed as well as extensions to handle concave domains or include certain interior edges of the domain. Dependent on the initial node placement, graded meshes can be achieved. However, for FFD applications, an equilateral triangulation is not always desirable, so an edge swapping algorithm that adapts the triangulation to the curvature is usually applied as a postprocess, see Schumaker [Sch93a]. Advancing front algorithms produce well shaped meshes and allow a lot of control over mesh density and element shape, which is reflected in their popularity in the computational fluid dynamic (CFD) community. FFD is essentially a three dimensional vector field of displacement vectors and has thus a lot in common with CFD, so it seem that an advancing front based algorithm would be a promising choice for FFD meshing. On the downside, the lack of a theoretical foundation for advancing front algorithms makes it difficult to ensure convergence.

The two best candidates for FFD meshing thus seem Delaunay triangulation and advancing front. We decided to use an advancing front algorithm for two reasons:

- When meshing FFD deformed objects, it is desirable to obtain triangles of good aspect ratio, i.e. as equilateral as possible. In FFD meshing, the triangulation takes place in the undeformed domain, whereas it is the deformed domain that is actually being displayed. Applying a Delaunay-based algorithm will guarantee a triangulation as equilateral as possible in the undeformed domain, which will not necessarily result in an optimal triangulation in the deformed domain, as the deformed domain can be substantially distorted (stretched or squashed). Equilateral triangles in the undeformed domain could therefore yield badly shaped triangles in the deformed domain. The advancing front algorithm offers us superior control over individual elements, as we can can not only influence their shape but also stretch them into certain di-

rections. It seems worth investigating how this additional control can actually be used to generate triangulations that result in better graded deformed meshes.

- Although well established in the finite element community, the advancing front algorithm has not yet made its way into computer graphics. One of its problems is that it is based on a set of complex heuristics which makes it hard to code a robust implementation. We therefore felt that gaining experience of its applicability to computer graphics problems would be worthwhile. The application of the advancing front meshing scheme to a number of different meshing problems should answer the question whether it is sufficiently robust to be employed in a general rendering system.

5.6 Summary

We have shown the two standard approaches to rendering curved patches: forward differencing and recursive subdivision. However, it is usually not feasible to compute the closed form expressions of the deformed surfaces. We therefore have presented Parry's and Griessmair's approach to subdividing the model into triangles and rendering the resulting triangle meshes. As these algorithms are unable to cope with objects of complex shape, we have surveyed FEM meshing algorithms and identified the advancing front algorithm as a suitable candidate to adapt to the specific needs of Free Form Deformation.

We will now develop a implementation of the modified advancing front algorithm for rendering Free Form Deformation objects.

Chapter 6

A Modified Advancing Front Algorithm

6.1 Introduction

The advancing front algorithm is well established in computational fluid dynamics applications and has undergone several stages of development. We start this chapter with a brief overview of the major contributions to the algorithm. We then discuss in detail the implementation of our FFD-specific version of the algorithm.

This discussion consists of two parts: how to generate the triangles and how to determine the triangle size according to curvature. The first part starts with the preprocessing stage and then gives a detailed description of our version of the advancing front technique. In the second part, we develop a local curvature measure for the deformed surfaces and show how to use it to determine the optimal triangle size.

6.2 Overview

Some early ideas of the advancing front technique can be found in (Gill [Gil72]), who developed a meshing scheme for Finite Element computations of shells defined by Coons patches. The original advancing front algorithm was introduced by Lo [Lo85] for general meshing problems in Finite Element computations. It can generate an unstructured triangular element mesh within any simply connected or multiply connected planar domain with or

without holes. The boundary of the domain is represented as a set of straight line segments which are entered in counterclockwise order for domain boundary nodes and in clockwise order for interior boundary nodes (holes). Lo's original mesher is actually a nodes first algorithm: it first generates interior nodes according to the average nodal spacing of all the boundary segments that make up the domain to be triangulated. Starting from the domain boundary, these nodes are then connected to triangular elements whilst ensuring that none of the generated triangles intersect and the entire domain is covered. The connection algorithm ensures that the generated triangles are as equilateral as the set of generated nodes permits.

Peraire et al. [PVMZ88] provide the first extension to the algorithm that makes it actually a *nodes and topology together* approach. It has been developed for the solution of steady state flow problems in fluid dynamics. Like Lo's algorithm, they start off with the line segments that connect the domain boundary nodes. All the segments that are available to form a triangle are called active and form the *active front.* Peraire's algorithm differs in how to form triangles. Rather than generating the interior nodes in advance, nodes and triangles are generated simultaneously as the active front proceeds through the domain. The algorithm requires the user to define a background grid over the domain to be meshed that contains the meshing parameters such as triangle size, stretching and direction. This technique has been extended to three dimensions (Peraire et al. [PPF+88]).

Jin and Wiberg's [JW90] improvement of Peraire's algorithm consists of eliminating the need for defining a background grid. Instead, the user can define control lines in the domain that contain the initial meshing parameters.

Despite its appealing features, the advancing front algorithm has so far only been applied to finite element computations in fluid dynamics and magnetic flow problems, but it has yet to find its way into the graphics community. We will now introduce our algorithm that applies the experience gained with advancing front algorithms in these areas to Free Form Deformation.

6.3 Implementation

Our implementation is based on Jin and Wiberg's version of the advancing front algorithm. The input to our FFD modelling system consists of polygonally bounded objects. As a first step, the solid is decomposed into its polygonal facets so that the mesher can process each facet individually.

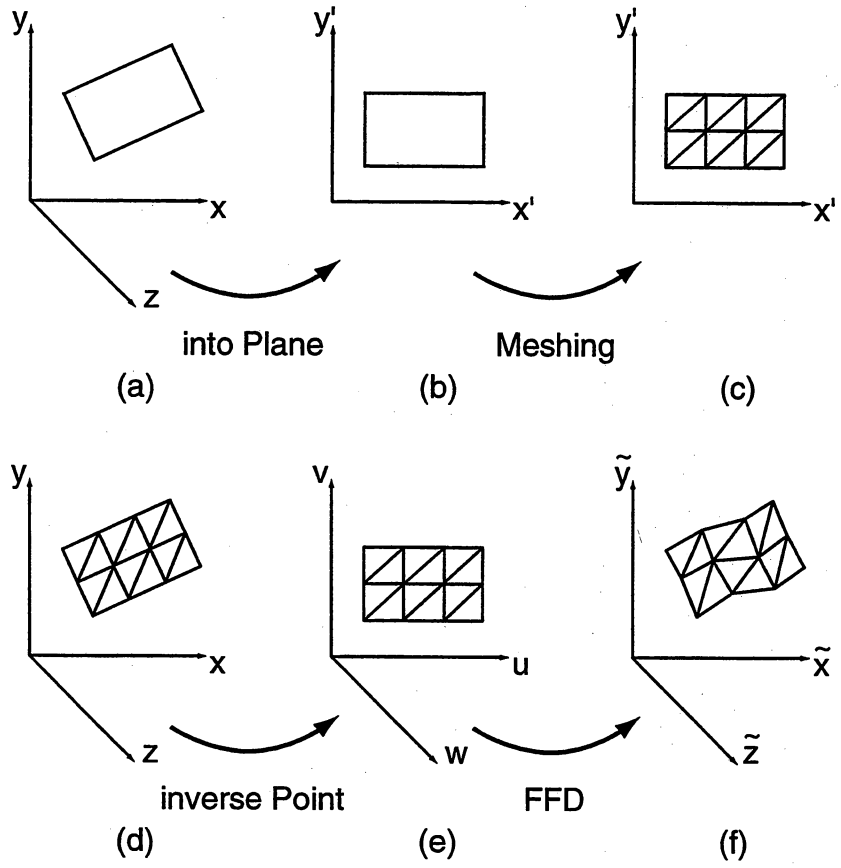


Figure 6.1: Domain in world space (a), transformation into plane (b), meshing (c), retransformation into world space (d), transformation into local solid space (e), FFD (f)

The new FFD process is shown in Figure 6.1. As we want to employ a two dimensional meshing scheme, we first have to transform each polygon into a 2D coordinate system (Figure 6.1 (b)). Care has to be taken to make sure that the outside edges of the polygon in the plane are defined in a counterclockwise manner, this enables the mesh generator to determine the inside and outside of the polygon. We then apply the meshing algorithm (Figure 6.1 (c)) in the plane. The next step consists of retransforming the polygon mesh back into world space (Figure 6.1 (d)). Now the mesh coordinates in local solid space have to be determined (inverse point problem, Figure 6.1 (e)). Finally, the actual deformation can be applied (Figure 6.1 (f)).

We will now describe the meshing process that happens once a polygon has been transformed into the 2D coordinate system.

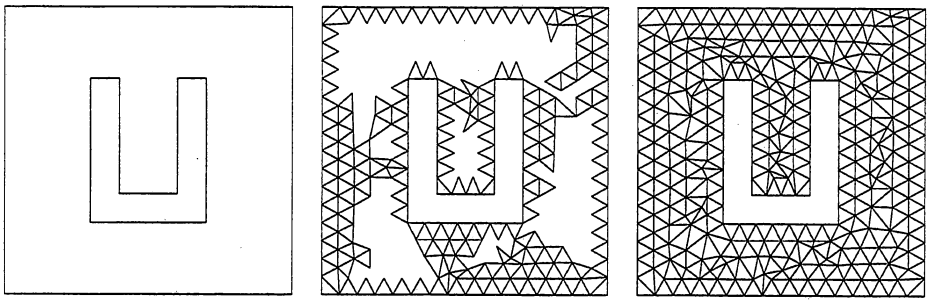
6.3.1 Advancing the Front

The main data structure of the advancing front algorithm is a heap (Cormen et al. [CLR90]) that contains the active front and consists of the edge segments on which triangles can be created. The edge segments are stored on the heap according to length. At each node in the heap, the length of the associated edge is shorter than or equal to the lengths of the edges associated with its children. The shortest edge is therefore always on the top of the heap.

Initially, the heap is empty. The meshing starts by taking all the domain edges and subdividing them according to some measure of curvature which will be discussed in Section 6.4.1. The resulting edge segments are then taken one by one and stored on the heap. Outside domain edges are stored in a counterclockwise manner and hole edges are stored in a clockwise manner, therefore the interior of the domain is always towards the left of edges.

When meshing a solid, each edge is shared by two surfaces. The question arises whether to subdivide all edges once on a per-solid basis or for each surface individually, which would result in all edges being meshed twice. In our current implementation, we mesh all edges first and then proceed to the individual surfaces. However, in a parallel implementation, where surfaces are distributed across processors, edges can also be subdivided on a per-surface basis. The edge subdivision criterion, discussed in Section 6.4.1, then ensures that the mesh is consistent across edges and no t-vertices are generated.

The process of generating a mesh is shown in Figure 6.2. The algorithm



(a) (b) (c)

Figure 6.2: Generation of a mesh for a planar domain

proceeds by extracting the shortest edge from the heap and constructing a triangle from it. The front advances by storing the newly created edges on the heap and deleting the edges that are now interior to the already meshed region. The process terminates when no edges are left on the heap. The algorithm can be summarised in C-style pseudocode:

```
while(edgeOnHeap) {
    extractEdgeFromHeap();
    generateTriangle();
    updateHeap();
}
```

The process of updating the front during the triangle generation is shown in Figure 6.3. The original domain consisted of vertices $1, 3, 5, 7$. After initial subdivision, edges 12, 23, 34, 45, 56, 67, 78, 81 were stored in the heap. In Figure 6.3 (a), we see the domain and the heap after two triangles 129 and 198 (dashed) have been created. The current front is shown as a solid line. Edge 89 is currently the shortest edge and therefore on top of heap. We extract edge 89 and construct the triangle 897. The result is shown in Figure 6.3 (b). Two changes have occurred: First, edges 78 and 89 are no longer part of the active front and have therefore been deleted. Second, edge 79, which is now part of the active front on which new triangles can be created, has been stored on the heap. The ordering of the edges in the heap reflects their length, so edge 34 on top is the shortest edge and will therefore be extracted next.

