UNIVERSITY COLLEGE LONDON
MSC IN INFORMATION TECHNOLOGY

MSc Project Report

CURVES AND SURFACES USING GLOOP

Nikolaos Platis

Supervisor: **Mel Slater**

Friday 6 September 1996

DISCLAIMER

This Report is submitted as part requirement for the MSc Degree in Information Technology at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

ABSTRACT

This Project studies and implements Bézier and B-Spline curves and surfaces. These are the most fundamental tools for describing curves and surfaces in Computer Aided Geometric Design.

The theoretical part of the Report examines at first *Bézier curves*, which are the basis for what follows. *B-Spline curves* are introduced next, as piecewise Bézier curves joining together with continuity constraints at the joins; *interpolation* with such curves is also discussed. Finally, *Bézier* and *B-Spline surfaces* are presented as a straightforward generalisation of the respective topics on curves.

The practical part of the Report concentrates on the Software Engineering issues regarding this Project, namely *Requirements*, *Design*, *Implementation* and *Testing*. The software constructed is built as an extension to GLOOP, a graphics package developed and used at UCL; it consists of *a set of C++ classes* implementing the graphical entities studied and is meant as a programming tool, providing all the necessary functionality for defining, manipulating and drawing curves and surfaces.

The Report concludes by presenting examples of the use of these classes, comparing the different methods implemented, and giving ideas for further work.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Mel Slater, for all his guidance and advice during my work on this Project. He has helped me understand the topics studied and his suggestions contributed greatly to the outcome of this Project. I also thank Simon Arridge for his co-operation with sorting out some issues related to GLOOP and for his constructive recommendations on my code.

Thanks also go to my fellow students; discussions with them have been enlightening on several issues and their company during the year was invaluable.

Finally I would like to thank my family, for encouraging me to do this course and for supporting me all the way through it; to them this Report is dedicated.

6 September 1996

CONTENTS

INTRODUCTION

The use of computers as a design tool in industry is widespread nowadays. They enable visualisation and modelling of various objects so that the designer can have a persuasive preview of their appearance and other properties; moreover, they can accurately control suitable machinery for the production of solid models and full scale articles. The automobile, ship and aircraft industries are among those that benefit the most from these techniques.

Such use of computers would not be possible without the appropriate mathematical tools to represent complex shapes in two and three dimensions. Whilst the simplest geometric primitives such as lines, arcs and polygons are treated by Computer Graphics, more general, freeform curves and surfaces are the field of Computer Aided Geometric Design (CAGD).

P. Bézier was working at Renault in the 1960's when he developed the theory of the curves now bearing his name. (The same concepts had been developed a little earlier by P. de Casteljau at Citroën, but his work was not published at the time.) His efforts were driven by the need to represent complex curves and surfaces in a way suitable for computer manipulation: numerically controlled machinery existed that could produce complexly shaped objects if the appropriate data was fed in by a computer, but no concrete representation of such shapes was available and only approximations could be used instead.[1]

Bézier curves provide a way of representing polynomial curves of arbitrarily high degree in a concise, mathematically simple, and intuitive way. They are defined by a set of control points and their shape is reasonably predictable from these points. They are the most simple tool in CAGD and yet the most fundamental, as they form the basis for more complex and powerful ways of representing curves.

B-Spline curves are piecewise Bézier curves joining together with continuity constraints at the joins, so that the overall shape of the curve is smooth. While retaining the desirable properties of Bézier curves, in addition they possess local control which adds flexibility to their shape and allows for easy manipulation, in interactive environments. They are well suited to the representation of complex shapes and thus they are commonly used in practice.

There exist other forms of representing curves, such as rational Bézier and B-Spline curves, ν-, β-, and γ- splines. All these offer greater flexibility on the shape of the curve at an increased mathematical and computational complexity.

---

[1] See Chapter 1 in Farin [1], written by P. Bézier himself.

The ideas developed for curves generalise to describe surfaces. The basic tool here is tensor product Bézier surfaces, which can be viewed as the outcome of the points of an initial Bézier curve tracing (Bézier) curved paths in three dimensions. It is straightforward that these surfaces have properties analogous to the ones of Bézier curves.

B-Spline surfaces can be formed, evidently, as piecewise Bézier surfaces with continuity constraints at the joins. Again, they are preferred in practice because they can reliably describe complex shapes.

Scope of this Project and context of work

This Project studies the two most basic tools of CAGD, namely Bézier and B-Spline curves and surfaces (in two and three dimensions respectively). It extends to other, more advanced concepts such as the ones mentioned above, only to suggest further work.

These tools are implemented as a set of C++ classes, to be used by a programmer in order to define, manipulate and draw such graphical entities. They are built as an extension to GLOOP, a Graphics Language for Object Oriented Programming developed and used within the Department of Computer Science at UCL. GLOOP already provides the functionality needed to define and draw basic geometric primitives in 2D and 3D such as points, lines and polygons; so, the effort in this Project is focused on providing efficient algorithms, clearly written and well integrated with the rest of the package. No end-user programs are constructed to handle these curves and surfaces interactively; only small programs are written for evaluation purposes.

Structure of the Report

The Report in hand consists of an Introduction, three chapters, a Conclusion and Appendices. An overview of the rest of the Report follows:

- Chapter 1 presents all the theoretical concepts and results needed for the work of this Project. It starts with some background material on multi-affine maps and blossoms, which will be used throughout in the Chapter. Then it moves on to Bézier curves, introduced with the traditional approach based on the Bernstein polynomials. The alternative approach based on blossoming leads easily to the de Casteljau algorithm for drawing Bézier curves. Finally, a second algorithm for drawing these curves is given, based on forward differences.

  The next section treats B-Spline curves, mainly using the blossoming approach. Several topics related to manipulation of the knot sequence of a given B-Spline curve are discussed. Finally, interpolation of a set of data points with a cubic B-Spline curve is examined.

  The last section of this Chapter presents Bézier and B-Spline surfaces. The discussion here is less elaborate than for curves, because essentially no new concepts are introduced – everything is a generalisation of the respective topics on curves.

- Chapter 2 deals with the Software Engineering issues of this Project. After an initial section presenting GLOOP, necessary for understanding the classes developed, there follow sections on the Requirements, Design, Implementation and Testing of the software constructed. These should form an adequate presentation of the software written for this Project.
- Chapter 3 shows some example results produced with the classes built. Through them, it contrasts the different ways of defining curves and surfaces and compares the alternative methods provided for their drawing.
- The Conclusion section presents critically the outcome of the Project, stating its successes and inefficiencies. It also provides guidelines for further work and continuation of what is achieved.
- Last but not least, the Appendices provide reference information for the user of the classes constructed, in the form of standard manual pages, and contain the full header files and source code. An example program drawing a teapot is given, to present the classes at work.

A remark on notation

The subject of this Project imposes the use of many mathematical formulae throughout. In the following we will denote by lowercase italic letters any real variables and functions used and by lowercase boldface letters any vector quantities such as points in 2D or 3D and vector functions used. For example, $f(t)$ is a real function of a real variable t, whereas $\mathbf{b}(t) = \big(x(t), y(t), z(t)\big)$ is a vector function of a real variable t with coordinate functions $x(t)$, $y(t)$ and $z(t)$.

1. THEORY OF CURVES AND SURFACES FOR CAGD

In this chapter are presented all the theoretical concepts and results used for the work of this Project: those necessary for rendering Bézier and B-Spline curves and surfaces (in 2D and 3D respectively), and those related to interpolating a set of given data points with a B-Spline curve or surface.

Effort has been made so that the material is self-contained; the only prerequisites for its understanding are basic mathematical concepts such as polynomials, functions and linear systems, as well as basic notions of Computer Graphics. For this reason, the presentation is quite lengthy; besides, one of the main objectives of the Project was the understanding of the relevant theory, and this justifies the extent of this chapter. On the other hand, topics that were not applied in this Project are not treated in any extent, however important or closely related to the subject they may be; such topics are usually mentioned briefly and the appropriate references are provided for further study.

In this chapter are incorporated some topics which were not mentioned in any of the referenced texts and whose development is the result of my own effort. This approach was chosen in favour of presenting them separately, because it adds to the coherence and continuity of the text.

1.1 Multi-affine maps and Blossoms

Multi-affine maps are an essential tool for our analysis of Bézier and B-Spline curves, and the concept of blossoming is also used throughout. So we devote to them a separate section in our discussion. Most of the concepts in this section can be found in [2], and Chapter 2 of [1] provided several details.

1.1.1 Affine and Multi-affine maps

Barycentric and convex combinations of points

Given a set of points $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n$, a *barycentric combination* of them $\mathbf{b}$ is a weighted sum of these points where the weights add to one:

$$\mathbf{b} = \sum_{i=0}^{n} a_i \mathbf{b}_i \quad \text{with} \quad a_0 + a_1 + \ldots + a_n = 1. \tag{1.1}$$

If all coefficients a_i in the last formula are non-negative, we have the notion of a *convex combination* of the points $\mathbf{b}_i$. It is termed so because it is always inside the *convex hull* of these points; this is the minimum convex polygon that contains all the given points, as depicted in Figure 1-1.

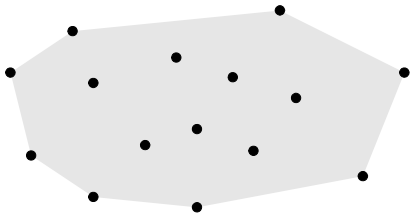

Figure 1-1: Convex hull (shaded) of a set of points.

Affine maps

An *affine map* is a function that preserves barycentric combinations. In more formal terms, if

$$x = \sum_{i=0}^{n} a_i x_i \quad \text{with} \quad \sum_{i=0}^{n} a_i = 1,$$

then $f(x)$ is an affine map if it satisfies

$$f(x) = f\left(\sum_{i=0}^{n} a_i x_i\right) = \sum_{i=0}^{n} a_i \, f(x_i). \tag{1.2}$$

The general form of a (real) affine map is

$$f(x) = c + dx \tag{1.3}$$

and it can be seen that this form actually satisfies (1.2).

The concept of affine maps generalises straightforwardly to functions of a vector variable. In this context it is important to note that all basic geometrical transformations used in Computer Graphics, viewed as functions mapping the plane (or 3D space) into itself, are in fact affine maps; they can all (except for projection) take the form (1.3) with d now being a suitable 2×2 (or 3×3) matrix and c a suitable vector. This will prove important later on when we discuss Bézier curves.

Multi-affine maps

Multi-affine maps are simply functions of many variables that are affine for each of their variables separately.

The general form of a multi-affine function of n variables, analogously to (1.3), is a sum of a constant and of factors of *all* combinations of the variables taken one, two, …, n at a time. An example should be clarifying:

$$f(t_1, t_2, t_3) = 1 + 2t_1 - t_2 + t_3 + 4t_1t_2 + 6t_1t_3 - 5t_2t_3 + 9t_1t_2t_3 .$$

A multi-affine map is called *symmetric* if all terms corresponding to equal number of variables (t_1t_2, t_1t_3 and t_2t_3 in the above example) have equal coefficients; in that case the order of its attributes is immaterial. The above multi-affine map is not symmetric.

An important element of a multi-affine map is its *diagonal*; this is the one-variable function that is obtained if all its arguments are substituted with a single new one. For a multi-affine map of n variables the diagonal is always an n^{th} degree polynomial. For example, the above map has the diagonal

$$f(t,t,t) = 1 + 2t + 5t^2 + 9t^3,$$

which is a third degree polynomial of t.

1.1.2 The Blossoming theorem

The previous property of multi-affine functions is in fact the first part of an important theorem for CAGD. This is the *Blossoming Theorem*, which we now present in full:

- Every n-variable multi-affine map has a unique n^{th} degree polynomial as its diagonal; and vice versa,
- Every n^{th} degree polynomial is the diagonal of a unique symmetric n-variable multi-affine map.

The first part is already mentioned above. The second part is not difficult to prove. Consider the polynomial

$$f(t) = \sum_{i=0}^{n} a_i t^i = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n .$$

To get from this a symmetric multi-affine function of n-variables we need to split each coefficient a_k equally to all terms corresponding to combinations of k (out of the n) variables which contribute to t^k in the inverse procedure. These are $t_1 t_2 \ldots t_{k-1} t_k$, $t_1 t_2 \ldots, t_{k-1} t_{k+1}$, …, $t_1 t_2 \ldots t_{k-1} t_n$, …, $t_{n-k+1} t_{n-k+2} \ldots t_{n-1} t_n$ and are in number as much as the combinations of n by k. From this analysis it follows that the n-variable function

$$F(t_1, t_2, \ldots, t_n) = a_0 + \frac{a_1}{n}[t_1 + \ldots + t_n] + \frac{a_n}{\binom{n}{2}}[t_1 t_2 + t_1 t_3 + \ldots + t_{n-1} t_n] + \ldots + a_n [t_1 t_2 \ldots t_n] \qquad (1.4)$$

is, by its form, multi-affine and symmetric, and has the original $f(t)$ as its diagonal.

The multi-affine map given in (1.4) is called the *blossom* or *polar form* of the polynomial $f(t)$. The interconnection between the two will be of great importance later.

1.2 Bézier curves

Bézier curves are the fundamental tool for drawing curves in Computer Aided Geometric Design. Their importance is due to their relatively simple mathematical form, their interesting properties and the fact that they form the basis for more complex and powerful tools such as B-Splines. In this section we present the basic concepts of Bézier

curves and also the two algorithms implemented for their rendering: the traditional *de Casteljau algorithm* and an algorithm using *forward differences*.

In the following, curves will be represented in their *parametric form*,

$$\mathbf{p}(t) = \big(x(t), y(t)\big), \quad t \in [r, s] \tag{1.5}$$

where x and y are taken to be functions of the *parameter* t. It is often convenient to use $t \in [0,1]$ but this is by no means restrictive as a reparametrisation to the arbitrary interval $[r, s]$ will always be possible.

The material in this section is a blend of the relevant sections of [2] and of Chapters 3 and 4 of [1], except for the differences method as noted in its introduction.

1.2.1 Bézier curves using the Bernstein basis

Interpolation

Given two points $\mathbf{p}_0$ and $\mathbf{p}_1$, the equation of the straight line segment from $\mathbf{p}_0$ to $\mathbf{p}_1$ is

$$\mathbf{p}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1, \quad t \in [0,1]. \tag{1.6}$$

This line segment is said to *interpolate* the two given points. With respect to what we mentioned earlier, this is a convex combination of the two points, and, moreover, an affine map of the interval $[0,1]$ into the straight line segment; to prove this, we observe that it can be written in the form

$$\mathbf{p}(t) = \mathbf{p}_0 + (\mathbf{p}_1 - \mathbf{p}_0)\,t$$

which is exactly analogous to (1.3).

Quadratic Bézier curves

We now consider three points $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$. As before, we can interpolate $\mathbf{p}_0$ and $\mathbf{p}_1$, and $\mathbf{p}_1$ and $\mathbf{p}_2$ separately, by

$$\begin{aligned} \mathbf{p}_0^1(t) &= (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \\ \mathbf{p}_1^1(t) &= (1-t)\mathbf{p}_1 + t\mathbf{p}_2 \end{aligned}$$

an then $\mathbf{p}_0^1(t)$ and $\mathbf{p}_1^1(t)$ again, by

$$\begin{aligned} \mathbf{p}_0^2(t) &= (1-t)\mathbf{p}_0^1(t) + t\mathbf{p}_1^1(t) \\ &= (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2 \end{aligned} \tag{1.7}$$

(In all $\mathbf{p}_i^n(t)$ above, i refers to the initial point and n refers to the interpolation step.) We observe that $\mathbf{p}_0^2(t)$ traces out a quadratic curve on t, i.e. a parabola, as t varies from 0 to 1. This curve is in fact the *quadratic Bézier curve* with control points $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$, and we will denote it by $\mathbf{p}^2(t)$. The procedure described above is presented graphically in Figure 1-2.

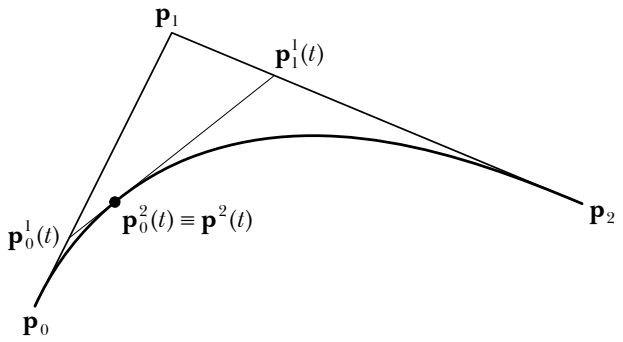

Figure 1-2: Quadratic Bézier curve.

Bézier curves of degree n

This procedure can be generalised. For example, with four control points we get a cubic Bézier curve; the interpolation steps needed here are obviously three. The intermediate points can be conveniently written into a triangular array of points called a *de Casteljau triangle*:

$$\begin{array}{llll} \mathbf{p}_0 & \mathbf{p}_0^1 & & \\ \mathbf{p}_1 & \mathbf{p}_1^1 & \mathbf{p}_0^2 & \underline{\mathbf{p}_0^3} \\ \mathbf{p}_2 & \mathbf{p}_2^1 & \mathbf{p}_1^2 & \\ \mathbf{p}_3 & & & \end{array}$$

In the more general case, it is not difficult to show that the n^{th} *degree Bézier curve* having *control points* $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ is given by

$$\mathbf{p}^n(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} \mathbf{p}_i \tag{1.8}$$

with the intermediate points being

$$\mathbf{p}_i^r(t) = (1-t)\mathbf{p}_i^{r-1}(t) + t\mathbf{p}_{i+1}^{r-1}(t), \quad \begin{array}{l} r = 1, \ldots, n \\ i = 0, \ldots, n-r \end{array} \tag{1.9}$$

Bernstein polynomials and Bézier curves

The coefficients of $\mathbf{p}_i$ in (1.8) are special polynomials, called the *Bernstein polynomials*. They are

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0,1,2,\ldots,n \tag{1.10}$$

and they have some interesting properties:

- They are a basis of the vector space of n^{th} degree polynomials. In other words, every polynomial $f(t)$ of degree n can be written in the form

$$f(t) = \sum_{i=0}^{n} c_i B_i^n(t)$$

where c_i are suitable (scalar) coefficients.

- For any n, all $B_i^n(t)$ are positive and sum up to one; this can be seen by their defining formula.
- The Bernstein polynomials are symmetric with respect to t and $1-t$; i.e.

$$B_j^n(t) = B_{n-j}^n(1-t).$$

With these in hand, our definition of the n^{th} degree Bézier curve may be re-written as:

$$\mathbf{p}^n(t) = \sum_{i=0}^{n} B_i^n(t)\,\mathbf{p}_i \tag{1.11}$$

Properties of Bézier curves

This last formulation of the definition of a Bézier curve reveals, in conjunction with many of the details exposed earlier, some useful properties of Bézier curves.

- **Convex hull property:** The Bézier curve is a barycentric and more precisely a convex combination of its control points. So, it is always inside the convex hull of its control points. This property is useful for predicting the shape of the curve; it may also be used to check for intersection of the curve with a straight line: if the line is outside the convex hull of the curve then it cannot possibly intersect with it. [An even simpler test for this purpose is to check for intersection of the line with the *minmax box* of the Bézier curve, i.e. the minimum upright rectangle which contains the control points and thus the whole curve.]
- **Affine invariance property:** Being a barycentric combination of its control points, a Bézier curve is invariant under affine transformations such as all (except for projection) of the basic geometric transformations. In other words, if we apply a transformation to the control points of the curve, then the resulting curve will be the initial curve transformed by the same transformation.
- **Invariance under affine parameter transformations:** The curve stays the same if we change the parameter interval from $t \in [0,1]$ to $u \in [a,b]$; in fact this is an affine transformation of the parameter t into $u = a + (b-a)t$, from which the name of the property follows. In this case the intermediate Bézier points are

$$\mathbf{p}_i^r(t) = \frac{b-u}{b-a}\mathbf{p}_i^{r-1}(t) + \frac{u-a}{b-a}\mathbf{p}_{i+1}^{r-1}(t). \tag{1.12}$$

- **Symmetry:** It does not matter if we take the control points to be in reverse order, $\mathbf{p}_n, \mathbf{p}_{n-1}, \ldots, \mathbf{p}_0$; simply the curve will be traced in the opposite direction. This follows by the symmetry of the Bernstein polynomials.
- **Linear precision:** If the control points are on a straight line, then the curve itself will be a straight line.
- **End point interpolation:** It is immediate to verify that

$$\mathbf{p}^n(0) = \mathbf{p}_0 \quad \text{and} \quad \mathbf{p}^n(1) = \mathbf{p}_n$$

  i.e. the curve starts at the first and ends at the last control point. This property is also useful for predicting the shape of the Bézier curve.

- **Tangent vectors:** The tangent vectors at the two ends of the Bézier curve are parallel to the sides of its control polygon (the polygon formed by the control points). More precisely, it can be shown that

$$\frac{d}{dt}\mathbf{p}^n(0) = n(\mathbf{p}_1 - \mathbf{p}_0) \quad \text{and} \quad \frac{d}{dt}\mathbf{p}^n(1) = n(\mathbf{p}_n - \mathbf{p}_{n-1}).$$

  This property further constraints the shape of the curve.

- **Pseudo-local control:** It is important to mention that Bézier curves do *not* have local control, i.e. a change in one of the control points will affect the whole curve. This happens because the $B_i^n(t)$, essentially the weights with which each point $\mathbf{p}_i$ contributes in the curve, are defined over the whole range of the parameter t; thus a change in $\mathbf{p}_i$ will have an effect over the whole range of t and so over the whole curve. *However*, each $B_i^n(t)$ has only one maximum and attains it at $t = i/n$; so a change in $\mathbf{p}_i$ will mostly affect the curve over the region around the parameter value i/n. This makes the effect of changes in the control points reasonably predictable.

Having examined Bézier curves in their Bernstein basis form, we now move on to describing them using an alternative approach based on blossoming. This will facilitate the introduction of the de Casteljau algorithm and will be even more useful for the discussion of topics on B-Splines. Especially for this reason, in the following we will assume the parameter t in an arbitrary interval $[a,b]$ and not in $[0,1]$.

1.2.2 Bézier curves using blossoms

Back to multi-affine maps and interpolation

Suppose $f(t)$ is an affine map and that we are given the values $f(a)$ and $f(b)$ for some $a < b$. Then we can find the value of $f(t)$ for any $t \in [a,b]$ as follows: It holds that

$$t = \frac{b-t}{b-a}a + \frac{t-a}{b-a}b;$$

so, since $f(t)$ is affine (and the coefficients of a and b above sum to one) it also holds that

$$f(t) = \frac{b-t}{b-a}f(a) + \frac{t-a}{b-a}f(b). \tag{1.13}$$

This is the analogous of our interpolation formula (1.6) for the arbitrary interval $[a,b]$ [cf. also (1.12)].

Now suppose that $f(t_1,t_2)$ is a symmetric multi-affine map of 2 variables and that we are given the values $f(a,a)$, $f(a,b)$ and $f(b,b)$ for some $a < b$. Then we can also find the value of $f(t,t)$ for any $t \in [a,b]$, this time by two stages of interpolation: first we need

$$f(a,t) = \frac{b-t}{b-a}f(a,a) + \frac{t-a}{b-a}f(a,b) \quad \text{and} \quad f(t,b) = \frac{b-t}{b-a}f(a,b) + \frac{t-a}{b-a}f(b,b)$$

and then we combine them to find

$$f(t,t) = \frac{b-t}{b-a} f(a,t) + \frac{t-a}{b-a} f(b,t)$$

[Note that $f(b,t) = f(t,b)$ thanks to the symmetry of the affine map.] The procedure outlined here is very similar to the one that led us to the definition of the Bézier curve. Indeed this is the case; essentially, we only introduce a new notation for the same concept.

The blossom of a Bézier curve

Suppose first that we are given four points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ and an interval $[a,b]$. We can find (though we will never need to!) a symmetric multi-affine map of three variables $f(t_1,t_2,t_3)$ such that $f(a,a,a) = \mathbf{p}_0$, $f(a,a,b) = \mathbf{p}_1$, $f(a,b,b) = \mathbf{p}_2$ and $f(b,b,b) = \mathbf{p}_3$.[1] Then, by repeated interpolation, we can find the value of $f(t,t,t)$ for any $t \in [a,b]$. The construction procedure shows that $f(t,t,t)$ will actually be a point on the cubic Bézier curve with $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ as control points. The de Casteljau triangle now takes the form

$$\begin{array}{llll} f(a,a,a) & & & \\ f(a,a,b) & f(a,a,t) & & \\ f(a,b,b) & f(a,t,b) & f(a,t,t) & \\ f(b,b,b) & f(t,b,b) & f(t,t,b) & \underline{f(t,t,t)} = \mathbf{p}^3(t) \end{array} \qquad (1.14)$$

For obvious reasons, $f(t,t,t)$ is called the *blossom* of the Bézier curve $\mathbf{p}^3(t)$. The procedure, as well as the relation between the two notations, is depicted in Figure 1-3.

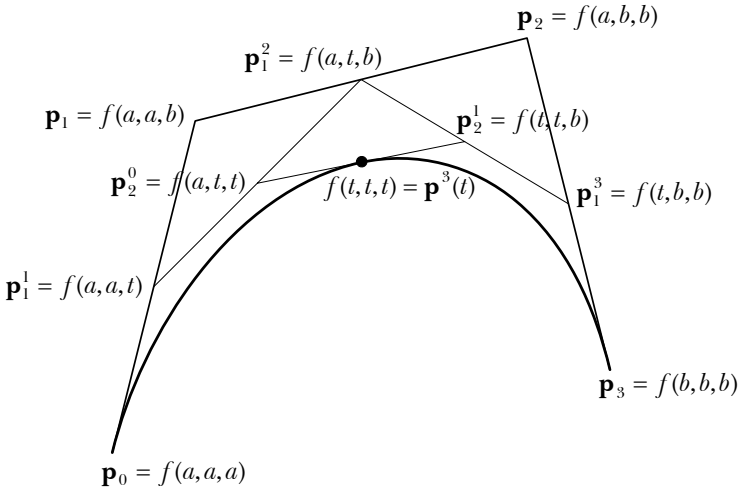

Figure 1-3: Cubic Bézier curve.

In the general case of an n^{th} degree Bézier curve defined over the interval $[a,b]$, the control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ are considered to take the values of an n-variable symmetric multi-affine map $f(t_1,t_2,\ldots,t_n)$ as follows:

$$\mathbf{p}_i = f(a^{<n-i>}, b^{<i>}), \quad \text{for } i = 0,\ldots,n \qquad (1.15)$$

[1] Such a map always exists: it is sufficient that the corresponding 3rd degree polynomial (the diagonal of the map) exists; this has four 'unknown' coefficients and the four given points are enough to make equal number of equations in order to find them.