The two main operations performed on the heap are insertion and deletion of edges. Both can be performed in $O(\log n_{edges})$ time, where n_{edges} is the number of edge segments on the heap. Since a new triangle will typically

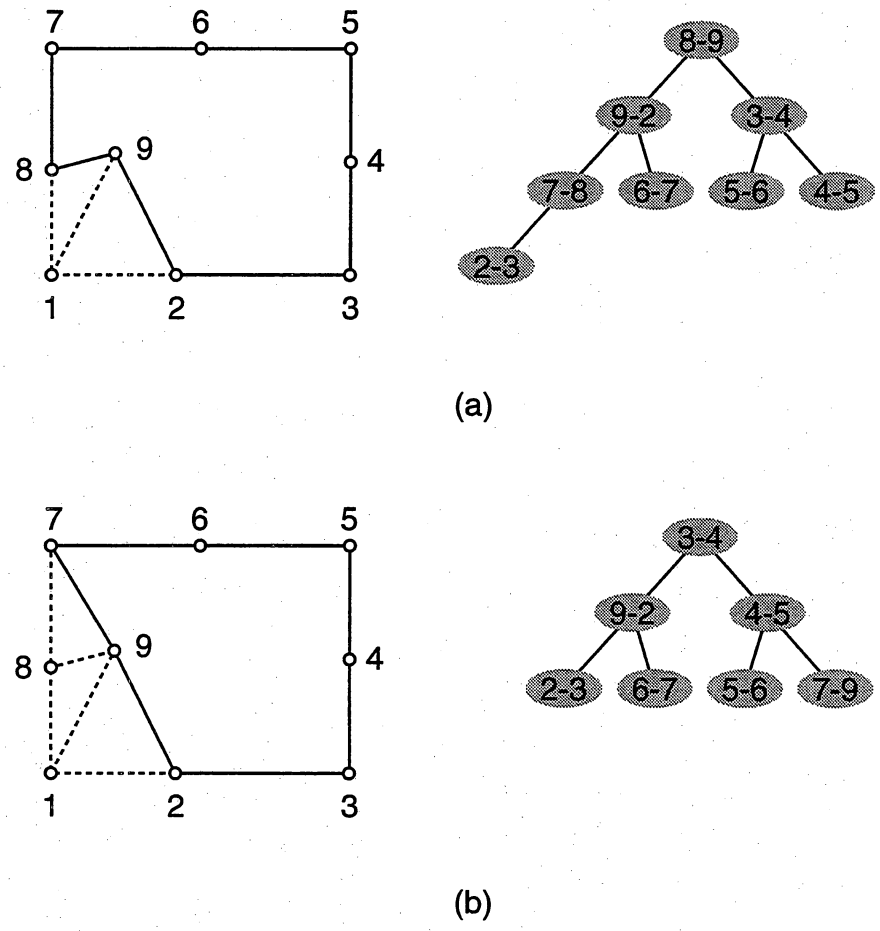


Figure 6.3: Generation of a new triangle in the domain (left) and updating the heap structure. The solid line indicates the current front.

yield two new edges, it is easy to see that this results in an $O(n \log n)$ time complexity of the meshing algorithm, where n is the number of generated triangles.

The rule always to build a triangle from the shortest edge of the heap is somewhat arbitrary, but works well in practice. However, if all edges are of similar size, the order of extraction does not seem to matter and we can also store edges in a first-in, first-out structure.

6.3.2 Generation of New Triangles

We will now discuss how to generate a new triangle on an edge **ab** of the current front. There are two ways of constructing a new triangle:

- creating a new node.
- using an existing vertex on the current front.

The creation of a new node is shown in Figure 6.4. Let h_d be the desired edgelength of the triangle to be constructed. The process of determining h_d will be discussed in Section 6.4.2. We first construct an isosceles triangle with edgelength h_d on edge **ab**. Let the resulting new node be **c**. This new node **c** is the first candidate for the new triangle. To collect other candidate nodes, we construct a node heap P of all existing nodes within a given distance of **c** (Figure 6.4). We have used a distance of $3h_{\max}$, where $h_{\max}$ is the length of the longest edge of the initially constructed isosceles triangle. Within the heap, the nodes are sorted according to distance to **c**, with the node closest to **c** on top.

Now we have to decide whether to use node **c** or an existing node, contained in the node heap P. Node **c** is rejected if

1. it is too close to an existing edge, i.e. its distance to any other edge on the heap is less than $0.4h_d$ (Figure 6.6 (a)).
2. there is an existing node in a certain, predefined region R_c around **c**.

The construction of region R_c is shown in Figure 6.5. The parameters have to be determined heuristically by numerical experiments. We resorted

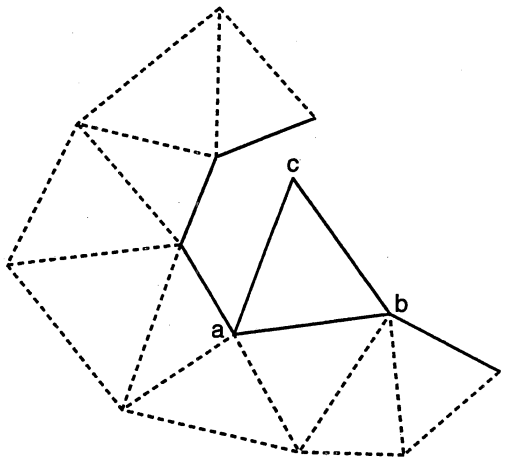


Figure 6.4: Construction of a candidate vertex **c** for a new triangle

to values used in the Jin and Wiberg implementation [JW90][1]:

$$\begin{aligned} r &= 1.1\sqrt{l_1^2 + l_2^2} \\ \beta_1 &= 15° + 0.9\alpha \\ \beta_2 &= \min(170° - \alpha, 105°) \\ \beta &= \max(\beta_1, \beta_2) \end{aligned}$$

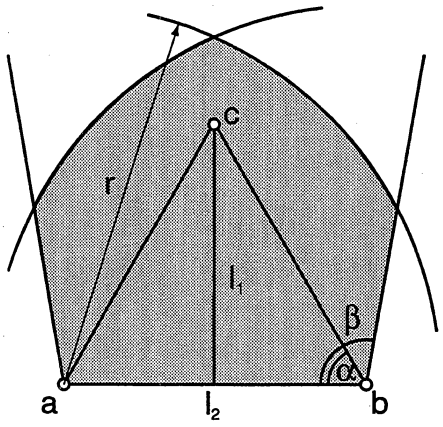


Figure 6.5: Region R_c (shaded) where existing nodes are preferable to **c**.

If there is a vertex in our favourite region R_c (condition 2), we choose this vertex. Otherwise, we check the new node **c**. If **c** is too close to an existing edge (condition 1), we choose the vertex $\mathbf{p} \in P$ closest to **c**.

[1] We found that varying the parameters slightly did not change the shape of the generated mesh noticably.

Let **e** be the chosen vertex. We now have to make sure that the newly generated triangle **abe** would result in a valid mesh element. This is done by checking the following conditions:

3. no existing node is closer than $0.3h_{\min}$ to the newly generated edges **ae** and **be**, where $h_{\min}$ is the length of the shortest edge in the new triangle (Figure 6.6 (b)).

4. the two new edges **ae** and **be** do not cross or overlap any existing edges on the active front (Figure 6.6 (c)).

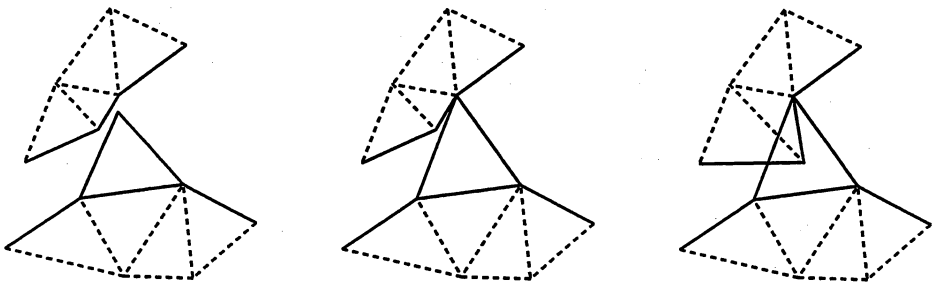


Figure 6.6: Rejection of a node: too close to existing edge (a), existing node too close to newly created edge (b), existing edge intersects new edge (c)

If our candidate node **e** fails to comply with any of these conditions, it is discarded and replaced by the next node in our favourite region. If there are no vertices left in this region, we choose the next $\mathbf{p} \in P$ from the node heap.

Checking conditions 3 and 4 are $O(n)$ operations. Since they have do be performed for each new triangle, they result in a $O(n^2)$ overall complexity. Löhner [Löh88] discusses how to use quadtree-based data structures to reduce this to $O(n \log n)$. In practice, we did not find this necessary, as both conditions can be checked very efficiently.

We give an implementation of this algorithm, written in C-style pseudocode:

```
generateTriangle(edgeSegment)
{
    isoNode = constructIsoscelesTriangle(edgeSegment);
    vicinityHeap = makeHeap(isoNode);

    /* first try nodes in region */

    while (node = extractHeap(vicinityHeap))
        if (inRegion(node) && nodeAcceptable(node))
            return node;

    /* else see if the isosceles triangle is acceptable */

    if(acceptable(isoNode))
        return isoNode;

    /* else take any acceptable node */

    restore(vicinityHeap);

    while (node = extractHeap(vicinityHeap))
        if (nodeAcceptable(node))
            return node;

    /* fail */

    return NO_NODE_FOUND;
}
```

where `nodeAcceptable(node)` checks if `node` complies with conditions (3) and (4).

We can see that the procedure is not guaranteed to find an acceptable node and might fail. This is most likely to occur if two fronts meet and it is one of the main problems of the nature of the advancing front algorithm. Careful selection of the heuristic meshing parameters should ensure that this situation does not occur. If the algorithm should fail to identify an acceptable node, we have to backtrack a number of steps and extract a different edge

from the heap. In practice, we did not encounter any situations where the algorithm was unable to find a suitable node and generate a valid mesh.

6.4 Adaptive Meshing According to Curvature

In order to adapt the mesh to the local curvature of the surface, we have to adapt the size of the generated triangles. For our advancing front algorithm, there are two ways of influencing the size of the triangles:

1. splitting the polygon edges into smaller segments.
2. varying the desired triangle size h_d.

Determining the optimal size for the edges and triangles requires the definition of local curvature parameters for edges and surfaces. We first introduce a measure for the local curvature of edges and discuss how to split the polygon boundaries according to this measure. We then introduce a measure for the local curvature of a surface that helps us determine values for the desired edge length parameter h_d.