(where $t^{<r>}$ means that t appears r times as an argument). The intermediate points are

$$\mathbf{p}_i^r(t) = f(a^{<n-i-r>}, t^{<r>}, b^{<i>}) \tag{1.16}$$

and the points on the curve are given by

$$\mathbf{p}^n(t) = f(t^{<n>}). \tag{1.17}$$

1.2.3 The de Casteljau algorithm

Subdivision

Let us have a second look at Figure 1-3 and re-label some of the points as follows:

$$\begin{aligned} \mathbf{l}_0 &= f(a,a,a), \quad \mathbf{l}_1 = f(a,a,t), \quad \mathbf{l}_2 = f(a,t,t), \quad \mathbf{l}_3 = f(t,t,t) \\ \mathbf{r}_0 &= f(t,t,t), \quad \mathbf{r}_1 = f(t,t,b), \quad \mathbf{r}_2 = f(t,b,b), \quad \mathbf{r}_3 = f(b,b,b) \end{aligned} \tag{1.18}$$

Comparing these relations to (1.15) we observe that the $\mathbf{l}_i$ are essentially control points of a cubic Bézier curve for the interval $[a,t]$ and the $\mathbf{r}_i$ are control points of a cubic Bézier curve for the interval $[t,b]$. Since the same blossom function f of the initial curve is involved, these two curves are exactly the 'left' and 'right' part of the initial curve. This is the basis for the *de Casteljau algorithm* for generating a Bézier curve.

The algorithm

Obviously the subdivision process may be continued recursively; eventually it will converge to the Bézier curve itself. Actually the subdivision can stop if the control points of a curve segment are on a straight line (cf. relevant property of Bézier curves). This reveals an important property of the de Casteljau algorithm, the fact that it is adaptive: it will do fewer steps where the curve is flat and more where it is presents abrupt changes in curvature. For practical purposes, the process can stop when the points lie on a straight line within a given tolerance level; this tolerance will necessarily be a compromise between the accuracy of drawing and the required speed.

The de Casteljau algorithm in the general case can be formulated as follows:

```
procedure Bézier(pi)  {
      if (pi are collinear) then
            draw line (p0, pn)
      else  {
            split pi into li and ri for parameter value t ;
            Bézier(li) ;
            Bézier(ri) ;
      }
}
```

Colinearity test

The recursion steps in de Casteljau algorithm stop when the control points of a curve segment lie on a straight line within a given tolerance level.

To check any set of points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$ for 'flatness', we must form the line passing through the two end points $\mathbf{p}_0$ and $\mathbf{p}_n$, and check the distance of every other point from the line for being less than the tolerance.

Setting $\mathbf{p}_i = (x_i, y_i)$, the line through $\mathbf{p}_0$ and $\mathbf{p}_n$ is given by the determinant

$$\begin{vmatrix} x & y & 1 \\ x_0 & y_0 & 1 \\ x_n & y_n & 1 \end{vmatrix} = 0 \tag{1.19}$$

which developed gives an equation of the form $Ax + By + C = 0$. In fact, for such an equation to represent a line, A and B must not be both zero. If this condition is satisfied, the distance of an arbitrary point (x, y) from this line is given by

$$d = \frac{Ax + By + C}{\sqrt{A^2 + B^2}}. \tag{1.20}$$

(Note that the denominator is not zero thanks to our condition.)

Relation (1.19) shows that $A = (x_0 - x_n)$ and $B = (y_n - y_0)$; thus $\sqrt{A^2 + B^2}$ is actually the distance between the two end points. So, if $A = B = 0$ the two end points coincide and then the distance to examine should be simply

$$d' = \sqrt{(x - x_0)^2 + (y - y_0)^2} \tag{1.21}$$

i.e. the distance from one of the end points. In practice it is useful to formulate $\sqrt{A^2 + B^2}$ and if this is less than some very small positive number ε to use (1.21), otherwise to use (1.20) for the distance formula.

The formulae presented here are computationally very intensive. In fact they are responsible for most of the computational complexity of the de Casteljau algorithm, as the subdivision process itself is rather simple, consisting only of few additions and multiplications.

1.2.4 Drawing Bézier curves using forward differences

In this section is presented an alternative algorithm for drawing Bézier curves, which uses forward differences for the evaluation of the function representing the curve. The description will inevitably be detailed and somewhat lengthy as this algorithm requires some background results independent to the material exposed so far. Moreover, although this seems a common procedure, it does not appear in any of the referenced texts and the following is based on my own understanding of this topic.

Forward differences of a function

Given a function $f(t)$ of a real variable t, and a real number h, the *forward difference operator* Δ can be defined as

$$\Delta f(t) = f(t + h) - f(t)$$

and furthermore the *iterated forward difference operator* Δ^r as

$$\Delta^r f(t) = \Delta^{r-1} f(t+h) - \Delta^{r-1} f(t)$$

for all $r = 1,2,\ldots$, with $\Delta^0 f(t) = f(t)$.

Here it will be of interest the case when $f(t)$ is a polynomial of degree n. Then it can be shown that $\Delta^n f(t)$ is constant for all t and thus $\Delta^r f(t) = 0$ for all $r > n$.

The last relation can be rewritten as

$$\Delta^{r-1} f(t+h) = \Delta^{r-1} f(t) + \Delta^r f(t).$$

This gives an efficient way to evaluate the function f for $t = 0,\ h,\ 2h,\ldots$ if the values of $f(0), \Delta f(0), \Delta^2 f(0),\ldots, \Delta^n f(0)$ are known, according to the following scheme:

t	$f(t)$	$\Delta f(t)$	$\Delta^2 f(t)$	$\cdots$	$\Delta^n f(t)$
0	$f(0)$	$\Delta f(0)$	$\Delta^2 f(0)$	$\cdots$	$\Delta^n f(0)$
h	$f(h)$	$\Delta f(h)$	$\Delta^2 f(h)$	$\cdots$	$\Delta^n f(h)$
$2h$	$f(2h)$				
$\vdots$					

In this table, each element is the sum of the two elements above and above and to the right of it. (We need not worry about the last column as $\Delta^n f(t) = \Delta^n f(0)$ for all t.) The whole process is very quick because it involves only additions, and the only difficult (and costly) point is the initial (but one-off) calculation of $\Delta^r f(0)$.

To find a formula for these differences[2], let us consider the general form of an n^{th} degree polynomial,

$$f(t) = \sum_{i=0}^{n} a_i t^i = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n.$$

Immediately $f(0) = a_0$. One step further,

$$\begin{aligned} \Delta f(t) &= f(t+h) - f(t) \\ &= a_1 h + a_2\left[(t+h)^2 - t^2\right] + \ldots + a_n\left[(t+h)^n - t^n\right] \end{aligned}$$

which yields

$$\Delta f(0) = a_1 h + a_2 h^2 + \ldots + a_n h^n = \sum_{i=1}^{n} a_i h^i.$$

For $r = 2$ we have

$$\begin{aligned} \Delta^2 f(t) &= \Delta f(t+h) - \Delta f(t) \\ &= a_2\left[(t+2h)^2 - 2(t+h)^2 + t^2\right] + \ldots + a_n\left[(t+2h)^n - 2(t+h)^n + t^n\right] \\ &= a_2 \cdot 2h^2 + \ldots + a_n\left[(t+2h)^n - 2(t+h)^n + t^n\right] \end{aligned}$$

and so

$$\Delta^2 f(0) = a_2 \cdot 2h^2 + \ldots + a_n h^n (2^n - 2).$$

The above generalise to

---

[2] We will often omit 'forward' for simplicity, as we will not refer to 'backward' differences in any case. (They do exist, however!)

$$\Delta^r f(t) = \sum_{i=r}^{n} a_i \left[\sum_{j=0}^{r} (-1)^{r-j} \binom{r}{j} (t + jh)^i \right]$$

which leads to

$$\Delta^r f(0) = \sum_{i=r}^{n} a_i h^i \left[\sum_{j=0}^{r} (-1)^{r-j} \binom{r}{j} j^i \right]. \tag{1.22}$$

(From the above analysis, we can convince ourselves that $\Delta^n f(t)$ is constant: in general, at step r all terms less than r have vanished and the first term to appear is $a_r h^r c$ where c is some constant.)

An algorithm for the calculation of $\Delta^r f(0)$ will be given in the following, to ease the use of this convoluted formula. Before that, some information on differences of a sequence of numbers is needed.

Forward differences of a sequence of numbers

Given a (finite) sequence of numbers $x_0, x_1, x_2, \ldots, x_n$, the *forward difference operator* Δ can be defined as[3]

$$\Delta x_i = x_{i+1} - x_i$$

for all $i = 0,1,2,\ldots,n-1$, and furthermore the *iterated forward difference operator* Δ^r as

$$\Delta^r x_i = \Delta^{r-1} x_{i+1} - \Delta^{r-1} x_i$$

for all $r = 1,2,\ldots,n-1$ and $i = 0,1,2,\ldots,n-r$, with $\Delta^0 x_i = x_i$.

For example, the first few differences of x_i are

$$\Delta^0 x_i = x_i$$
$$\Delta^1 x_i = x_{i+1} - x_i$$
$$\Delta^2 x_i = x_{i+2} - 2x_{i+1} + x_i$$

and in the general case it can be shown that the r^{th} difference of x_i is given by

$$\Delta^r x_i = \sum_{j=0}^{r} (-1)^{r-j} \binom{r}{j} x_{i+j} .$$

For x_0 this gives

$$\Delta^r x_0 = \sum_{j=0}^{r} (-1)^{r-j} \binom{r}{j} x_j \tag{1.23}$$

which is very similar to the formula in brackets in (1.22) !

An intuitive way of calculating differences of numbers is shown in the following scheme:

---

[3] This should not be confused with the difference operator for functions; their meaning is different and which one Δ refers to is always clear in the formulae.

$$\begin{array}{llllll} x_0 & & & & & \\ x_1 & \Delta x_0 = x_1 - x_0 & & & & \\ x_2 & \Delta x_1 = x_2 - x_1 & \Delta^2 x_0 = \Delta x_1 - \Delta x_0 & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ x_n & \Delta x_{n-1} = x_n - x_n & \Delta^2 x_{n-2} = \Delta x_{n-1} - \Delta x_{n-2} & \cdots & \Delta^n x_0 = \Delta^{n-1} x_1 - \Delta^{n-1} x_0 \end{array} \tag{1.24}$$

Before we go on to the algorithm for calculation of $\Delta^r f(0)$, let us note that the concept of differences of numbers can be generalised straightforwardly to differences of points in 2D (and 3D) space, with any subtractions being made for each of the coordinates separately; thus, given a sequence of points $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n$, forward differences can be defined as

$$\Delta^r \mathbf{b}_i = \Delta^{r-1} \mathbf{b}_{i+1} - \Delta^{r-1} \mathbf{b}_i$$

for all $r = 1,2,\ldots,n-1$ and $i = 0,1,2,\ldots,n-r$, with $\Delta^0 \mathbf{b}_i = \mathbf{b}_i$. All the above details hold in this case, too.

Calculating forward differences of a function

We are now ready to calculate the forward differences $\Delta^r f(0)$. For this, we observe that the sum in brackets in (1.22) is very similar to the sum giving the differences of a sequence of numbers, and by (1.23) it is obvious that we need

$$x_j = j^i$$

[i is constant within the inner sum of (1.22)]. In other words, to use the scheme of (1.24), we have to form a vector of length r with elements $0, 1, 2^i, 3^i, \ldots, r^i$ and use it to calculate the r^{th} difference of its first element (zero!). If we call this vector 0^i (to remind us of x_0 and of the index i of the outer summation – the length r is not very important as we will soon find out), we may rewrite (1.22) as

$$\Delta^r f(0) = \sum_{i=r}^{n} a_i h^i \, \Delta^r 0^i \, .$$

Since we need all differences $\Delta^r f(0)$, it would be too wasteful to calculate them separately and re-calculate the differences of 0^i each time, just to add $\Delta^r 0^i$ at step r. Instead of running r from 1 to n and for each r running i from r to n to make the summation, we should run i from 1 to n, form all differences $\Delta^r 0^i$ up to i and then add to each $\Delta^r f(0)$ the term $a_i h^i \Delta^r 0^i$, for $r = 1$ to i. The following table is helpful:

	$a_1 h^1$	$a_2 h^2$	$a_3 h^3$	$\cdots$	$a_n h^n$
$\Delta f(0)$	1	1	1	$\cdots$	1
$\Delta^2 f(0)$		$\Delta^2 0^2$	$\Delta^2 0^3$	$\cdots$	$\Delta^2 0^n$
$\Delta^3 f(0)$			$\Delta^3 0^3$	$\cdots$	$\Delta^3 0^n$
$\vdots$				$\ddots$	$\vdots$
$\Delta^n f(0)$					$\Delta^n 0^n$

Application to Bézier curves

A Bézier curve of degree n is essentially an n^{th} degree polynomial (with vector coefficients – so in fact two polynomials, one for each coordinate); we saw it expressed with the Bernstein basis $B_i^n(t)$ as

$$\mathbf{p}^n(t) = \sum_{i=0}^{n} \mathbf{p}_i B_i^n(t)$$

with $t \in [0, 1]$. We can convert it to the more natural monomial form

$$\mathbf{p}^n(t) = \sum_{i=0}^{n} \mathbf{a}_i t^i$$

and then use the procedure involving forward differences to evaluate it, as outlined above. Here the function $f(t) = \mathbf{p}^n(t)$ will be essentially two separate functions, one for each of the coordinates x and y, and the coefficients $a_i = \mathbf{a}_i$ will be points in the 2D space.