6.4.1 A Curvature Measure for Edges

The discretisation process starts with the adaptive subdivision of the domain edges. Edges are recursively subdivided until the resulting segments are sufficiently close to straight lines. The original curve is then replaced by the straight line approximation. The curvature of a curve segment is determined by evaluating the tangent vectors at its start and endpoint in world space (Figure 6.7 (a)) and transforming them into deformed space (Figure 6.7 (b)). Subdivision stops if the angle between the tangent vectors at the start and endpoints of a curve segment in deformed space is sufficiently small. The angle between two tangent vectors $\tilde{\mathbf{t}}_i$ and $\tilde{\mathbf{t}}_{i+1}$ is given by

$$\cos\alpha = \frac{\tilde{\mathbf{t}}_i \cdot \tilde{\mathbf{t}}_{i+1}}{|\tilde{\mathbf{t}}_i||\tilde{\mathbf{t}}_{i+1}|}.$$

For sufficiently small angles and normalised tangent vectors, we can set $\cos\alpha \approx 1 - \alpha^2/2$. Setting $\varepsilon^2 = \alpha^2/2$, we obtain

$$\varepsilon^2 = 1 - (\tilde{\mathbf{t}}_i \cdot \tilde{\mathbf{t}}_{i+1}).$$

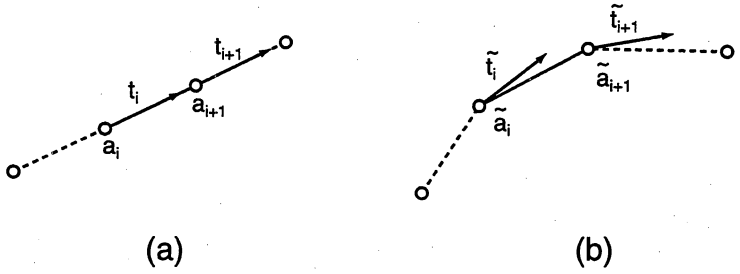


Figure 6.7: Subdivision of edge according to tangent vectors: world space (a) and deformed space (b)

The direction of the tangent vector in world space coincides with the direction of the edge. The tangent vectors can be transformed to deformed space using the formulae given in Section 4.5, this involves two matrix multiplications. Note that the length of the tangent vectors is not important, as our curvature estimate uses normalised tangent vectors.

It is important to sample the curve tangents initially at a sufficient number of points in order to avoid sampling errors such as illustrated in Figure 6.8. We therefore initially subdivide each edge such that each segment is shorter than some user-defined maximum edge length $h_{\max}$.

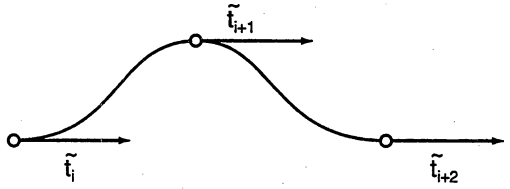


Figure 6.8: Undetected curvature due to coarse sampling

6.4.2 A Curvature Measure for Triangles

Once the edges have been subdivided according to tangent vectors, we can start the advancing front triangulation process. We now want to generate triangles with size according to the local curvature of the surface to be triangulated. The meshing parameter that controls the size of the generated elements is h_d, the desired triangle size. Therefore, we have to adapt h_d according to some curvature measure.

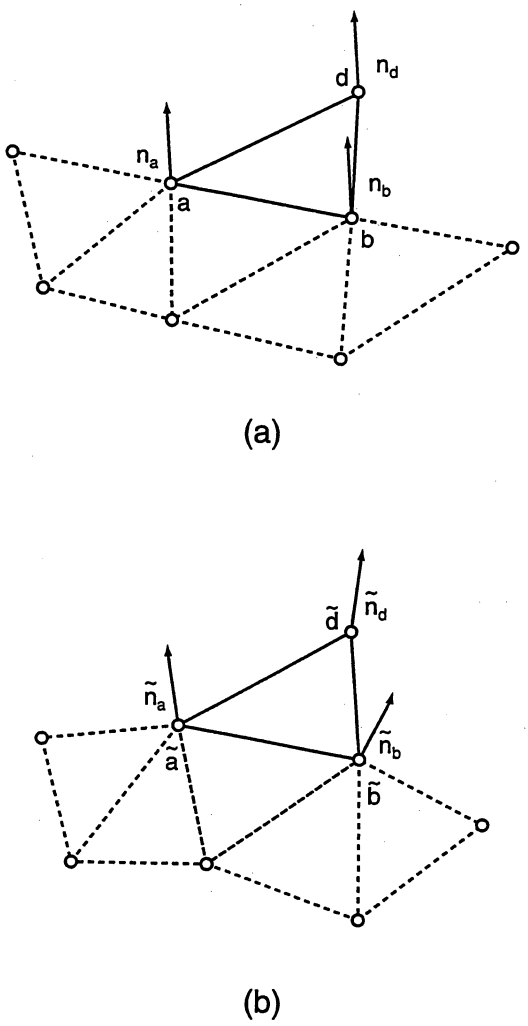


Figure 6.9: Determining the curvature of a triangle: normal vectors in world space (a) and deformed space (b)

Figure 6.9 shows the process of determining the local curvature. Let **ab** be the edge on which we want to generate the next triangle. We start by constructing a vertex **d** that results in an equilateral reference triangle **abd**[1]. To determine the curvature of this reference triangle, we evaluate the normals at $\tilde{\mathbf{a}}$,$\tilde{\mathbf{b}}$ and $\tilde{\mathbf{d}}$ in deformed space.

For a planar domain, all normals in world space are collinear (figure 6.9(a)). The transformation of normal vectors from world space to deformed space is

[1]Note that the reference vertex **d** is only used to determine local curvature and is usually different from vertex **c** used in Section 6.3.2

described in Section 4.5 and only involves two matrix multiplications. Let $\tilde{\mathbf{n}}_\mathbf{a}$,$\tilde{\mathbf{n}}_\mathbf{b}$ and $\tilde{\mathbf{n}}_\mathbf{d}$ be the normal vectors in deformed space (Figure 6.9(b)).

The angle between two normal vectors $\tilde{\mathbf{n}}_1$ and $\tilde{\mathbf{n}}_2$ is given by

$$\cos\alpha = \frac{\tilde{\mathbf{n}}_1 \cdot \tilde{\mathbf{n}}_2}{|\tilde{\mathbf{n}}_1||\tilde{\mathbf{n}}_2|},$$

which, for sufficiently small angles and normalised normal vectors, can be approximated by

$$\varepsilon^2 = 1 - (\tilde{\mathbf{n}}_1 \cdot \tilde{\mathbf{n}}_2).$$

We could now define a curvature measure by computing the normals $\mathbf{n}_\mathbf{a}$,$\mathbf{n}_\mathbf{b}$ and $\mathbf{n}_\mathbf{d}$, transforming them to deformed space and determining the largest angle between any pair of them.

In practice, we find that for short edges **ab**, this method of triangle curvature estimation leads to inconsistent curvature estimates and poorly graded meshes. We therefore replace the triangle **abd** by a larger equilateral triangle **efg**, with the same centre as **abd** but with edgelength $h_{\max}$, the user-defined maximum edge length (Figure 6.10). We can now compute the normals $\tilde{\mathbf{n}}_\mathbf{e}$, $\tilde{\mathbf{n}}_\mathbf{f}$ and $\tilde{\mathbf{n}}_\mathbf{g}$ in deformed space and use the largest angle between any pair of normals as a new curvature measure. Let this largest angle be $\varepsilon_\mathbf{n}$. Increasing the size of the reference triangle has an effect similar to applying a low-pass filter to the curvature estimates, which results in a smoother mesh.

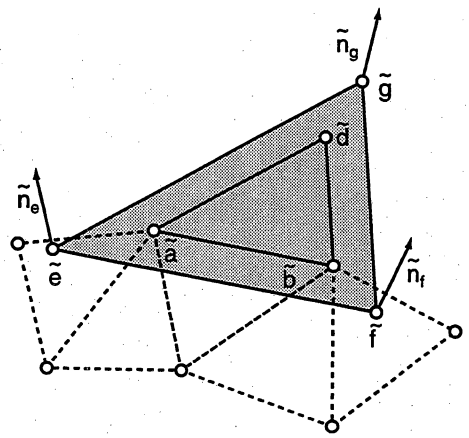


Figure 6.10: Modified curvature measure

Determining the triangle curvature according to the vertex normals only is not always sufficient. Figure 6.11 shows a domain that has been deformed

in a plane into a "u-shape". Although the edge curvature measure would detect the deformation of the rectangle edges, the normal vectors to any interior triangle would stay parallel and our curvature measure would therefore detect no curvature. The interior of such a domain would not receive finer subdivision and the mesh of the deformed domain would contain badly distorted triangles which cause rendering artefacts (see Figure 7.7 in Chapter 7).

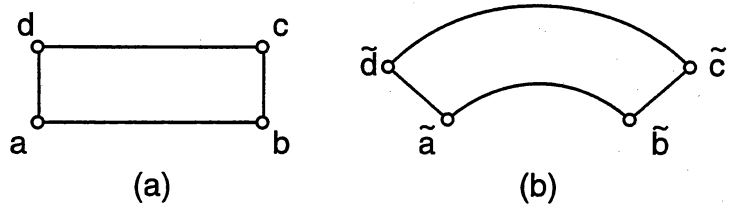


Figure 6.11: Planar deformation of a rectangular domain

Planar distortions can be detected by computing the tangent vectors along triangle edges. Figure 6.12 shows the tangent vectors at the start and endpoints of edges $\tilde{\mathbf{e}}\tilde{\mathbf{g}}$ and $\tilde{\mathbf{f}}\tilde{\mathbf{g}}$. We use the edge curvature measures for domain edges, given in Section 6.4.1, to compute a curvature term for each pair of tangent vectors along an edge. Let $\varepsilon_{\mathbf{t}} = \max(\varepsilon_{\mathbf{t_{ef}}}, \varepsilon_{\mathbf{t_{eg}}})$ be the largest angle between the tangent vectors of edge $\tilde{\mathbf{e}}\tilde{\mathbf{g}}$ and $\tilde{\mathbf{f}}\tilde{\mathbf{g}}$.

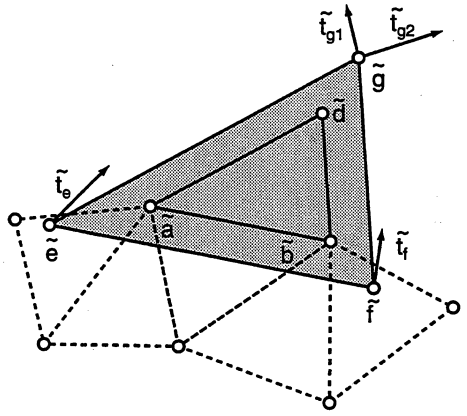


Figure 6.12: Tangent vectors as a curvature measure

6.4.3 Determining the Triangle Size

The new vertex **d** has only been generated for the purpose of determining the curvature and will not be part of the newly generated triangle. The new triangle will be generated using the algorithm given in Section 6.3.2, which generates a new triangle depending on the desired triangle size h_d. We will now show how to determine the triangle size measure h_d.

We have determined two curvature parameters, $\varepsilon_{\mathbf{n}}$ and $\varepsilon_{\mathbf{t}}$, one based on the normal vectors and one based on the tangent vectors. Our objective is to construct a triangle such that no adjacent normal or tangent vectors differ by more than some predefined measure $\varepsilon_{\max}$. Let $\varepsilon_{\text{curv}} = \max(\varepsilon_{\mathbf{n}}, \varepsilon_{\mathbf{t}})$ be a measure of the triangle curvature. We then define the desired triangle size as

$$h_d = \begin{cases} h_{\max} & \text{if } \varepsilon_{\text{curv}} < \varepsilon_{\max} \\ h_{\max}\frac{\varepsilon_{\max}}{\varepsilon_{\text{curv}}} & \text{otherwise} \end{cases}, \tag{6.1}$$

where $\varepsilon_{\max}$ is a measure of the maximum acceptable triangle curvature.