The conversion from the Bernstein to the monomial form involves again some convoluted calculations. We will make them for the case $n = 2$, as an example, and then give the formula for the general case. So, for $n = 2$

$$\begin{aligned}
\mathbf{p}^2(t) &= \sum_{i=0}^{2} \mathbf{p}_i B_i^n(t) = \sum_{i=0}^{2} \binom{2}{i} t^i (1-t)^{2-i} \mathbf{p}_i \\
&= (1-t)^2 \mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2 \mathbf{p}_2 \\
&= (1 - 2t + t^2)\mathbf{p}_0 + 2(t - t^2)\mathbf{p}_1 + t^2 \mathbf{p}_2 \\
&= t^2(\mathbf{p}_2 - 2\mathbf{p}_1 + \mathbf{p}_0) + 2t(\mathbf{p}_1 - \mathbf{p}_0) + \mathbf{p}_0 \\
&= t^2\, \Delta^2 \mathbf{p}_0 + 2t\, \Delta \mathbf{p}_0 + \mathbf{p}_0
\end{aligned}$$

In the general case, it is not difficult to show that

$$\mathbf{p}^n(t) = \sum_{i=0}^{n} \binom{n}{i} \Delta^i \mathbf{p}_0\, t^i \tag{1.25}$$

which is the monomial form of the Bézier curve.

To summarise: In order to evaluate a Bézier curve of degree n using forward differences, we need to find the coefficients of its monomial form,

$$\mathbf{a}_i = \binom{n}{i} \Delta^i \mathbf{p}_0\ , \quad i = 0, 1, 2, \ldots, n$$

then to calculate

$$\Delta^r \mathbf{p}^n(0) = \sum_{i=r}^{n} \mathbf{a}_i h^i\, \Delta^r 0^i\ , \quad r = 0, 1, 2, \ldots, n$$

and use these to evaluate $\mathbf{p}^n(t)$ at $t = 0,\ h,\ 2h, \ldots, 1$. The intermediate values are joined together with straight line segments, and, provided that h is small enough, the result gives the impression of a smooth curve.

It should be clear that this method for drawing Bézier curves is not adaptive: points on the curve are generated for parameter values at equal distances h, even for relatively flat segments of the curve; this is a disadvantage of this method, compared with the de Casteljau algorithm, but its speed pays up as at each step it requires only a minimal

number of floating point operations. The choice of h here should take into account the steepest changes in curvature of the curve to be drawn so as to produce smooth 'corners'.

1.2.5 Additional material: convex hull of a set of 2D points

Closing the section on Bézier curves, we give the outline of an algorithm to calculate the convex hull of a set of 2D points. As mentioned earlier, the convex hull property of Bézier curves can be used to check intersection of a straight line with the curve. For this, it would be useful to have a way of calculating the convex hull of its control points.

This algorithm presented below be found in [4], section 8.3.1 and the reader is referred to this book[4] for more details. There are also a second algorithm on this topic as well as algorithms giving the convex hull for higher dimensions (which were too complicated to implement for this Project).

We note at first that the vertices of the convex hull will be considered stored in a circular doubly-linked list in counterclockwise order. So, for a point $\mathbf{p}$ in the convex hull, succ($\mathbf{p}$) will be the next point counterclockwise and pred($\mathbf{p}$) will be the next point clockwise. Moreover, for this algorithm it will be helpful to pre-sort the points with respect to some direction, for example their x coordinates.

Suppose we are given points $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$. To begin with, we can trivially construct the convex hull of the first three points; actually we might only need to exchange two of them to put them in counterclockwise order. We will use the notation $P_i = \text{conv}\{\mathbf{p}_1, \mathbf{p}_2, \ldots \mathbf{p}_i\}$ to denote the convex hull already constructed when all points up to and including the i^{th} have been examined. The problem is how to add the point $\mathbf{p}_{i+1}$ to the existing convex hull P_i, for $i = 3, \ldots, n$.

We observe that, since the points are sorted with respect to their x coordinates, $\mathbf{p}_{i+1}$ will lie outside the already constructed convex hull P_i (see Figure 1-4). Starting with the last inserted point $\mathbf{p}_i$, we scan the convex hull counterclockwise until we find a point $\mathbf{p}_t$ (*top*) such that $\mathbf{p}_{i+1}$ is for the first time to the left of the line from $\mathbf{p}_t$ to succ($\mathbf{p}_t$); and we scan the convex hull clockwise until we find a point $\mathbf{p}_b$ (*bottom*) such that $\mathbf{p}_{i+1}$ is for the first time to the right of the line from $\mathbf{p}_b$ to pred($\mathbf{p}_b$). Then we simply need to remove all the vertices between $\mathbf{p}_t$ and $\mathbf{p}_b$ and store $\mathbf{p}_{i+1}$ in the convex hull so that $\text{succ}(\mathbf{p}_b) = \mathbf{p}_{i+1} = \text{pred}(\mathbf{p}_t)$, $\text{pred}(\mathbf{p}_{i+1}) = \mathbf{p}_b$ and $\text{succ}(\mathbf{p}_{i+1}) = \mathbf{p}_t$.

---

[4] I believe it is interesting to note *how* I have been able to find this algorithm: Following the suggestion of my supervisor, I searched in the newsgroup comp.graphics.algorithms and actually found one message mentioning 'convex hull'. This contained a reference to some Web page with graphics algorithms, from where I took a program implementing an algorithm similar to this one by someone working at AT&T (always brilliant people in there!). Unfortunately, this program was written in plain C that only its author could decipher and completely lacked helping comments; it only said "This one is like algorithm 8.2 in Edelsbrunner", and that is how I reached this good book on Combinatorial Geometry!

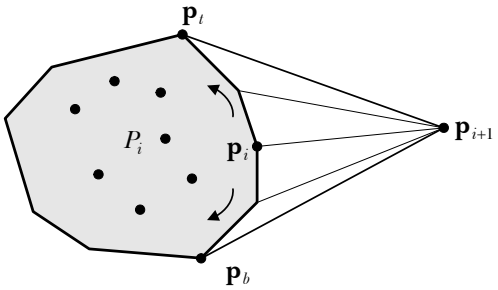

Figure 1-4: Adding a point to the convex hull.

1.3 B-Spline curves

The design of any complex shape with a Bézier curve would require many control points and thus a curve of high degree. However this approach is hardly useable, because

- The computational complexity increases greatly as the degree of the Bézier curve rises.
- The high degree polynomials involved become numerically unstable.
- Bézier curves do not have local control; this, combined with the last point, makes the manipulation of the shape of high degree curves difficult.

The method used in practice is to join together a number of low degree Bézier curves in a way that certain continuity conditions are satisfied at the joins. B-Splines are such piecewise Bézier curves.

In the following we introduce B-Spline curves, we examine their properties and present the most important techniques to manipulate them; finally we examine interpolation of a set of data points with a cubic B-Spline curve. Most of the discussion is due to [2], except for knot deletion which is my own contribution to the topic.

1.3.1 B-Spline curves using blossoms

Parametric continuity

Suppose, in general, that we have two polynomial curves in their parametric form, $F(t)$ with $t \in [t_0, t_1]$ and $G(t)$ with $t \in [t_1, t_2]$. These two curves are said to join *with* C^r *continuity at* t_1 if their r^{th} derivatives are equal at $t = t_1$:

$$F^{(r)}(t_1) = G^{(r)}(t_1). \tag{1.26}$$

It can be proved that C^r continuity implies also C^m continuity for all $0 \le m < r$. For example, if $F(t)$ and $G(t)$ join together at $t = t_1$ with C^2 continuity, then their values

(C^0), their tangents (C^1), and their second derivatives (C^2) are equal at t_1; so this join is obviously 'smooth'.[5]

For any polynomial of degree k, the k^{th} derivative is constant and all derivatives above the k^{th} are zero; so, in the case of two k^{th} degree polynomial curves joining together, it is only interesting to require up to C^{k-1} continuity.

B-Spline curves are exactly piecewise k^{th} degree Bézier curves joining together with C^{k-1} continuity. Omitting any derivation process which, for our purposes, would not add much to the discussion of the topic, we next formulate a definition of B-Spline curves. The interested reader is referred to the relevant sections in [2] for a more elaborate description.

B-Spline curves of degree k

The k^{th} degree B-Spline curve having $(n+1)$ *control points* $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$ and *knot sequence* $t_1, t_2, \ldots, t_{n+k}$ $(t_i \le t_{i+1})$ consists of $(n-k+1)$ k^{th} degree Bézier curves, each defined over an interval $[t_i, t_{i+1}]$ for $i = k, k+1, \ldots, n$; if $f(t,t,\ldots,t)$, $t \in [t_i, t_{i+1}]$ is the blossom of each Bézier curve, then the following hold:

- The B-Spline control points and the knots are connected by
$$\mathbf{v}_j = f(t_{j+1}, t_{j+2}, \ldots, t_{j+k}), \quad j = i-k, i-k+1, \ldots, i \tag{1.27}$$
- The points on the B-Spline curve are given by
$$\mathbf{p}(t) = f(t^{<k>}), \quad \text{for} \quad t \in [t_i, t_{i+1}]. \tag{1.28}$$
- The k^{th} degree Bézier curve defined over the interval $[t_i, t_{i+1}]$ has control points
$$\mathbf{p}_r = f(t_i^{<r>}, t_{i+1}^{<k-r>}), \quad r = 0, 1, \ldots, k. \tag{1.29}$$

From this definition it follows that the B-Spline curve is defined over the interval $[t_k, t_{n+1}]$. If the first k knots are equal, then $\mathbf{v}_0 = f(t_1, \ldots, t_k) = f(t_k^{<k>}) = \mathbf{p}(t_k)$, i.e. the curve starts at its first control point; similarly, if the last k knots are equal, it will end at its last control point. In general, when a knot is repeated, one degree of continuity is lost at that join; so no knot can have multiplicity higher than k.

Drawing B-Splines as piecewise Bézier curves

The definition of B-Spline curves suggests a way of drawing them: we can calculate the control points of the Bézier curve segments, given by (1.29), and draw each of these Bézier curves by one of the methods mentioned above.

For the quadratic case, we have to calculate only $f(t_i, t_i)$, $f(t_i, t_{i+1})$ and $f(t_{i+1}, t_{i+1})$. These points can be given conveniently by two de Casteljau triangles,

$$\begin{matrix} \mathbf{v}_{i-2} = f(t_{i-1}, t_i) \\ \mathbf{v}_{i-1} = f(t_i, t_{i+1}) \end{matrix} \quad \underline{f(t_i, t_i)} \quad \text{and} \quad \begin{matrix} \mathbf{v}_{i-1} = f(t_i, t_{i+1}) \\ \mathbf{v}_i = f(t_{i+1}, t_{i+2}) \end{matrix} \quad \underline{f(t_{i+1}, t_{i+1})}$$

---

[5] It can be seen that parametric continuity is too strict a condition for visually smooth joins. Another kind of continuity, *geometric continuity*, is less strict and also produces 'smooth' joins; it is used for other kinds of Splines such as ν-, β- and γ- splines. However, parametric continuity is easier to deal with mathematically. For more on geometric continuity, the reader is referred to [1], chapters 12-13.

while the middle point is already given, $f(t_i, t_{i+1}) = \mathbf{v}_{i-1}$.

For the cubic case we need $f(t_i, t_i, t_i)$, $f(t_i, t_i, t_{i+1})$, $f(t_i, t_{i+1}, t_{i+1})$ and $f(t_{i+1}, t_{i+1}, t_{i+1})$. In this case, the de Casteljau triangles are:

$$\begin{array}{lll} \mathbf{v}_{i-3} = f(t_{i-2}, t_{i-1}, t_i) & & \\ \mathbf{v}_{i-2} = f(t_{i-1}, t_i, t_{i+1}) & f(t_{i-1}, t_i, t_i) & \\ \mathbf{v}_{i-1} = f(t_i, t_{i+1}, t_{i+2}) & \underline{f(t_i, t_{i+1}, t_i)} & \underline{f(t_i, t_i, t_i)} \end{array}$$

and

$$\begin{array}{lll} \mathbf{v}_{i-2} = f(t_{i-1}, t_i, t_{i+1}) & & \\ \mathbf{v}_{i-1} = f(t_i, t_{i+1}, t_{i+2}) & \underline{f(t_i, t_{i+1}, t_{i+1})} & \\ \mathbf{v}_{i} = f(t_{i+1}, t_{i+2}, t_{i+3}) & f(t_{i+1}, t_{i+2}, t_{i+1}) & \underline{f(t_{i+1}, t_{i+1}, t_{i+1})} \end{array}$$

Unfortunately, for higher degree B-Splines the calculation of the Bézier control points is not that straightforward; also it turns out that the curve should not have knots with multiplicity equal to k (other than the end knots) because this causes a division by zero in the interpolation steps. A more general method for drawing B-Spline curves is presented next.

Evaluating B-Spline points (de Boor algorithm)

As we did in the case of the differences method for drawing Bézier curves, we can evaluate several points on the B-Spline curve and then connect them with straight lines to draw the curve. The points are given by (1.28) and again they may be calculated with repeated interpolation steps.

The *de Boor algorithm* provides the formulae for this calculation: To calculate any point $\mathbf{p}(t) = f(t, t, \ldots, t)$, $t \in [t_i, t_{i+1}]$ on the B-Spline curve, set

$$\mathbf{p}_j^0(t) = \mathbf{v}_j, \quad j = i-k, i-k+1, \ldots, i \tag{1.30}$$

and

$$\mathbf{p}_j^r(t) = \left(\frac{t_{k+1+j} - t}{t_{k+1+j} - t_{r+j}}\right)\mathbf{p}_j^{r-1}(t) + \left(\frac{t - t_{r+j}}{t_{k+1+j} - t_{r+j}}\right)\mathbf{p}_{j+1}^{r-1}(t), \quad \begin{array}{l} r = 1,2,\ldots,k \\ j = i-k, i-k+1, \ldots, i-r \end{array} \tag{1.31}$$

Then the required point on the curve is $\mathbf{p}(t) = \mathbf{p}_{i-k}^k(t)$.

It should be noted that, as the formulae above show, calculation of the de Boor points involves division with differences of many combinations of the knots. This can be dangerous if the knot sequence contains multiple knots (other than the end ones) as it can potentially lead to division by zero. However, this is not likely with the knot sequences used in practice.

1.3.2 B-Spline curve topics

Properties of B-Spline curves

B-Spline curves provide a very flexible and intuitive tool for drawing curves of arbitrary complexity.

- Their shape is reasonably predictable from the control points, as each curve segment retains all the good properties of Bézier curves.
- B-Spline curves *have* local control: a change in a control point only affects the (at most k) curve segments that use it and thus has a local effect on the curve.
- In practice, low degree B-Spline curves are used, quadratic and especially cubic ones; these are simple enough to calculate while they have enough flexibility to suit most purposes. The two previous properties further justify this point.
- The choice of knot sequence determines significantly the shape of the curve. For example, if we increase a knot value or have multiple knots, the curve is pulled over the direction of the respective control point. As a start the knot values can be chosen at equal distances, but in many cases this does not produce a satisfactory curve; other methods of *parametrisation* exist which take into account the geometry of the control points and may produce better results. (For an elaborate discussion see [1], section 9.4, and [3], section 4.4.1)

The following topics deal with manipulation of the knot sequence of a B-Spline curve.

Knot insertion (Boehm's algorithm)

Inserting a new knot into the knot sequence of a B-Spline curve while maintaining the shape of the curve is a fundamental operation. When inserting a new knot, a new control point must be added in the control points of the curve to satisfy its definition (and other should be changed to preserve its shape). This additional point clearly allows for greater flexibility in the shape of the curve. Another application of knot insertion is rendering the curve, as it can be shown that as the number of knots increases while the curve remains the same, the control points converge to the B-Spline curve itself.

So, consider adding a new knot s in the interval $[t_i, t_{i+1}]$. The knot sequence will now be $t_1,\ldots,t_i,s,t_{i+1},\ldots,t_{n+k}$. If $\mathbf{w}_i$, $i=0,1,\ldots,n+1$ are the new control points, then (1.27) gives in this case

$$\begin{aligned}
\mathbf{w}_j &= f(t_{j+1},t_{j+2},\ldots,t_{j+k}) = \mathbf{v}_j, \qquad && j=0,\ldots,i-k \\
\mathbf{w}_j &= f(t_{j+1},\ldots,t_{j+k-1},s), && j=i-k+1,\ldots,i \\
\mathbf{w}_j &= f(t_j,t_{j+1},\ldots,t_{j+k-1}) = \mathbf{v}_{j-1}, && j=i+1,\ldots,n+1
\end{aligned} \tag{1.32}$$

and thus the only unknowns are $\mathbf{w}_j$ for $j=i-k+1,\ldots,i$. Observing that $s\in[t_i,t_{i+1}]$, by interpolation we can get

$$f(t_{j+1},\ldots,t_{j+k-1},s) = \left(\frac{t_{j+k}-s}{t_{j+k}-t_j}\right) f(t_j,\ldots,t_{j+k-1}) + \left(\frac{s-t_j}{t_{j+k}-t_j}\right) f(t_{j+1},\ldots,t_{j+k})$$

or

$$\mathbf{w}_j = \left(\frac{t_{j+k}-s}{t_{j+k}-t_j}\right)\mathbf{v}_{j-1} + \left(\frac{s-t_j}{t_{j+k}-t_j}\right)\mathbf{v}_j \tag{1.33}$$

The formula (1.33) is called *Boehm's insertion formula*. There is a restriction for the interval into which insertion can be done, imposed by (1.32): i should be between k and n inclusive.

Multiple knot insertion (Oslo algorithm)

It is possible to insert many new knots in the knot sequence of a B-Spline curve and compute the new control points simultaneously. This requires less calculations than repeated single knot insertions with Boehm's algorithm.[6] Here we present a simple version of the *Oslo algorithm* which deals with the insertion of k or $k+1$ knots in the same interval $[t_i, t_{i+1}]$.

Suppose we want to add three new knots s_1, s_2 and s_3 in the interval $[t_i, t_{i+1}]$ in the case of a cubic B-Spline curve. The new knot sequence will be

$$t_1, \ldots, t_{i-2}, t_{i-1}, t_i, s_1, s_2, s_3, t_{i+1}, t_{i+2}, t_{i+3}, \ldots, t_n$$

and the new control points $\mathbf{w}_i$, $i = 0,1,\ldots,n+3$ will satisfy relations analogous to (1.32). We need to calculate the points $\mathbf{w}_{i-2} = f(t_{i-1}, t_i, s_1)$, $\mathbf{w}_{i-1} = f(t_i, s_1, s_2)$, $\mathbf{w}_i = f(s_1, s_2, s_3)$, $\mathbf{w}_{i+1} = f(s_2, s_3, t_{i+1})$ and $\mathbf{w}_{i+2} = f(s_3, t_{i+1}, t_{i+2})$, and the calculations can be arranged conveniently into two de Casteljau triangles: in the first we insert the knots in increasing order, starting from the first one,

$$\begin{array}{llll} f(t_{i-2}, t_{i-1}, t_i) & & & \\ f(t_{i-1}, t_i, t_{i+1}) & \underline{f(t_{i-1}, t_i, s_1)} & & \\ f(t_i, t_{i+1}, t_{i+2}) & f(t_i, t_{i+1}, s_1) & \underline{f(t_i, s_1, s_2)} & \\ f(t_{i+1}, t_{i+2}, t_{i+3}) & f(t_{i+1}, t_{i+2}, s_1) & f(t_{i+1}, s_1, s_2) & \underline{f(s_1, s_2, s_3)} \end{array}$$

and in the second we insert them in decreasing order, starting from the last one:

$$\begin{array}{llll} f(t_{i-2}, t_{i-1}, t_i) & & & \\ f(t_{i-1}, t_i, t_{i+1}) & f(t_{i-1}, t_i, s_3) & & \\ f(t_i, t_{i+1}, t_{i+2}) & f(t_i, t_{i+1}, s_3) & f(t_i, s_3, s_2) & \\ f(t_{i+1}, t_{i+2}, t_{i+3}) & \underline{f(t_{i+1}, t_{i+2}, s_3)} & \underline{f(t_{i+1}, s_3, s_2)} & f(s_3, s_2, s_1) \end{array}$$

This procedure generalises easily to B-Spline curves of arbitrary degree k. The same restrictions for the interval into which the knots are inserted apply in this case too, as for single knot insertion.

Knot deletion

Knot deletion is not mentioned in any of the referenced texts on CAGD, possibly because it is considered of limited practical use: knot insertion is necessary to give the designer more freedom in the shape of the curve, so why would anyone want less freedom?

However, in the context of an interactive computer program the designer may insert too many knots which (s)he later discovers are not necessary to obtain the required shape; (s)he then should have the ability to remove the ones in excess, thus also reducing the complexity of the curve and of the required calculations.

One straightforward way of removing a knot is to reverse the Boehm's insertion formula. This approach works very well for a newly inserted knot as it obviously does not alter the shape of the curve. Unfortunately the results are not so good in other cases.

---

[6] However, it was pointed out to me by Mel Slater that tests have shown the gain not to be significant in practice.

Suppose we are removing knot t_i from the knot sequence $t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_{n+k}$ of a k^{th} degree B-Spline; then relations (1.32) solved backwards (after some necessary changes in indices and exchanging $\mathbf{v}_i$ and $\mathbf{w}_i$ so that the latter are again the final points after the knot removal) give the following:

$$\begin{aligned}
&\mathbf{w}_j = \mathbf{v}_{j+1}, && j = n, n-1,\ldots,i-1\\
&\mathbf{w}_j = \left(\frac{t_{j+k+2}-t_{j+1}}{t_{j+k+2}-t_i}\right)\mathbf{v}_{j+1} + \left(\frac{t_{j+1}-t_i}{t_{j+k+2}-t_i}\right)\mathbf{w}_{j+1}, && j = i-2, i-3,\ldots,i-k\\
&\mathbf{w}_j = \mathbf{v}_j, && j = i-k-1,\ldots,1,0
\end{aligned} \tag{1.34}$$

The knot to be removed should have index i between $k+1$ and $n+1$ inclusive, as in these positions it may have been inserted using the inverse procedure. Also this knot should not be the first of (at least) three equal knots as in this case the second of (1.34) will attempt division by zero.

1.3.3 B-Spline curves using the B-Spline basis

Just like Bézier curves, B-Spline curves can also be represented as a weighted sum of their control points. So, a k^{th} degree B-Spline curve $\mathbf{p}(t)$ with control points $\mathbf{v}_i$, $i = 0,1,\ldots,n$ and knot sequence t_i, $i = 1,2,\ldots,n+k$ can be written in the form

$$\mathbf{p}(t) = \sum_{i=0}^{n} N_i^k(t)\,\mathbf{v}_i, \tag{1.35}$$

with $N_i^r(t)$ defined recursively as

$$N_i^0(t) = \begin{cases} 1, & \text{if } t_i \le t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

and

$$N_i^r(t) = \left(\frac{t-t_i}{t_{i+r}-t_i}\right)N_i^{r-1}(t) + \left(\frac{t_{i+r+1}-t}{t_{i+r+1}-t_{i+1}}\right)N_{i+1}^{r-1}(t), \quad \text{for } r > 0$$

The $N_i^r(t)$ defined above are a basis of the vector space of n^{th} degree polynomials defined over $[t_i, t_{i+n+1}]$. Every $N_i^r(t)$ is an r^{th} degree polynomial with *local support* on the interval $[t_i, t_{i+r+1}]$, i.e. it is non-zero only over that interval; it consists of $r+1$ polynomial segments joined together with C^{r-1} continuity.

The blossoming approach to B-Splines can be derived from (1.35) and vice versa, and all the aforementioned properties of B-Spline curves can be shown by studying the B-Spline basis functions. However, we omit any treatment of this topic as it would not add much to our discussion. The reader is referred for more information to Chapter 10 of [1], to the relevant sections in [2] and to section 4.3 of [3]; in fact, this last approach to B-Splines is the 'traditional' one and thus preferred by many books.

1.3.4 Cubic B-Spline interpolation

Bézier and B-Spline curves as described above *approximate* their control points. However, in practice it is more likely to seek for a curve which *interpolates* a given (and often large)

set of points. In the following we discuss interpolation with cubic B-Splines, which are mostly suitable and thus a common choice for this purpose.

The problem can be formulated as follows: Given a set of *data points* $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n$ and a set of knots t_i (the range of i to be determined later), find a cubic B-Spline curve with the given knot sequence which interpolates the given points $\mathbf{p}_i$.

If $f(t_1, t_2, t_3)$ is the blossom of the required B-Spline curve, then the $\mathbf{p}_i$ will be points on the curve and so they will satisfy $\mathbf{p}_i = f(t_i, t_i, t_i)$. This requires knots $t_1, t_2, \ldots, t_n$ which should be different as long as the $\mathbf{p}_i$ are different. Moreover, we obviously want $\mathbf{p}_1$ to be the starting point of the curve and $\mathbf{p}_n$ to be its ending point; for this we need the first and the last three knots equal, so we add knots $t_{-1} = t_0 (= t_1)$ and $(t_n =) t_{n+1} = t_{n+2}$.

From these elements, we need to derive the control vertices $\mathbf{v}_k = f(t_{k+1}, t_{k+2}, t_{k+3})$ of the curve. The range for k is defined by the one for i: the $(n+4)$ knots defined above can generate $n+2$ control points with $k = -2, -1, \ldots, n-1$. Immediately,

$$\mathbf{v}_{-2} = f(t_{-1}, t_0, t_1) = f(t_1, t_1, t_1) = \mathbf{p}_1 \quad \text{and} \quad \mathbf{v}_{n-1} = f(t_n, t_{n+1}, t_{n+2}) = f(t_n, t_n, t_n) = \mathbf{p}_n \tag{1.36}$$

Furthermore, interpolation gives us the de Casteljau triangle:

$$\begin{array}{lll} \mathbf{v}_{i-3} = f(t_{i-2}, t_{i-1}, t_i) & & \\ \mathbf{v}_{i-2} = f(t_{i-1}, t_i, t_{i+1}) & f(t_{i-1}, t_i, t_i) & \\ \mathbf{v}_{i-1} = f(t_i, t_{i+1}, t_{i+2}) & f(t_i, t_{i+1}, t_i) & f(t_i, t_i, t_i) = \mathbf{p}_i \end{array}$$

which yields:

$$\frac{(t_{i+1} - t_i)^2}{t_{i+1} - t_{i-2}} \mathbf{v}_{i-3} + \left[\frac{(t_{i+1} - t_i)(t_i - t_{i+2})}{t_{i+1} - t_{i-2}} + \frac{(t_i - t_{i-1})(t_{i+2} - t_i)}{t_{i+2} - t_{i-1}} \right] \mathbf{v}_{i-2} + \frac{(t_i - t_{i-1})^2}{t_{i+2} - t_{i-1}} \mathbf{v}_{i-1} = (t_{i+1} - t_{i-1}) \mathbf{p}_i$$

or

$$\alpha_i \mathbf{v}_{i-3} + \beta_i \mathbf{v}_{i-2} + \gamma_i \mathbf{v}_{i-1} = (t_{i+1} - t_{i-1}) \mathbf{p}_i . \tag{1.37}$$

Relations (1.37) hold for $i = 2, 3, \ldots, n-1$; so, together with (1.36) they provide n equations referring to all the unknown control points except for $\mathbf{v}_{-1}$ and $\mathbf{v}_{n-2}$. There is no other way to make an equation incorporating these control points, so we must choose these arbitrarily, say

$$\mathbf{v}_{-1} = \mathbf{r}_1 \quad \text{and} \quad \mathbf{v}_{n-2} = \mathbf{r}_2 . \tag{1.38}$$

This may seem odd, but in fact it is for good as it gives us two degrees of freedom to play around with the shape of the curve near the end points.[7]

Relations (1.36), (1.37) and (1.38) make up the following system for the calculation of the control points $\mathbf{v}_i$ from the known points $\mathbf{p}_i$ and $\mathbf{r}_i$:

[7] For ways of defining these arbitrary points, see section 3.3 of [3].