The expression for h_d is based on the assumption that if the angle between two normal or tangent vectors of the reference triangle is n times the acceptable angle $\varepsilon_{\max}$, generating a triangle with edgelength $1/n$th of the original triangle will yield an acceptable triangle. Note, however, that it is not guaranteed that all tangent and normal vectors of the resulting new triangle will conform to the error measure $\varepsilon_{\max}$.

6.5 Summary

The advancing front mesh generation scheme is well established in finite element computations, but has not yet made its way into computer graphics.

We have developed a graphics-oriented version of the algorithm for objects bounded by planar polygons. We have addressed two problems: how to generate triangle meshes and how to adapt the mesh size to the curvature of the objects to be meshed.

The mesh generation consists of first composing the solid into its polygons. Each polygon is meshed individually by first subdividing its edges according to curvature and storing them on a heap. We then take edge segments of the heap and construct triangles on them until the whole region is meshed and no edges remain on the heap. Care has to be taken to ensure mesh

compatibility across shared edges. The size of the generated triangles is determined according to a set of curvature measures. We have developed measures that detect the curvature of triangles as well as planar distortions.

In the next chapter, we show results of the application of our algorithm to a number of Free Form Deformation problems.

Chapter 7

Results

7.1 Introduction

We now show results of the application of our advancing front algorithm to a number of deformation problems.

The FFD algorithm and the advancing front mesh generator have been implemented using the C programming language [KR88]. The program has been developed on a SUN Sparc 10 workstation and later ported to a Silicon Graphics Indy workstation running version V.4 of the IRIX operating system. The meshes shown in this section were drawn using PostScript [Ado85]. The timings shows are for the Silicon Graphics Indy workstation[1] .

Shaded images were raytraced using rayshade (Kolb [Kol91]). Colour images of the deformed objects can be found in Appendix A. These colour plates were generated using Adobe Photoshop [Ado91] and printed as 300dpi 24 bit colour images on a Mitsubishi S3600-30D dye sublimation printer.

The images were Phong-shaded (Bui-Tuong [BT75]) with surface normals which were analytically determined using the procedures described in Section 4.5.

Unless otherwise stated, objects have been deformed by a deformation solid defined by a set of triparametric Bernstein polygons as in the Sederberg and Parry [SP86b] implementation of FFD.

[1] single R4600SC CPU running at 133 MHz

7.2 Deformation of a Rectangular Domain

We start with a rectangular polygon that has been deformed by a $7 \times 2 \times 2$ deformation lattice. The original polygon is shown in Figure 7.1 (a), Figure 7.1 (b) shows the deformed polygon.

We have meshed the polygon using a number of different error parameters $\varepsilon_{\max}$. Figure 7.3 shows the resulting meshes for $\varepsilon_{\max} = \infty$ (non-adaptive meshing), $\varepsilon_{\max} = 0.5$, $\varepsilon_{\max} = 0.2$ and $\varepsilon_{\max} = 0.05$. It demonstrates that the algorithm produces triangulations of variable size depending on the local curvature of the surface and how a variation of the error parameter $\varepsilon_{\max}$ affects the number of generated triangles. The relationship between the timings and numbers of generated triangles for a wider range of values of $\varepsilon_{\max}$ is shown in Table 7.1.

Figure 7.2 shows the number of generated triangles plotted against usage of CPU time. We notice a near-linear growth rate.

$\varepsilon_{\max}$	none[a]	0.50	0.20	0.10	0.05	0.02
triangles	36	54	322	1056	3680	19440
time[b]	0.19	0.29	2.30	10.40	60.37	826.83

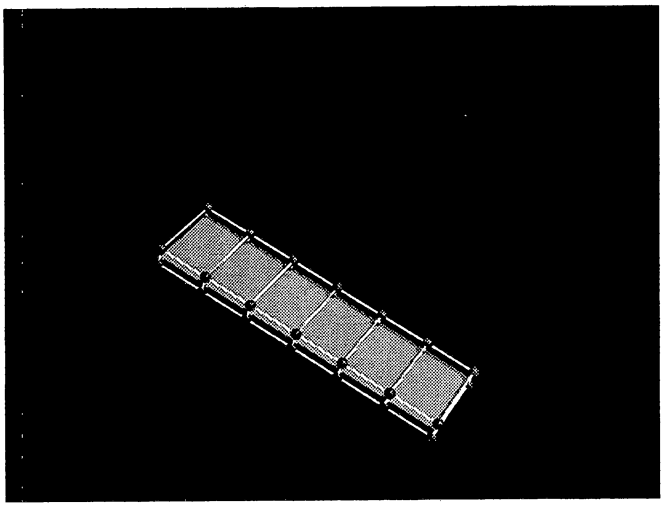
[a] non-adaptive meshing

[b] seconds CPU time on an Silicon Graphics Indy workstation, obtained by the UNIX clock() system call

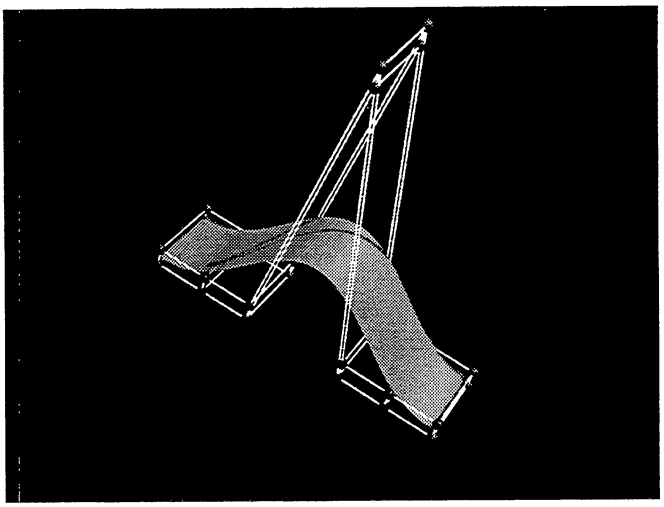
Table 7.1: Numbers of triangles and timings for different error measures.

In Appendix A, Figure A.1, we show texture-mapped images for $\varepsilon_{\max} = \infty$ (non-adaptive meshing), $\varepsilon_{\max} = 0.5$ and $\varepsilon_{\max} = 0.2$. The surface of all images appears fairly smooth. This is largely due to the analytically determined surface normals at the triangle mesh points. Difference in image quality can be seen in three areas:

- polygon edges
- sharp shadows
- texture patterns



(a)



(b)

Figure 7.1: Deformation of a rectangular domain

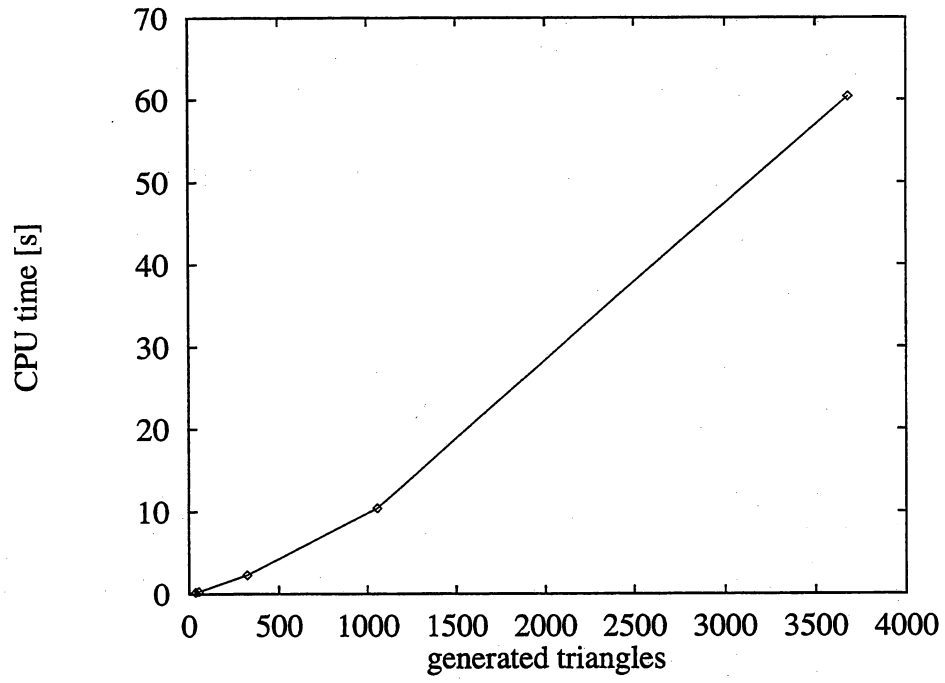
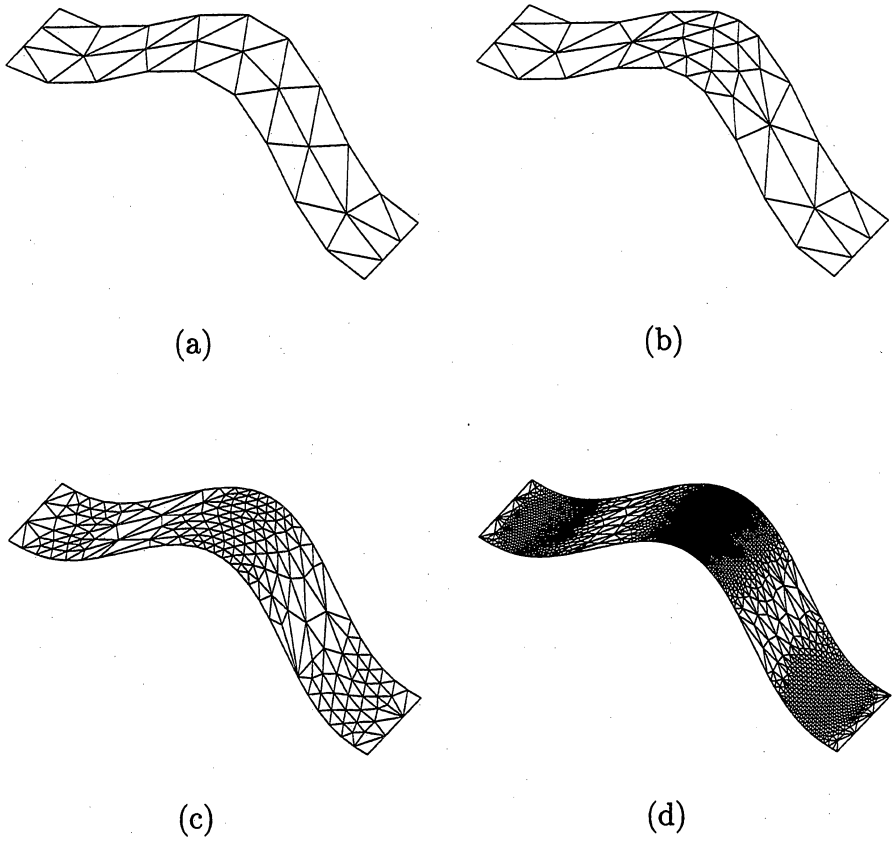


Figure 7.2: Mesh generation timings for rectangular domain

The non-adaptively meshed image shows visible artefacts along the boundary edges and across texture edges. Also, the shadows cast by the deformation grid appear segmented. We can see that $\varepsilon_{\max} = 0.5$ already results in a significantly improved image quality, but the artefacts across texture edges are still visible. An error measure of $\varepsilon_{\max} = 0.2$ (322 triangles) is sufficient to produce an image that is free of noticeable artefacts. Note that the mesh for this image took only 2.3 seconds to generate.