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & \gamma_2 & 0 & \cdots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ 0 & \cdots & 0 & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}_{-2} \\ \mathbf{v}_{-1} \\ \mathbf{v}_0 \\ \vdots \\ \mathbf{v}_{n-3} \\ \mathbf{v}_{n-2} \\ \mathbf{v}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{r}_1 \\ (t_3 - t_1)\mathbf{p}_2 \\ \vdots \\ (t_n - t_{n-2})\mathbf{p}_{n-1} \\ \mathbf{r}_2 \\ \mathbf{p}_n \end{bmatrix} \tag{1.39}$$

This is a tridiagonal system on $\mathbf{v}_i$ which can be solved most efficiently by a direct method such the LU-analysis, in which case the resulting matrices are of very simple form.

1.4 Surfaces

Having studied curves in 2D in adequate detail, we move on to discuss curved surfaces in 3D. Fortunately, all the concepts mentioned above for curves generalise easily to be applied to surfaces; so, after the general idea is given, only a brief description of the topics will follow.

1.4.1 Parametric form of surfaces

The parametric equation of a surface in 3D, analogously to (1.5), is

$$\mathbf{p}(t,u) = \big(x(t,u), y(t,u), z(t,u)\big) \tag{1.40}$$

with t and u in specified intervals, often in $[0,1]$. It is not difficult to see that this formula actually describes a surface. If we fix t, it reduces to a function of u which traces a curve in 3D, just as (1.5) traced a curve in 2D; then if we start with any point on this curve and let t vary, we will trace another curve in 3D. All these curves together form a surface, the one given by (1.40).

1.4.2 Bézier surfaces

To generate a Bézier surface as described above, we will obviously need a rectangular array of control points, in the place of the sequence of control points for a single curve:

	$u \rightarrow$			
t	$\mathbf{p}_{00}$	$\mathbf{p}_{01}$	$\cdots$	$\mathbf{p}_{0n}$
$\downarrow$	$\mathbf{p}_{10}$	$\mathbf{p}_{11}$	$\cdots$	$\mathbf{p}_{1n}$
	$\vdots$	$\vdots$		$\vdots$
	$\mathbf{p}_{m0}$	$\mathbf{p}_{m1}$	$\cdots$	$\mathbf{p}_{mn}$

Each row and column of this array defines a Bézier curve in 3D, so all together they define a surface, called *tensor product Bézier surface*. The formula here is

$$\mathbf{p}^{m,n}(t,u) = \sum_{i=0}^{m} B_i^m(t) \left(\sum_{j=0}^{n} B_j^n(u) \mathbf{p}_{ij} \right). \tag{1.41}$$

This form of the Bézier surface conforms to our earlier description of the generation of surfaces. Also, all the properties of Bézier curves hold for Bézier surfaces, too, suitably rephrased for 3D. We pay more attention to the algorithms for drawing these surfaces.

The de Casteljau algorithm for surfaces

In the case of curves, the de Casteljau algorithm was splitting the curve recursively for a given parameter value until the control points of a curve segment were on a straight line. The generalisation to surfaces is as follows: Split the surface recursively into *four* sub-surfaces of $m \times n$ control points each, until all control points of a sub-surface are on the same plane (the one defined by three of the points at the corners of the array); then display this plane. It is obvious that this method of description and rendering of surfaces produces *rectangular patches*.[8]

The tests for coplanarity should again be accurate within some tolerance level; they may also incorporate tests for colinearity of the points on the sides of the plane, which may lead to easier rejections when appropriate and also more accurate drawing.

The splitting of the initial array of control points to the four arrays of control points of the sub-surfaces is characteristic of the technique applied in many case for surfaces: First, we split each row of points, using the traditional de Casteljau algorithm, just as in 2D; we end up with two new arrays of $m \times n$ points. Then we split each of (all) the new columns, using again the familiar de Casteljau algorithm; now we end up with four arrays of $m \times n$ points, which are exactly the control points of the sub-surfaces.

Coplanarity test

The colinearity test presented above generalises easily to give a coplanarity test for points in 3D. In this case we are interested in checking if all control points of a Bézier surface lie on the same plane.

Since any plane is defined by three points, we arbitrarily choose three of the four points at the corners of the array and label them $\mathbf{p}_i = (x_i, y_i, z_i)$ for $i = 1,2,3$. The plane through these three points is given by

$$\begin{vmatrix} x & y & z & 1 \\ x_0 & y_0 & z_0 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \end{vmatrix} = 0 \tag{1.42}$$

---

[8] Bézier surfaces can be alternatively be represented and rendered as *triangular patches*. This representation is mathematically simpler than the one of (1.41) and can easier describe complex curved surfaces. The adaptation from 2D curves to surfaces is not so straightforward though. The reader is referred for an introduction to [2] and for more details to chapter18 of [1] and section 6.3 of [3].

or equivalently by $Ax + By + Cz + D = 0$. The distance of an arbitrary point (x, y, z) from this plane is given by

$$d = \frac{Ax + By + Cz + D}{\sqrt{A^2 + B^2 + C^2}} \tag{1.43}$$

provided that $\sqrt{A^2 + B^2 + C^2}$ (when the three points do not coincide) or by

$$d' = \sqrt{(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2} \tag{1.44}$$

otherwise. It can be seen from the formulae exposed here that coplanarity tests carry most of the computational cost of the de Casteljau algorithm. (There exist most efficient methods of performing them, which are however outside of the scope of this Project.)

Drawing Bézier surfaces using differences

The differences method described earlier for curves in 2D can also be used to draw surfaces. Now the values of $\mathbf{p}^{m,n}(t, u)$ are needed, for t and u at steps of h_t and h_u respectively. Relation (1.41) shows that the inner sum should first be evaluated (using the familiar differences method) to calculate $\mathbf{p}^{m,n}(t, u)$ for $u = 0, h_u, 2h_u, \ldots, 1$; having done this, the outer sum can be calculated, thus obtaining the desired values of $\mathbf{p}^{m,n}(t, u)$ for $u = 0, h_u, 2h_u, \ldots, 1$ and for $t = 0, h_t, 2h_t, \ldots, 1$. This means to work first on each row of the array of control points, store the generated points in a new array and then work on each of its columns. The final points provide a rectangular mesh which is our Bézier surface.

1.4.3 B-Spline surfaces

B-Spline surfaces present no more difficulty. They are piecewise tensor product Bézier surfaces, and thus their rendering is essentially one of Bézier surfaces. B-Spline surfaces also have a rectangular array of control points, and they have independent knot sequences and independent degrees for t and u. Schematically,

	u_1	u_2	$\cdots$	u_{n+1}	$\cdots$	u_{n+k}
t_1	$\mathbf{v}_{00}$	$\mathbf{v}_{01}$	$\cdots$	$\mathbf{v}_{0n}$		
t_2	$\mathbf{v}_{10}$	$\mathbf{v}_{11}$	$\cdots$	$\mathbf{v}_{1n}$		
$\vdots$	$\vdots$	$\vdots$		$\vdots$		
t_{m+1}	$\mathbf{v}_{m0}$	$\mathbf{v}_{m1}$	$\cdots$	$\mathbf{v}_{mn}$		
$\vdots$						
t_{m+l}						

Now all properties and procedures mentioned for 2D B-Spline curves generalise straightforwardly for the surfaces. To find the Bézier points corresponding to each interval $[t_i, t_{i+1}] \times [u_k, u_{k+1}]$ we work first on the rows and then on all the new columns; the same procedure applies to the calculation of points on the B-Spline surface using the de Boor algorithm; knot insertion and deletion can be performed separately for rows and columns provided we update accordingly all the rows/columns of the array of control points.

1.4.4 Interpolation

Finally, interpolation with bi-cubic B-Spline surfaces of a given array of $m \times n$ data points in also simple. We can interpolate first for t, by columns, and then for u, using all the new rows. The arbitrary points needed for all the columns form two additional rows $\mathbf{r}1$ and $\mathbf{r}2$, and those needed for the rows form two additional columns $\mathbf{c}1$ and $\mathbf{c}2$; to complete the scheme, we need four more points at the intersections of the new rows and columns. The following table shows all the points needed for the right-hand sides of the systems:

$$
\begin{array}{cccccccc}
\mathbf{p}_{00} & \mathbf{c}1_0 & \mathbf{p}_{01} & \cdots & \mathbf{p}_{0,n-1} & \mathbf{c}2_0 & \mathbf{p}_{0n} \\
\mathbf{r}1_0 & \mathbf{r}1\mathbf{c}1 & \mathbf{r}1_1 & \cdots & \mathbf{r}1_{n-1} & \mathbf{r}1\mathbf{c}2 & \mathbf{r}1_n \\
\mathbf{p}_{10} & \mathbf{c}1_1 & \mathbf{p}_{11} & \cdots & \mathbf{p}_{1,n-1} & \mathbf{c}2_1 & \mathbf{p}_{1n} \\
\vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
\mathbf{p}_{m-1,0} & \mathbf{c}1_{m-1} & \mathbf{p}_{m-1,1} & \cdots & \mathbf{p}_{m-1,n-1} & \mathbf{c}2_{m-1} & \mathbf{p}_{m-1,n} \\
\mathbf{r}2_0 & \mathbf{r}2\mathbf{c}1 & \mathbf{r}2_1 & \cdots & \mathbf{r}2_{n-1} & \mathbf{r}2\mathbf{c}2 & \mathbf{r}2_n \\
\mathbf{p}_{m0} & \mathbf{c}1_m & \mathbf{p}_{m1} & \cdots & \mathbf{p}_{m,n-1} & \mathbf{c}2_m & \mathbf{p}_{m,n}
\end{array}
$$

2. SOFTWARE ENGINEERING

In this chapter are discussed the Software Engineering issues of this Project. A separate section in the beginning provides a description of GLOOP, a graphics package used as the background of the classes implementing curves and surfaces. Then follow sections on Requirements, Design, Implementation and Testing of the software developed.

A good knowledge of C++, the language that was used for coding, is assumed throughout, and particularly of its features related to class hierarchies. Stroustrup [8] provides a complete treatment of the subject.

2.1 GLOOP

The software constructed for this Project was built as an extension to GLOOP. GLOOP is a Graphics Language for Object Oriented Programming developed at UCL and used for the Graphics courses within the Department of Computer Science. This choice was made for the following reasons:

- GLOOP already provides ways of defining and drawing the basic geometric primitives such as lines and polygons in 2D and 3D. Thus the work for this Project could concentrate on its objective and would not need to deal with the implementation of lower-level constructs.
- GLOOP is written in C++ and uses many of its advanced features. Building new classes upon the existing ones seemed a good way to practise the language.
- The full source code of the package is available, and this could further help familiarisation with the language and with the package itself.
- Finally, GLOOP had been used during the year and acquaintance with its features and structure was gained; this would minimise any adaptation period and allow more time for the core work of the Project.

In this section we describe the facilities provided by GLOOP and pay attention to its object model, which guided the structure of the classes implementing Bézier and B-Spline curves and surfaces. For a complete presentation of the package, the reader is referred to [10].

2.1.1 GLOOP features

GLOOP is a set of classes that allow the C++ programmer to manipulate basic geometric primitives in two and three dimensions. More specifically, it provides:

- A way of creating and drawing points, lines, polypoints (sets of points treated together), polylines (sets of lines treated together), polygons, rectangles (only in 2D), circular and elliptical arcs (only in 2D), circles and ellipses (only in 2D). In 3D it also allows the representation of arbitrary objects using Boundary Representation, Constructive Solid Geometry or Voxels.
- A way of specifying the drawing attributes of objects, such as line style, line width, whether the edges and/or the interior (when applicable) are to be drawn, line colour and fill colour (when applicable).
- A means of transforming graphical entities, either by standard transformations such as translation, scaling, rotation, reflection or shear, or by a general transformation specified by an appropriate matrix.
- A way of grouping together graphical entities into hierarchies so that they can be transformed or drawn all at once.
- The capability of specifying multiple viewports on the screen showing different parts of the world. In 3D these features are expanded to include specification of the camera and of the method of projection used for the display of 3D entities.

The display of drawings on the screen is not done by GLOOP itself. The package only outputs commands in textual form which can be understood by an individual program that performs the drawing. This approach maximises portability of the graphics classes, provided that the drawing program can be ported to other systems; it also allows for other forms of output (for example Postscript) to be generated as long as suitable translators exist.

2.1.2 The object model of GLOOP

After this brief presentation of the features provided by GLOOP, we now examine the hierarchy of its classes. Here we will only cover the points that are necessary for the understanding of the classes built for this Project.

Every graphics primitive (e.g. point, line, etc.) has in GLOOP two counterparts: the *geometric* and the *drawable* one. As far as naming in concerned, the geometric primitive is prefixed by a `G` whereas the drawable primitive has no prefix; they both have a suffix indicating whether they are 2D or 3D primitives.