(a) (b)

(c) (d)

Figure 7.3: Resulting meshes for $\varepsilon_{\max} = \infty$, $\varepsilon_{\max} = 0.5$, $\varepsilon_{\max} = 0.2$ and $\varepsilon_{\max} = 0.05$

7.3 Deformation of a Solid

In our next example, we show the meshing of a solid bar with a concave hole (Figure 7.4). The top and bottom surface of this solid consist of only one polygon each, the hole is treated as a feature of these polygons. This example illustrates several points:

- mesh generation on the surfaces of a solid in boundary-representation
- meshing of domains with holes
- use of the tangent curvature measure (Section 6.4.2) to detect planar distortions.

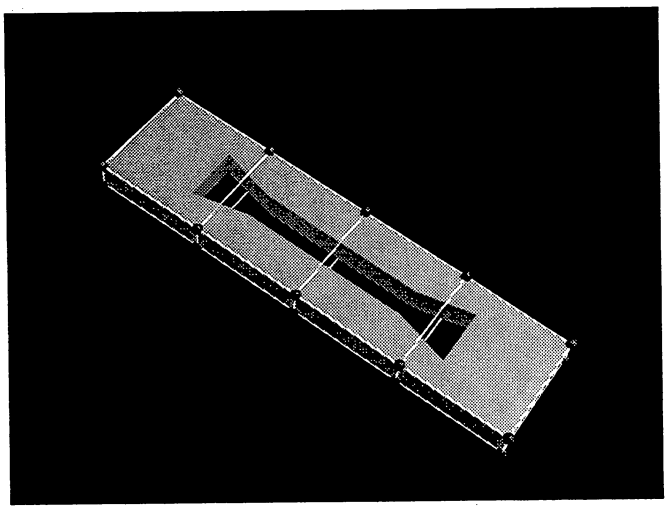


Figure 7.4: Solid with concave hole

Using a $5 \times 2 \times 2$ deformation grid, we have deformed the solid into a wave shape. The resulting mesh is shown in Figure 7.5, a shaded version can be seen in Figure A.2 in Appendix A. With the error measure set to $\varepsilon_{\max} = 0.1$, it took 7.26 seconds to generate and consists of 1252 triangles.

The meshing algorithm generates a smooth mesh and copes well with the hole in the top surface. The mesh is consistent across edges and does not generate t-vertices which would result in visible cracks. Note that the polygon along the visible long side of the solid has only undergone planar distortion. The tangent curvature measure detects this distortion and adapts the mesh accordingly.

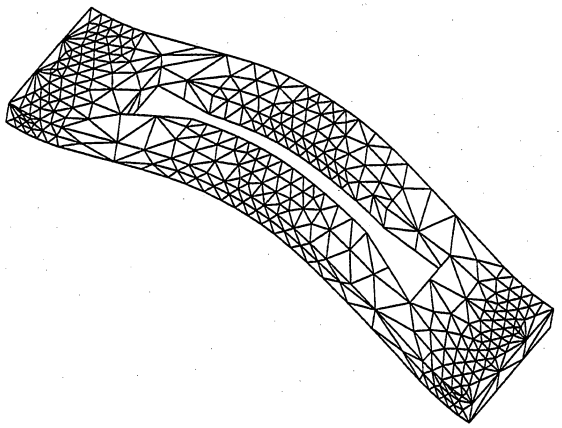


Figure 7.5: Meshed solid for $\varepsilon_{\max} = 0.1$, 1252 triangles

7.4 Planar Distortion of a Solid

In order to illustrate more clearly the planar distortion problem, we have applied a planar deformation to the solid of the previous example by bending it into a "u-shape". The resulting mesh, again with the error measure set to $\varepsilon_{\max} = 0.1$, is shown in Figure 7.6. It consists of 1606 triangles and took 9.38 seconds to generate. Figure A.3 contains the shaded image, which also shows the deformed lattice.

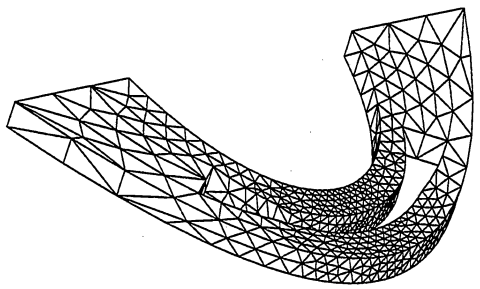


Figure 7.6: Meshed solid for $\varepsilon_{\max} = 0.1$, 1606 triangles

The top surface has undergone planar distortion and it can be seen that the meshing algorithm detects this distortion and adapts the triangle size accordingly.

It can be argued that a fine subdivision of surfaces that have undergone only planar distortion is not necessary. We have therefore meshed the same surface without the triangle tangent subdivision criterion. The resulting mesh, again for $\varepsilon_{\max}$, is shown in Figure 7.7. It consists of 884 triangles and took 3.37 seconds to generate, which represents a considerable improvement. However, we can see that the top surface now contains a high number of slivers, i.e. triangles that contain small interior angles. As discussed in Section 5.3.1, slivers are undesirable due to the rendering artefacts they cause.

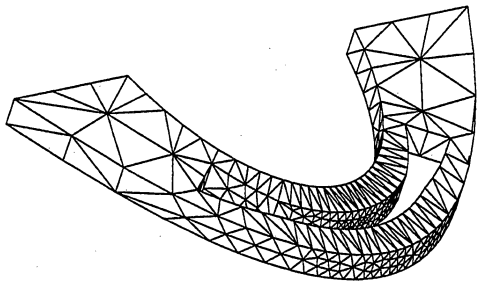


Figure 7.7: Meshed solid without planar distortion detection. $\varepsilon_{\max} = 0.1$, 884 triangles

7.5 Deformation of a Tile into a Bell-Shape

So far, our examples consisted of deformations that resulted in curvature in only one direction. We will now deform a tile-shaped solid into a "bell-shape", which results in curvature along different directions. The tile, which contains a star-shaped hole, is shown in Figure 7.8. The deformation grid consists of $7 \times 7 \times 2$ vertices. We obtained the deformation by moving the two central control points to positions above the tile, see Figure A.4 for a shaded image.

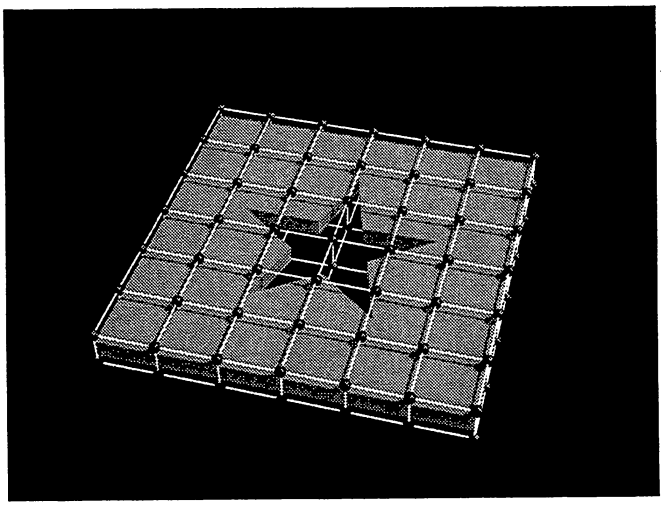


Figure 7.8: Tile with star-shaped hole

We have meshed the deformed tile using error measures of $\varepsilon_{\max} = 0.2$ and 0.1, which resulted in 2920 and 9230 triangles. The mesh for $\varepsilon_{\max} = 0.2$ is shown in Figure 7.9. Zones of increased triangle density can be seen along the rim of the bell and towards the star-shaped hole in the centre.

In Figure 7.10, we show the mesh of a tile deformed by the same deformation grid, this time with a finer error tolerance of $\varepsilon_{\max} = .1$. The circular region of high mesh density is now more pronounced, we can also see that the mesh density increases towards the centre of the tile.

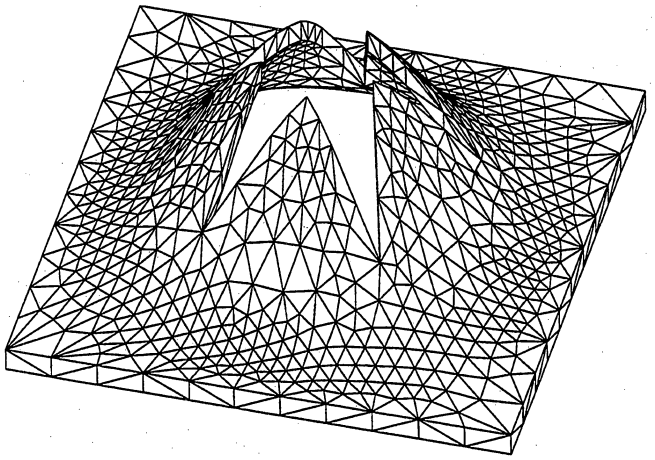


Figure 7.9: Mesh of bell-shaped tile. $\varepsilon_{\max} = 0.2$, 2920 triangles

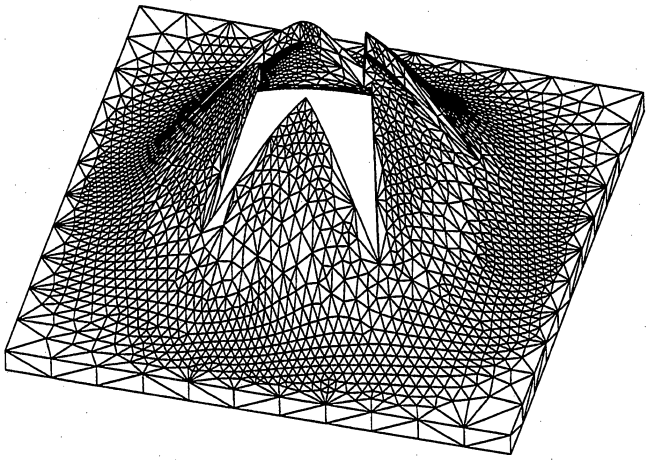


Figure 7.10: Mesh of bell-shaped tile. $\varepsilon_{\max} = 0.1$, 9230 triangles

7.6 Deformation of a Complex Solid

We now show the meshing process of a deformed girder. A shaded image of the undeformed girder is shown in Figure 7.11. It consists of 26 polygons, two of which are concave and two have three holes each assigned to them.

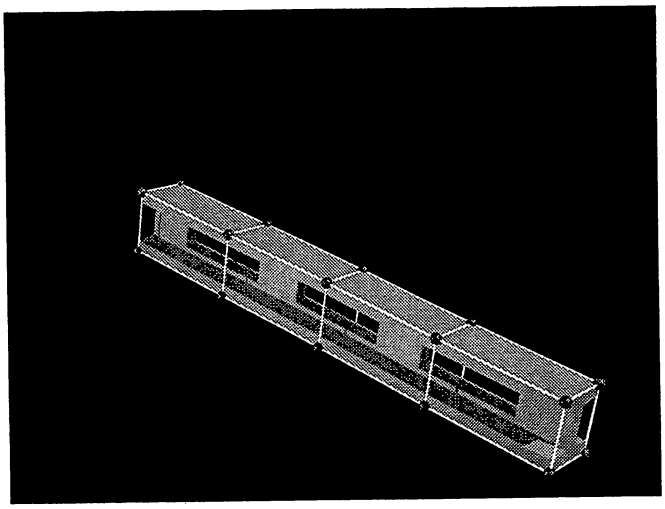


Figure 7.11: Girder

Using a $5 \times 2 \times 2$ grid, we have deformed the girder into a shape similar to the solid in Section 7.3. Figure A.5 shows a shaded image. Figure 7.12 shows the meshed solid for $\varepsilon_{\max} = 0.1$. It took 6.20 seconds to generate and consists of 1402 triangles. We have then decreased the error measure to $\varepsilon_{\max} = 0.05$. The resulting mesh, consisting of 2668 triangles, is shown in Figure 7.13 and took 13.38 seconds to generate. The results are summarised in Table 7.2. Again, we observe an almost linear relationship between generation time and number of triangles.