We deal mostly with the 2D primitives because their structure is quite simple; this suffices for an overall understanding of the GLOOP model. The 3D primitives follow the same general pattern although the class hierarchies in this case are much more convoluted.

Geometric primitives

The geometric primitive carries only the geometry of the entity and cannot be drawn. Member functions of a geometric primitive class provide all the elements of the entity, for example the number of points in a `GPolyPoint2d` and their coordinates, and they cater for any possible operation on the entity, such as addition of a new point in a `GPolyPoint2d`.

All geometric primitives inherit an instance of the abstract base class `GPrimitive2d`, which caters for the transformation of the entity. This approach allows for uniform manipulation of geometric primitives of different types.

Drawable primitives

A drawable primitive has as base class the respective geometric primitive. Thus it inherits the geometry, access and other member functions of the geometric primitive, which consequently refer and may applied directly to the drawable primitive.

Additionally, a drawable primitive inherits an instance of the class `Primitive2d`. This is derived from the class `Attributes`, which caters for the drawing attributes of the entity, and from the class `Object2d` which is an abstract base class defining the prototypes of functions for drawing the primitives. This again allows for uniform treatment of drawable primitives of different types.

Other GLOOP classes

For reference only, we give short descriptions of other important classes defined in GLOOP and used throughout the classes constructed.

- `Trans2d`, `Trans3d`: Transformation classes to handle standard and general transformations of the graphical entities; these are in fact derived classes such as `TranslateTrans2d`, `GeneralTrans3d`, etc.
- `View2d`, `View3d`: These classes specify the viewport and drawing window. In 3D the `View3d` class is used in conjunction with the `Camera` class which defines the camera and method of projection of 3D objects into 2D.
- `BRep`: A class to display (there is no geometric counterpart) objects in 3D given their boundary representation. In its current implementation, the object is considered as being comprised of four-sided polygons; it is described as a list of its vertices and a list of pointers to these, pointing by four to the vertices of each polygon. `BRep`s can be shaded and the appropriate lighting model can be specified using the class `Lighting_model`.

2.2 Requirements

This section provides the requirements that guided the development of the software built for this Project. It should be read in conjunction with the relevant theory presented in Chapter 1.

Following the usual practice, *functional* and *non-functional* requirements are distinguished.

2.2.1 Functional requirements

To quote Sommerville [7], functional requirements "are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do".

So, the functional requirements for this Project are the following:

1. **System definition**
   a) The system is to be a programming interface for Bézier and B-Spline curves and surfaces.
   b) In brief, the system should implement all the theoretical results presented in Chapter 1.
   c) No other functionality is to be expected, such as end-user programs for interactive manipulation of the curves and surfaces.
2. **Bézier curves**
   a) The user (programmer) should be able to create arbitrary Bézier curves in multiple ways from their control points, their coordinates etc.
   b) The user should be able to inspect the Bézier curve and acquire the number of control points and the control points themselves.
   c) Facilities of finding the minmax box and the convex hull of the curve should be provided.
   d) The user should have a means of subdividing the curve for a given parameter value and of checking it for flatness within a given tolerance level. These facilities are primarily used in the de Casteljau algorithm but may have independent uses as well.
   e) The user should have a means of transforming the Bézier curve by arbitrary geometric transformations.
   f) The user should be able to draw the Bézier curve using either the de Casteljau algorithm or the differences method; in each case, he should have control over the parameter specifying the accuracy of drawing.
   g) The user should have the ability to specify the drawing attributes of the curve and change the viewport and drawing window coordinate systems.

3. B-Spline curves

a) The user should be able to create arbitrary B-Spline curves in multiple ways from their control points, their coordinates, their knot sequence etc. The system should check the elements of the B-Spline curve for consistency (for example that the knots suffice for the required curve, that the knots are non-decreasing) and display suitable error messages.

b) The user should be able to inspect the B-Spline curve and acquire the number of control points, the control points, the number of knots and the knot sequence.

c) The user should have a means of inserting one or many knots at a time in the knot sequence of the curve. The system should ensure that insertion is only attempted in an allowed interval. The user should also be able to delete a knot from the sequence. Again the system should check that an allowed knot is removed.

d) Facilities of interpolating a set of given data points with a cubic B-Spline curve should be provided. The user should be free to specify the arbitrary points needed for the process.

e) The user should have a means of transforming the B-Spline curve by arbitrary geometric transformations.

f) The user should be able to draw the B-Spline curve either as a piecewise Bézier curve or by finding the de Boor points on it; for the first case he should have a choice in the drawing method of the Bézier curve; in each case, he should have control over the parameter specifying the accuracy of drawing. The system should make all relevant checks for applicability of the drawing method required and warn accordingly.

g) The user should have the ability to specify the drawing attributes of the curve and change the viewport and drawing window coordinate systems.

4. Bézier surfaces

a) The requirements for Bézier curves also apply to surfaces, suitably adapted. In the following only the differences from the curve case are mentioned.

b) The user should have access to the number of control points for parameters t and u separately.

c) The convex hull of the Bézier surface need not be provided.

d) The parameter values for subdivision should be distinct for t and u.

e) The same applies to the evaluation steps for t and u for the differences drawing method.

f) The user should have the ability to render the surface either as a simple wireframe or shaded. In the latter case he should be able to specify the lighting parameters.

5. **B-Spline surfaces**
   a) The requirements for B-Spline curves also apply to surfaces, suitably adapted. In the following only the differences from the curve case are mentioned.
   b) The user should have access to the number of control points, number of knots and knot sequences for parameters t and u separately.
   c) Knot insertion and deletion should be provided for both parameters t and u.
   d) The parameters specifying the accuracy of drawing should be distinct for t and u when applicable.
   e) The user should have the ability to render the surface either as a simple wireframe or shaded. In the case he should be able to specify the lighting parameters.

2.2.2 Non-functional requirements

Borrowing again from Sommerville, non-functional requirements are "constraints imposed on the software and restrictions on the freedom of the designer". They "might include details of specific data representation, response time and memory requirements, and so on. Product and process standards which must be followed should be specified".

For the software of this Project, non-functional requirements are closely related to the choice of tools for its implementation.

1. **C++**
   a) The software will be written in C++. It will make use and benefit from its advanced features, mainly the ones related to class hierarchies.
   b) Compatibility should be kept in mind during the development of the software. Unfortunately the C++ standard is still evolving and different versions of compilers are in use which support different 'versions' of evolving features; these features should be used with caution. In addition, more advanced language features such as templates, for which no standard implementation exists yet, should best be avoided.
2. **GLOOP**
   a) GLOOP will provide the background for the classes implementing Bézier and B-Spline curves and surfaces, so these classes will make use of its capabilities.
   b) That said, the classes built will have to use existing GLOOP features when possible and to abide by standard GLOOP rules as far as naming, structure and inheritance are concerned. This includes provision of standard forms of several functions, usually because they are necessary as definitions of virtual functions in abstract base classes.
   c) In this context, it is also important that maximum interoperability with other GLOOP classes is provided.

3. **Other system constraints**
   a) There are no constraints as far as the size of the data structures is concerned. The upper limit is determined by the available resources of the machine into which the system runs.
   b) Every effort should be made to produce efficient code; however this should not be against the clarity of the code written.

2.3 Design

The requirements exposed above form the basis for the design of the software built for this Project. The following sub-sections present the design considerations of the classes implementing curves and surfaces; more specifically, they present *what* the classes provide, whereas a discussion of *how* it is provided to the user/programmer is left up to the section on Implementation.

It should be noted at this point that the overall model of the system existed prior to this Project; in fact it is one of the requirements that the classes built should integrate well with GLOOP. Consequently, any design decisions had to concern the high level onto which the new classes stand within GLOOP and no lower-level design had or needed to be done.

2.3.1 Classes for Bézier curves

Following (and obeying to) GLOOP structure, two separate classes were required for each geometric entity. The ones for Bézier curves were named, quite naturally, `GBezier2d` and `Bezier2d`.

`GBezier2d` class

As mentioned earlier, this class should hold the geometry of the Bézier curve and should be responsible for any manipulation applicable to the curve as a geometric primitive.

The theory exposed in section 1.2 suggests that any Bézier curve is defined completely by its control points; thus, the data members of the `GBezier2d` class should be exactly the number and coordinates of its control points.

Functions that concern the Bézier curve as a purely geometric entity deal with:

- Finding its minmax box.
- Finding its convex hull,
- Subdividing it into two Bézier curves of the same degree.
- Checking the curve for flatness within a given tolerance level.
- Transforming the curve by some geometric transformation.

All these facilities should be provided as member functions of this class. In addition, it will be necessary to have member functions to inquire the elements of the curve, i.e. the number of its control points and their coordinates.

`Bezier2d` *class*

The `Bezier2d` class should treat the Bézier curve as a drawable primitive. Being derived from `GBezier2d`, this class should complement it with the necessary information and functions for drawing the curve.

It is clear that since two drawing methods are to be implemented, there should be a way of distinguishing which one the user has chosen. This information could conveniently be held in a state variable of the class. Furthermore, each method provides the user a means of controlling the accuracy/smoothness of drawing in order to attain a required drawing speed: for the de Casteljau algorithm this is the tolerance used in the flatness tests and for the differences method it is the step h at which the function describing the curve is evaluated. These should also consist state variables of the class. Although a single floating point number is used in any case, it would be more natural to have two separate parameters and use them appropriately, according to the drawing method chosen.

Member functions should allow all these parameters to be set by the user. Obviously, they should also provide the interface for drawing the curve using the specified method and parameter. As far as the attributes of the curve are concerned, these are specified, following GLOOP practice, during its construction; more on this in the section on implementation.

Borrowing from Sommerville, we formalise the design resolutions outlined above in the next diagrams (Figure 2-1). At the top of each diagram is the name of the class, then follow the data members of this class and in the last part are shown the member functions.

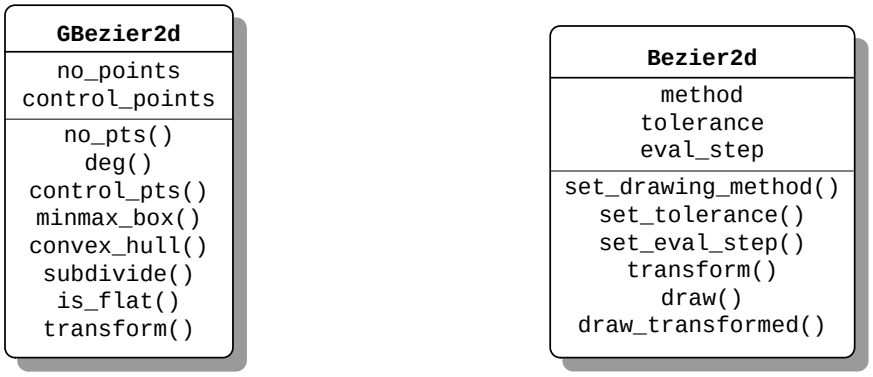

Figure 2-1: Outline of classes implementing Bézier curves.

2.3.2 Classes for B-Spline curves

Things are not much different for B-Spline curves. The two classes implementing them would conveniently be named `GBSpline2d` and `BSpline2d`, and the functions provided by them should be as follows.

`GBSpline2d` class

This will treat the geometric primitive of a B-Spline curve.

The defining elements of the B-Spline curve are: the degree of the Bézier curve segments, the number and coordinates of control points, and the number and values of the knots. All these should be the data members of the class.

As far as the member functions are concerned, these should first cater for inspection of the elements of the B-Spline curve. Then, all the following operations have only to do with the geometric properties of the curve and should be provided within this class:

- Knot insertion in a specified interval.
- Multiple knot insertion in a specified interval.
- Deletion of a specified knot.
- Transformation of the curve by some geometric transformation.

Finally, *interpolation* with a cubic B-Spline can be provided as a member function of the `GBSpline2d` class; it is, after all, a way of specifying such a curve.

`BSpline2d` class

The `BSpline2d` class will provide the necessary functionality for drawing B-Spline curves.

Just as for Bézier curves, we are to have two different algorithms of drawing B-Splines: as piecewise Bézier curves and by the de Boor algorithm (however, as in the theory, the first method is restricted to quadratic and cubic B-Splines). Again, the class should have a state variable to hold the method chosen. An additional state variable is needed to hold the drawing method chosen for the Bézier segments (cf. the relevant subsection above). Other state variables should store the parameters specifying the accuracy of drawing: for the de Boor method this refers to the evaluation step h at which points on the curve are calculated; for the Bézier method, this depends on the method used, as outlined above. A separate variable may be used for each of these parameters for reasons of clarity.

Member functions of the class `BSpline2d` should allow the user to specify these parameters. Other member functions should deal with the drawing of the B-Spline curve using the method and parameter specified.

The two classes are outlined in the following diagrams.