$\varepsilon_{\max}$	none[a]	0.1	0.05
triangles	840	1402	2668
time	3.56	6.20	13.38

[a] non-adaptive meshing

Table 7.2: Numbers of triangles and timings for different error measures.

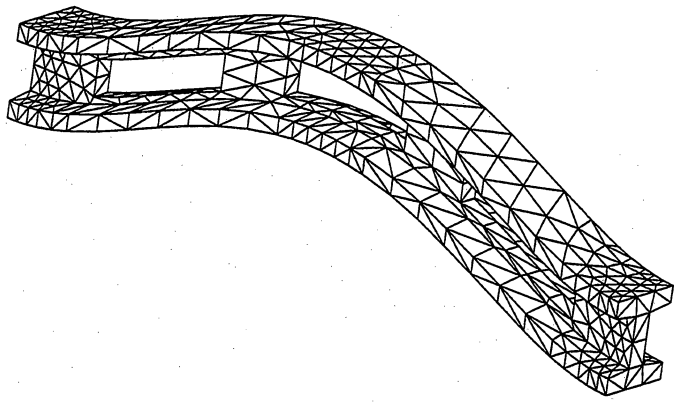


Figure 7.12: Mesh of deformed girder. $\varepsilon_{\max} = 0.1$, 1402 triangles

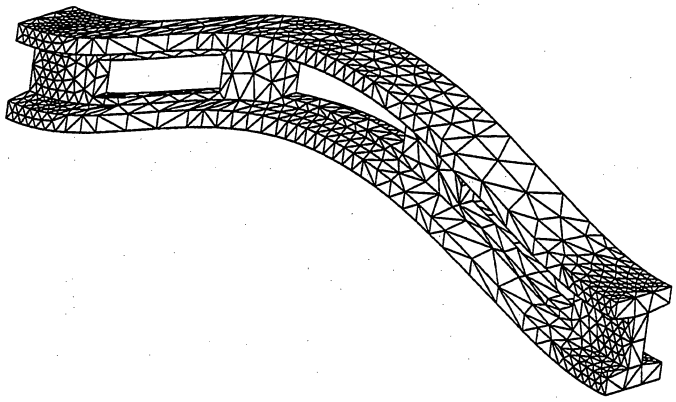


Figure 7.13: Mesh of deformed girder. $\varepsilon_{\max} = 0.05$, 2668 triangles

7.7 Deformation by a B-spline volume

One advantage of our advancing front method is that it can be used to render deformed objects where analytic expressions of the deformed surfaces can not be found. It only relies on points, tangents and normals being transformed from world to deformed space.

To illustrate this point, we show the meshing process of the same girder deformed by a cubic B-spline volume with irregularly spaced control points and a non-uniform knot vector. When applying our meshing scheme to deformations by B-spline volumes, attention has to be paid to two points:

- The B-spline volume has to be at least C^1 continuous for tangents and normals to be transformed into deformed space.
- B-spline deformations can result in highly local deformations, so the initial sampling density has to be fine enough to capture these deformations.

The initial grid is shown in Figure 7.14.

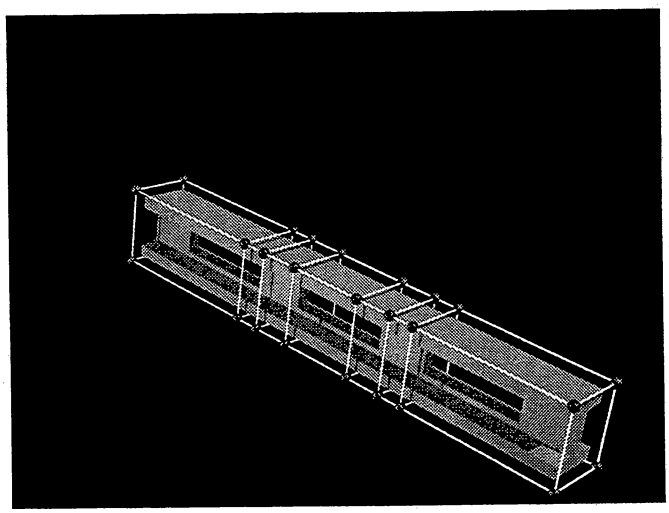


Figure 7.14: Girder

The deformation has been applied by displacing the 8 control points in the centre, see Figure A.6 for a shaded image. Note that the resulting curvature is significantly higher than in the Bézier volume deformed girder. The resulting mesh for $\varepsilon = 0.1$ is shown in Figure 7.15. We can see that the meshing algorithm adapts to the higher curvature and produces a well graded mesh.

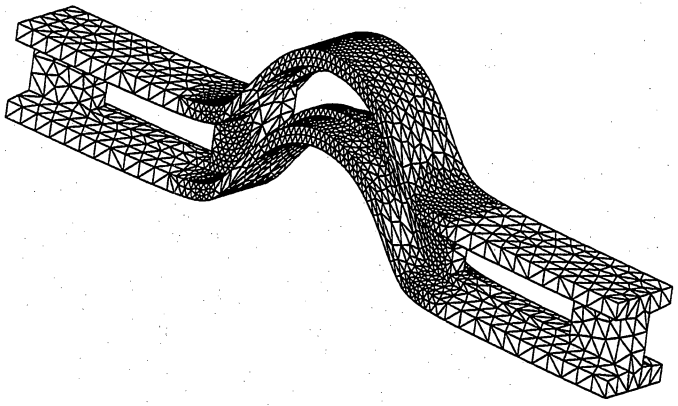


Figure 7.15: Mesh of B-spline deformed girder. $\varepsilon_{\max} = 0.1$, 6378 triangles.

7.8 Summary

We have implemented the FFD algorithm and our advancing front mesher on a UNIX workstation and applied them to a number of deformation problems. The meshed solids were of different complexity, from a simple single polygon to complex solids containing a number of concave surfaces and holes. The deformation solids were defined in terms of Bézier volumes and non-uniform B-splines volumes.

The density of the generated meshes is well adapted to the curvature of the surfaces and the triangles are generally well shaped. The algorithm prevents cracks in deformed objects by generating consistent meshes without t-vertices across shared edges.

The solids have been meshed using varying error measures $\varepsilon_{\max}$, resulting in meshes of different density. The relationship between the number of generated triangles and the running time of the mesher is almost linear. For most of the solids, the algorithm generated a mesh that resulted in a smooth shaded image of high quality in about 3-5 seconds.

Using the heuristic parameters given in Section 6.3.2, we found that the meshing algorithm was very robust. We did not encounter any situations where the algorithm got stuck and failed to produce a mesh.

Chapter 8

Summary & Conclusions

We conclude this thesis with a summary of our major results and some suggestions for further work.

8.1 Results

We have provided a discussion of the rendering aspects of free-form deformation. We started by developing a consistent framework for Bézier curves, surfaces and hypersurfaces in blossoming notation. The blossoming notation, introduced by Ramshaw, is a recent development and its extension to tensor product surfaces and hypersurfaces has not received much attention.

The first contribution of this dissertation consisted of providing geometrically intuitive algorithms for the composition of Bézier curves and surfaces with tensor product Bézier hypersurfaces.

We have then shown that these composition algorithms can be applied to Sederberg's FFD technique. The deformed object surfaces can be expressed in Bézier form, if the object has been deformed by a Bézier hyperpatch with a grid of initially regularly spaced control points.

The FFD scheme can not only be used to obtain object points in deformed space. Obtaining correct surface normals is important to determine information about surface curvature for subdivision and shading purposes. Normal transformation has not been mentioned in the existing FFD literature. We have shown that, using Jacobian matrices, we can analytically determine deformed surface normals.

The practical applicability of the functional composition algorithms to rendering deformed objects is limited. Rendering the deformed surfaces directly from their analytic description is either not possible, because closed form expressions for the surfaces can only be obtained for parallelepipedical deformation grids, or prohibitively expensive, because the resulting surfaces are of very high order.

An alternative approach consists of tessellating the surfaces into triangular meshes that approximate the surface closely and applying the deformation to these meshes. Using dedicated rendering hardware, the triangle meshes can then be rendered very efficiently. We have surveyed finite element meshing algorithms and found that the advancing front algorithm possesses a number of properties that make it a suitable candidate for meshing FFD deformed objects.

Our second major contribution consisted of developing such an advancing front meshing algorithm for meshing FFD deformed models. Deformed surfaces are subdivided according to local curvature to a user-specified accuracy. The generated meshes are consistent across shared surface edges and therefore do not cause visible cracks. Our meshing scheme is superior to the existing two techniques by Parry and Griessmair in that it can cope with any polygonally bounded object rather than only spheres, cylinders and blocks (Parry) or convex polygons (Griessmair).

Using a number of examples, we have shown that our implementation meshes complex objects and produces well graded meshes. We did not encounter any convergence problems. Most objects can be meshed in an adequate density in about 3 seconds and the growth rate between time and number of generated triangles seems near linear.

8.2 Future Work

We can see possible extensions to this work in five main areas:

- extension of the composition algorithms to B-splines
- extension of meshable surfaces to NURBS
- parallel implementation

- view dependent meshing
- alternative meshing algorithms.

In Chapter 3, we have briefly mentioned the extension of our blossoming composition algorithms to B-splines and rational curves. B-splines can be decomposed into Bézier curves. We have therefore argued that we can apply our composition algorithms by decomposing the B-splines into Béziers, composing the individual Béziers and rejoining the composed Béziers into B-splines. However, it should be possible to derive a composition algorithm that can be directly applied to the B-spline control points, thus relieving us from the decomposition and rejoining process. Also, we have mentioned the problems that arise when composing surfaces with B-spline volumes. Further investigation of this type of composition problem is needed.

As for our meshing algorithm, our current implementation is restricted to meshing solids defined by planar surface patches. However, in many CAD applications, the objects are described in terms of trimmed non-uniform rational B-spline (NURBS) surfaces, see Versprille [Ver75] or Piegl [Pie91]. Extending our meshing scheme to handle this type of surface type would be possible by applying our meshing algorithm in the parameter space of the NURBS patch. However, finding a curvature measure for the deformed edges and surfaces is more complex now, because both the mapping from parameter space to the undeformed patch and from the undeformed patch to the deformed patch have to be considered.

Our meshing scheme meshes each surface individually whilst ensuring mesh compatibility across adjacent surfaces. A parallel implementation, with individual surfaces distributed across processors, would therefore be straightforward. Our current single processor implementation meshes objects in about 3 seconds on a bottom-of-the-range, single processor Silicon Graphics Indy workstation. A parallel implementation on a MIMD architecture should achieve these results in well under a second.

The ability to obtain meshes at near interactive rates would make meshing dependent on the current viewpoint an option. Our shaded images of deformed objects (Appendix A) show that meshing artefacts can most easily be picked up along surface boundaries and silhouette lines. Given a view vector, we could thus integrate a silhouette measure, based on the view vector and the surface normal vector, into our subdivision criterion and achieve higher subdivision along silhouette lines.

Finally, using the same subdivision criteria, the use of alternative meshing algorithms should be investigated. Although we did not encounter any convergence problems for our advancing front scheme, algorithms such as the Guibas and Stolfi version of the Constrained Delaunay Triangulation (Guibas and Stolfi [GS85]) seem to be more robust (Lischinski [Lis94]). Any floating point implementation of a geometric algorithm has to rely on heuristics to resolve degenerate cases and numerical problems due to finite precision. However, due to the simplicity of the Delaunay triangulation, this seems to be less of a problem than for geometrically more complex algorithms such as the advancing front scheme.

Bibliography

[Ado85] Adobe Systems Incorporated, Hill Place House, London, UK. *PostScript Language: Reference Manual*, 1985.

[Ado91] Adobe Systems Incorporated, Hill Place House, London, UK. *Adobe Photoshop: User Guide*, 1991.

[AES91] S. S. Abi-Ezzi and L. A. Shirman. Tesselation of curved surfaces under highly varying transformations. In *Proceedings of Eurographics ' 91*, pages 385–397. North Holland, 1991.

[Bar84] Alan H. Barr. Global and local deformations of solid primitives. *Computer Graphics*, 18(3):21–30, July 1984. SIGGRAPH '84 Conference Proceedings.

[Bar85] R. E. Barnhill. Surfaces in computer aided geometric design: A survey with new results. *Computer Aided Geometric Design*, 2:1–17, 1985.

[Bar91] Richard Bartels. Polar forms and splines. In *Topics in the Construction, Manipulation and Assessment of Spline Surfaces*, number C 25 in SIGGRAPH Course Notes, pages 2.0–2.12. ACM SIGGRAPH, 1991.

[Béz78] P. Bézier. General distortion of an ensemble of biparametric surfaces. *Computer Aided Design*, 10(2):116–120, March 1978.

[BFK84] W. Böhm, G. Farin, and J. Kahmann. A survey of curve and surface methods in CAGD. *Computer Aided Geometric Design*, 1(1):1–60, July 1984.

[BMSW91] Daniel Baum, Stephen Mann, Kevin Smith, and James Winget. Making radiosity usable: Automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. *Computer Graphics*, 25(4):51–60, July 1991. SIGGRAPH '91 Conference Proceedings.

[Bow81] A. Bowyer. Computing dirichlet tessellations. *Computer Journal*, 24(2):162–166, 1981.

[BRW89] Daniel Baum, Holly Rushmeier, and James Winget. Improving radiosity solutions through the use of analytically determined form-factors. *Computer Graphics*, 23(3):325–334, July 1989. SIGGRAPH '89 Conference Proceedings.

[BT75] Phong Bui-Tuong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[BWS+87] Peggy L. Baehmann, Scott L. Wittchen, Mark S. Shepard, Kurt R. Grice, and Mark A. Yerry. Robust, geometrically based, automatic two-dimensional mesh generation. *International Journal for Numerical Methods in Engineering*, 24:1043–1078, 1987.

[Cat74] E. Catmull. A subdivision algorithm for computer display of curved surfaces. Technical Report UTEC-CSc-74-133, Department of Computer Science, University of Utah, Salt Lake City, UT, December 1974.

[Cav74] J. C. Cavendish. Automatic triangulation for arbitrary planar domains for the finite element method. *International Journal for Numerical Methods in Engineering*, 8:679–696, 1974.

[CFF85] J. C. Cavendish, D. A. Field, and W. H. Frey. An approach to automatic three-dimensional finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 21:329–347, 1985.

[Che89] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR 89-983, Department of Computer Science, Cornell University, Ithaca, NY, April 1989.

[CJ91] Sabine Coquillart and Pierre Jancene. Animated free-form deformation: An interactive animation technique. *Computer Graphics*, 25(4):23–26, July 91. SIGGRAPH '91 Conference Proceedings.

[Cla79] J. H. Clark. A fast scan-line algorithm for rendering parametric surfaces. *Computer Graphics*, 13(2):174, August 1979. SIGGRAPH '79 Conference Proceedings.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT electrical engineering and computer science series. The MIT Press, Cambridge, MA, 1990.

[Coo67] Steven A. Coons. Surfaces for computer-aided design of space forms. Technical Report MAC-TR-41, Massachusetts Institute of Technology Project MAC, Cambridge, MA, 1967.

[Coq90] Sabine Coquillart. Extended free-form deformation: A sculpturing tool for 3D geometric modeling. *Computer Graphics*, 24(4):187–196, August 90. SIGGRAPH '90 Conference Proceedings.

[CW93] Micheal F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*, chapter 8. Academic Press, London, 1993.

[dB72] Carl de Boor. On calculating with B-splines. *Journal of Approximation Theory*, 6(1):60–62, 1972.

[DeR88] Tony DeRose. Composing bézier simplexes. *ACM Transactions on Graphics*, 7(3):198–221, July 1988.

[dFdC59] P. de Faget de Casteljau. Outillages méthodes calcul. Technical report, A. Citroën, Paris, France, 1959.

[dFdC86] P. de Faget de Casteljau. *Shape Mathematics and CAD.* Kogan Page, London, 1986.

[dFFP85] L. de Floriani, B. Falcidieno, and C. Pienovi. Delaunay-based representation of surfaces defined of arbritrarily shaped domains. *Computer Vision, Graphics and Image Processing*, 32:127–140, 1985.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, Berlin, 1987.

[Far83] G. Farin. Algorithms for rational bézier curves. *Computer Aided Design*, 15(2):73–77, March 1983.

[Far91] R. Farouki. On the stability of transformations between power and bernstein polynomial forms. *Computer Aided Geometric Design*, 8(1):29–36, 1991.

[Far93] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, London, third edition, 1993.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[Fre87] W. H. Frey. Selective refinement: A new strategy for automatic node placement in graded triangular meshes. *International Journal for Numerical Methods in Engineering*, 24:2183–2200, 1987.

[FvDFH90] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practise.* IBM Systems Programming Series. Addison Wesley, New York, NY, second edition, 1990.

[Geo91] P. L. George. *Automatic mesh generation.* John Wiley & Sons, Chichester, 1991.

[Gil72] James Ian Gill. *Computer-Aided Design of Shell Structures using the Finite Element Method.* PhD thesis, Cambridge University Computer Laboratory, 1972.

[Gou71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20(6):302–308, June 1971.

[GP89] Josef Griessmair and Werner Purgathofer. Deformation of solids with trivariate B-Splines. In W. Hansmann, F.R.A. Hopgood, and W. Strasser, editors, *EUROGRAPHICS '89*, pages 137–148. Eurographics Association, Elsevier Science Publishers B.V. (North-Holland), 1989.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[HHK92] William M. Hsu, John F. Hughes, and Henry Kaufman. Direct manipulation of free-form deformations. *Computer Graphics*, 26(2):177–184, July 1992. SIGGRAPH '92 Conference Proceedings.

[HL88] K. Ho-Le. Finite element mesh generation methods: a review and classifiation. *Computer Aided Design*, 20(1):27–38, January 1988.

[JS86] B. Joe and R. B. Simpson. Triangular meshes for regions of complicated shapes. *International Journal for Numerical Methods in Engineering*, 23:751–778, 1986.

[JW90] H. Jin and N.-E. Wiberg. Two-dimensional mesh generation, adaptive remeshing and refinement. *International Journal for Numerical Methods in Engineering*, 29:1501–1526, 1990.

[Kol91] Craig E. Kolb. *Rayshade User's Guide and Reference Manual*, draft 0.1 edition, January 1991. Rayshade is available via anonymous ftp from ftp.princeton.edu.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Software Series. Prentice Hall, Englewood Cliffs, NJ, second edition, 1988.

[LC79] J. Lane and L. Carpenter. A generalized scan line algorithm for the computer display of parametrically defined surfaces. *Computer Graphics and Image Processing*, 11(3):290–297, November 1979.

[LCWB80] J. Lane, L. Carpenter, T. Whitted, and J. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23(1):23–34, January 1980.

[Lis94] D. Lischinski. *Graphics Gems IV*, chapter Incremental Delaunay Triangulation, pages 47–59. Academic Press, 1994.

[Lo85] S. H. Lo. A new mesh generation scheme for arbitrary planar domains. *International Journal for Numerical Methods in Engineering*, 21:1403–1426, 1985.

[Löh88] Rainald Löhner. Some useful data structures for the generation of unstructured grids. *International Journal for Numerical Methods in Engineering*, 4(1):123–135, 1988.

[LR80] J. Lane and R. Riesenfeld. A theoretical development for the computer generation of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):35–46, January 1980.

[LSP87] S.-L. Lien, M. Shantz, and V. Pratt. Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics*, 21(4):111–117, July 1987. SIGGRAPH '84 Conference Proceedings.

[MP77] R. S. Millman and G. D. Parker. *Elements of Differential Geometry.* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[Nyd72] R. Nydegger. A data minimization algorithm of analytical models for computer graphics. Master's thesis, University of Utah, Dept. of Computer Science, Salt Lake City, UT, 1972.

[Par77] Richard E. Parent. A system for sculpting 3d data. *Computer Graphics*, 11(2):138–147, August 1977. SIGGRAPH '77 Conference Proceedings.

[Par86] Scott A. Parry. *Free-Form Deformations in a Constructive Solid Geometry Modeling System.* PhD thesis, Department of Civil Engineering, Brigham Young University, Provo, UT, 1986.

[Pie91] Leslie A. Piegl. On NURBS: A survey. *IEEE Computer Graphics & Applications*, 11(1):55–71, January 1991.

[PPF+88] J. Peraire, J. Peiro, L. Formaggia, K. Morgan, and O. Zienkiewicz. Finite element euler computations in three dimensions. *International Journal for Numerical Methods in Engineering*, 26(10):2135–2159, 1988.

[PR95] Leslie A. Piegl and Arnaud M. Richard. Tesselating trimmed NURBS surfaces. *Computer-Aided Design*, 27(1):16–26, January 1995.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer Verlag, New York, NY, 1985.

[PVMZ88] J. Peraire, M. Vahdati, K. Morgan, and O. Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72(2):449–466, 1988.

[Ram87] L. Ramshaw. Blossoming: a connect–the–dots approach to splines. Technical Report 19, Digital Systems Research Center, Palo Alto, CA, June 1987.

[Ram88] L. Ramshaw. Béziers and B-splines as multiaffine maps. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI Series*, pages 757–776, Berlin, 1988. Springer Verlag.

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *Computer Graphics*, 25(4):107–116, August 1989. SIGGRAPH '89 Conference Proceedings.

[SAG84] T. W. Sederberg, D.C. Anderson, and R.N. Goldman. Implicit representation of parametric curves and surfaces. *Computer Vision, Graphics and Image Processing*, 28:72–84, 1984.

[Sch93a] Larry L. Schumaker. Computing optimal triangulations using simulated annealing. *Computer Aided Geometric Design*, 10:329–345, 1993.

[Sch93b] Larry L. Schumaker. Triangulations in CAGD. *IEEE Computer Graphics & Applications*, 13(1):47–52, January 1993.

[Sed83] Thomas W. Sederberg. *Implicit and Parametric Curves and Surfaces for Computer Aided Geometric Design.* PhD thesis, Purdue University, West Lafayette, IN, 1983.

[Sed93] Thomas W. Sederberg. Personal communication, 1993.

[Sei93] H.-P. Seidel. An introduction to polar forms. *IEEE Computer Graphics & Applications*, 13(1):38–46, January 1993.

[SH92] X. Sheng and B. E. Hirsch. Triangulation of trimmed surfaces in parametric space. *Compute-Aided Design*, 24(8):437–444, August 1992.

[Sha78] Michael I. Shamos. *Computation Geometry.* PhD thesis, Yale University, New Haven, CT, 1978. UMI #7819047.

[She88] Mark S. Shepard. Approaches to the automatic generation and control of finite element meshes. *Applied Mechanics Review*, 41(4):169–184, April 1988.

[SL87] M. Shantz and S. Lien. Shading bicubic patches. *Computer Graphics*, 23(4):189–196, July 1987. SIGGRAPH '84 Conference Proceedings.

[SP86a] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of polygonal data. In *Proceedings International Electronic Image Week*, pages 633–639, Nice, France, April 1986.

[SP86b] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *Computer Graphics*, 20(4):151–160, August 1986. SIGGRAPH '86 Conference Proceedings.

[SS90] W. J. Schroeder and M. S. Shepard. A combined octree/Delaunay method for fully automatic 3-d mesh generation. *International Journal for Numerical Methods in Engineering*, 29:37–55, 1990.

[vdW70] B. L. van der Waerden. *Algebra*, volume 2. Frederick Ungar, New York, NY, fifth edition, 1970.

[Ver75] K. J. Versprille. *Computer-Aided Design Application of the Rational B-Spline Approximation Form.* PhD thesis, Syracuse University, Syracuse, NY, 1975.

[Wal90] Bob Wallis. *Graphics Gems*, chapter Tutorial on Forward Differencing, pages 594–603. Academic Press, 1990.

[Wat81] D. F. Watson. Computing the n-dimensional delaunay tesselation with application to voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

[Wör81] Burkard Wördenweber. Automatic mesh generation of 2 and 3 dimension curvilinear manifolds. Technical Report 18, Computer Laboratory, University of Cambridge, Cambridge, UK, November 1981.

[WW92] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*, chapter The theory and practice of parametric representation techniques. ACM Press. Addison-Wesley, New York, NY, 1992.

[YS83] Mark A. Yerry and Mark S. Shepard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics & Applications*, 3(1):39–46, January 1983.

[YS84] Mark A. Yerry and Mark S. Shepard. Automatic tree-dimensional mesh generation by the modified-octree technique. *International Journal for Numertical Methods in Engineering*, 20:1965–1990, 1984.

[ZP71] O. C. Zienkiewicz and D. V. Phillips. An automatic mesh generation scheme for plane and curved surfaces by isoparametric co-ordinates. *International Journal for Numerical Methods in Engineering*, 3(4):518–528, 1971.

Appendix A

Colour Plates

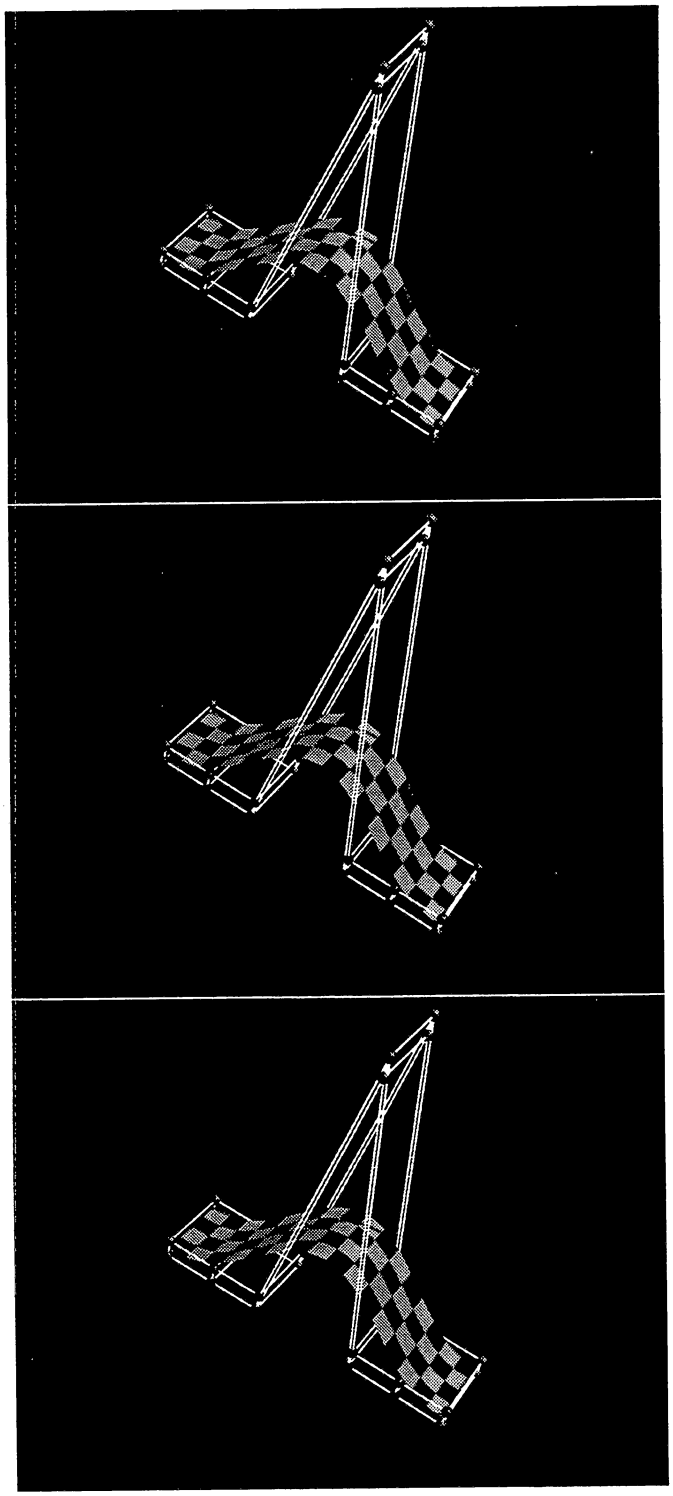


Figure A.1: Deformed polygon with $\varepsilon_{\max} = \infty$, $\varepsilon_{\max} = 0.5$ and $\varepsilon_{\max} = 0.2$

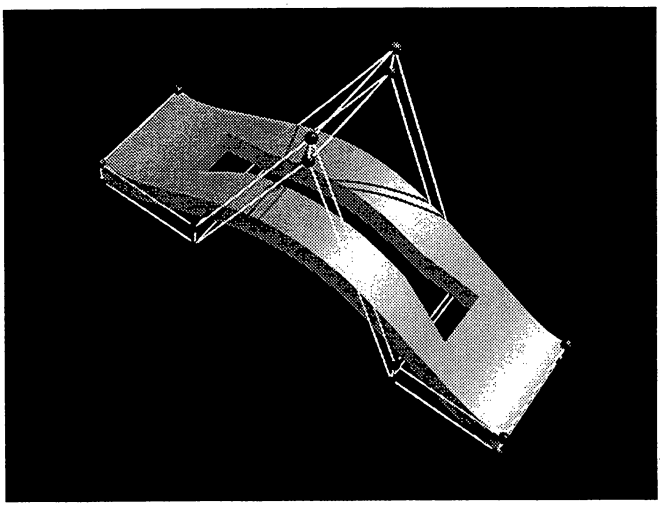


Figure A.2: Solid with $\varepsilon_{\max} = 0.1$

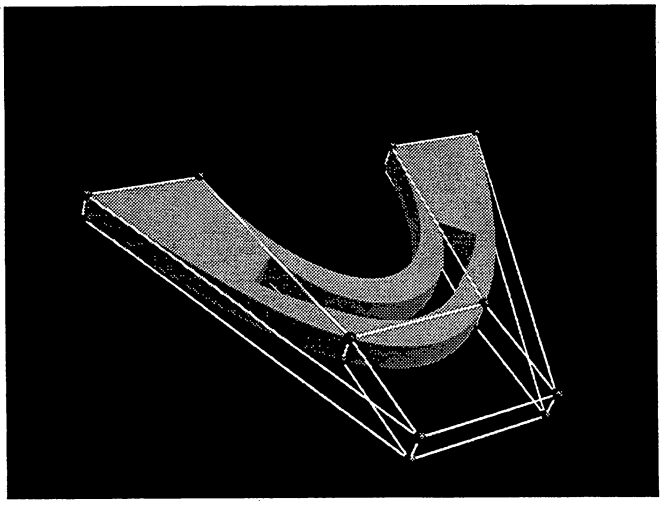


Figure A.3: Planar deformation of solid with $\varepsilon_{\max} = 0.1$

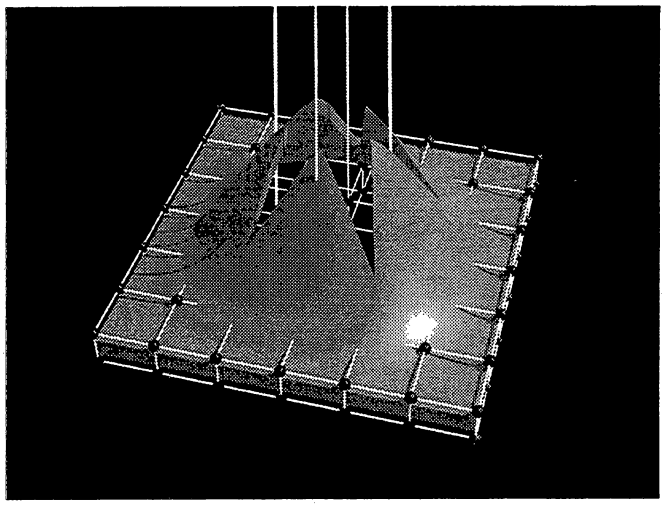


Figure A.4: Bell-shaped deformation of tile with star-shaped hole, $\varepsilon_{\max} = 0.2$

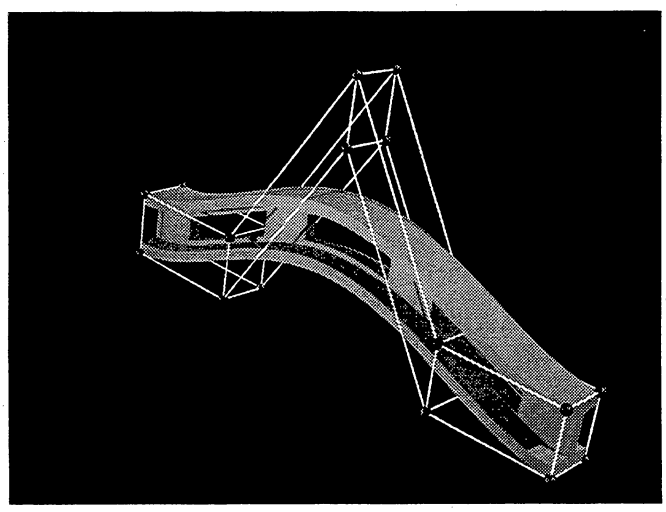


Figure A.5: Deformed girder with $\varepsilon_{\max} = 0.05$

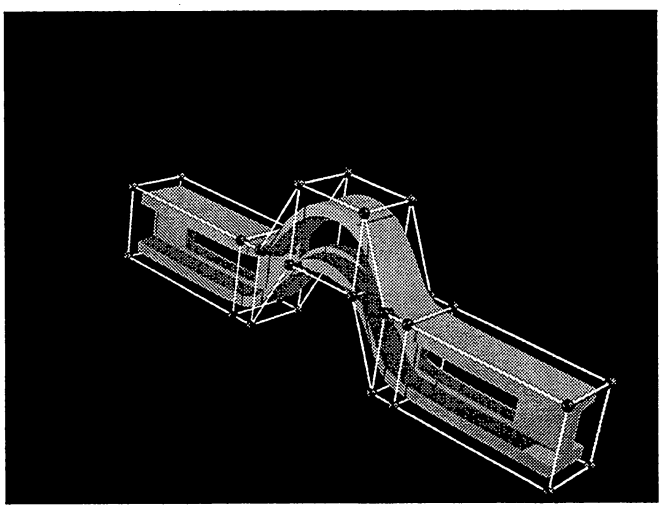


Figure A.6: Girder deformed by non-uniform B-spline, $\varepsilon_{\max} = 0.1$