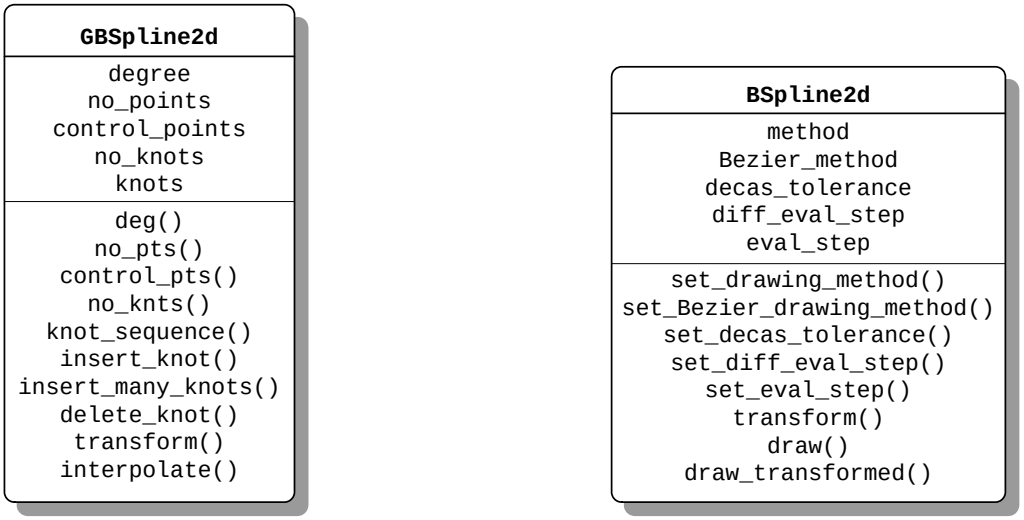

Figure 2-2: Outline of classes implementing B-Spline curves.

2.3.3 Bézier surfaces

In 3D, surfaces are essentially a straightforward generalisation of the relevant concepts for curves in 2D and so they do not present much difficulty. The discussion in this and in the following sub-section will be shorter than the one for curves, taken into account that most of the details exposed there also hold for surfaces.

For Bézier surfaces, the two classes needed should be `GBezier3d` and `Bezier3d`.

`GBezier3d` class

Following the usual practice, this class should hold the geometry of the Bézier surface.

The data members of this class should be the (rectangular array of) control points of the Bézier surface and the number of rows and columns of the array.

Member functions should be analogous to the ones for curves, only with the exception of the convex hull which is not required for surfaces.

`Bezier3d` class

The `Bezier3d` class which caters for the drawing of the Bézier surface is very similar to the `Bezier2d` class described above.

The data members should again specify the drawing method and its parameters. This time for the de Casteljau method the parameter is just the tolerance for the coplanarity tests whereas for the differences method two different evaluation steps for t and u may be held. [In this case, the parameters for the two different methods could not be represented by the same structure, for example a single floating point number, as happened for curves; this, in part, guided the choice for separate state variables in the `Bezier2d` class.]

Member functions again allow for setting of the drawing parameters and cater for drawing of the surface. In this case, separate drawing functions (or rather overloaded

functions with the same name) should be needed for drawing the surface as a wireframe or shaded; in the latter case the lighting model needs to be specified.

The classes for Bézier surfaces are outlined in the following diagrams. It should be noted that some member functions not mentioned in the discussion above are necessary because of the abstract base classes used.

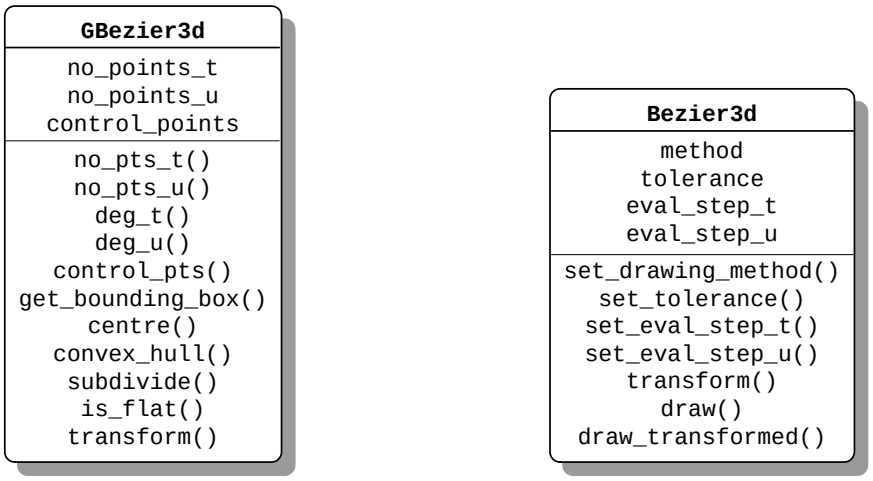

Figure 2-3: Outline of classes implementing Bézier surfaces.

2.3.4 B-Spline surfaces

The design of the classes implementing B-Spline surfaces should be obvious by now. The two classes `GBSpline3d` and `BSpline3d` are next presented in brief.

`GBSpline3d` class

This class treats the B-Spline surface as a geometric primitive.

The data members of this class should be the degrees of the Bézier sub-surfaces, for t and u separately, the array of control points of the B-Spline surface and the knot sequences for t and u.

Member functions should be analogous to the ones for curves, with the difference of being applicable to rows and columns separately when this is the case.

`BSpline3d` class

The `BSpline3d` class caters for the drawing of the B-Spline surface.

The data members should again specify the drawing method and its parameters. This time for the de Boor method two parameters are needed to hold the evaluation steps for t and u separately. For the case in which the surface is drawn as a piecewise Bézier surface, all the parameters mentioned in the relevant section are required.

Just as for the `Bezier3d` class, member functions allow the user to set the drawing parameters and to draw the surface. Different (or overloaded) functions should cater for the drawing of the surface as a wireframe or shaded, in which case the lighting model should be specified.

Figure 2-4 depicts the classes for B-Spline surfaces.

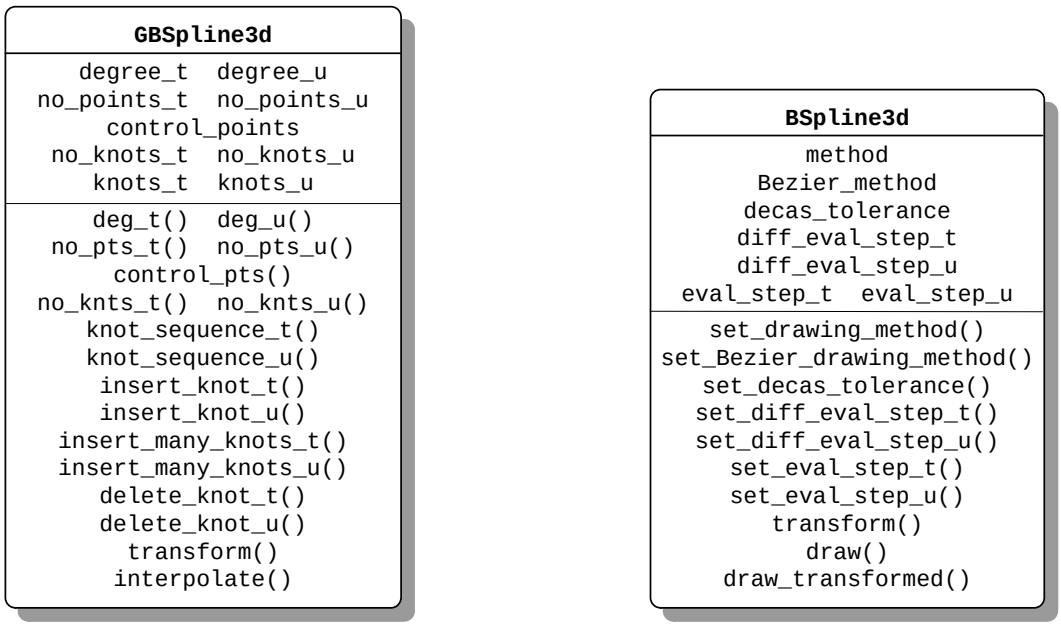

Figure 2-4: Outline of the classes implementing B-Spline surfaces.

2.3.5 Arrays of points

Bézier and B-Spline surfaces are defined by means of rectangular arrays of control points in 3D. To define and manipulate such entities in C++ we would need to use constructs like `Gpoint3d**`. Little knowledge of the language suffices to convince us that this would be both hard to deal with and prone to errors.[1] So, it seems natural to design and implement a separate class to deal with matrices of `Gpoint3d`s. All the complex pointer manipulations would be written once and could be tested independently of the other classes (and thus more easily and safely).

A descriptive (but rather lengthy) name for such a class is `GPoint3dArray`. This class should hold as data members the number of rows and columns of the array and its elements (`Gpoint3d`s).

Member functions should enable the programmer to inspect these elements. Moreover, is seems convenient to have direct access to each element of the matrix, by specifying its row and column, both for inspecting and for changing it; direct access to its coordinates would also be helpful. The algorithms described in section 1.4 for manipulating Bézier and B-Spline surfaces suggest that a lot of work will be done for each row and column of some matrix; thus, it would be expected to have functions to get and set one row or column of the matrix at once. Finally, the control points of the surfaces are likely to be given in a (text) file of some appropriate format, and thus a function to read a `GPoint3dArray` from such a file would be welcome.

The next Figure presents these elements of the `GPoint3dArray` class.

---

[1] Now we are thinking of the *implementation* of the classes, a little ahead of time as it might seem. However, design and implementation cannot be completely separated, especially if they are carried out by the same person; in that case, they are rather done in parallel and not sequentially.

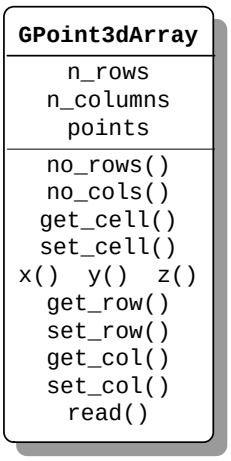

Figure 2-5: Outline of the class implementing matrices of 3D points.

2.4 Implementation

The implementation consolidates all the elements outlined in the requirements section and analysed in more depth in the design section of this Chapter.

In this section are exposed details considering the implementation of the classes for Bézier and B-Spline curves and surfaces. They are presented in a series of topics regarding correlated parts of the software built. This approach was preferred for this part of the document because the structure of the classes has been analysed adequately by now; thus it need not be followed again, as in the last two sections, and instead a unified presentation would be more concise. The reader should consult Appendices B and C which contain the full header files and source code written.

2.4.1 Constructors

Just a quick browsing of the header files in Appendix B reveals the existence of many overloaded constructors for each class. This was necessary in order to keep up with GLOOP tradition of multiplicity of constructors, to maximise interoperability with its other classes and also because this can prove very convenient to the programmer/user of the new classes.

First of all, all classes have a default constructor which yields an empty, null object of that type. Then, for example a `GBezier2d` can be constructed with the control points given as *x* and *y* coordinates, as `Gpoint2d`s or as a `GPolyPoint2d`; in the last case the information on the number of points is provided by the `GPolyPoint2d` class itself and need not be given. The situation is similar for the `GBSpline2d` class; for this class the additional element of the knot sequence is 'optional', the default setting being a uniform knot sequence. In the 3D classes there are fewer constructors, because the `GPoint3dArray` was considered easier and safer to use than any `Gpoint3d**` or

`float**` construct that could possibly be utilised instead; other GLOOP classes such as `GPolyPoint3d` represent inherently linear structures, not suitable for this purpose.

As far as the drawable primitive classes are concerned, each constructor of the respective geometric primitive exists there, in two forms: the first with default drawing attributes (black, solid, thin lines etc.) and the second with attributes specified by the user. Furthermore, drawable objects may be constructed directly from their geometric counterparts, with default or specified attributes.

2.4.2 Indices in the arrays

The entities studied and implemented for this Project make extensive use of one- and two-dimensional arrays, either of points in 2D and 3D (for the control points) or of floating point numbers (for the knot sequences). Unfortunately, C++ is very restrictive in that all (one-dimensional) arrays should start at index 0 and end at index $n-1$ where n is the number of elements in the array. This has proved a real headache when the algorithms described in Chapter 1 were implemented because the arrays treated there very often start with index 1 (at best!). The substitutions performed were frequently very complex and it would not be interesting to present them in detail; the reader should be convinced that they are correct by the fact that they work as expected, for what is yet examined. A brief guide to the most essential conversions follows in the next paragraphs.

An n^{th} degree Bézier curve has exactly $n+1$ control points, labelled in the theory $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$. Constructors of Bézier curves do not have as argument the degree of the curve but the length of the arrays provided, i.e. the number of control points. Thus a declaration such as

```
GBezier2d gbez(no_pts, pts);
```

would construct a Bézier curve of degree `no_pts – 1` with control points `pts[0]`, `pts[1]`,..., `pts[no_pts-1]`. This approach can be confusing at first, because it is the degree and not the number of control points that characterises a Bézier curve; it was favoured, though, for two reasons:

- To follow GLOOP practice. For example a `GPolyPoint2d` consisting of `no_pts` points provided in the array `pts` is declared as

  ```
  GPolyPoint2d gpp(no_pts, pts);
  ```

- Because this form should be easier to use in practice. Before the Bézier curve is declared, the array of control points has to be created somewhere, and for this the number of points is needed; this same number can then be used to declare the curve. For example:

  ```
  int no_pts = 4;   // For a cubic Bézier curve.
  Gpoint2d* pts = new GPoint2d[no_pts];
  …
  GBezier2d gbez(no_pts, pts);
  ```

The situation for Bézier surfaces is very similar, for each parameter t (corresponding to the number of rows) and u (corresponding to the number of columns of the array of control points) separately.

For B-Spline curves, things are even more complicated. A k^{th} degree B-Spline curve with $n+1$ control points $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_n$ had $n+k$ knots labelled $t_1, t_2, \ldots, t_{n+k}$. Here the constructors accept the degree and number of control points of the curve, thus a declaration such as

```
GBSpline2d gbspl(deg, no_pts, pts, knots);
```

will result to a B-Spline curve with control points `pts[0]`, `pts[1]`,..., `pts[no_pts-1]`, consequently requiring `no_pts-1+deg` knots which will be `knots[0]`, `knots[1]`,..., `knots[no_pts-2+deg]`. It should be noticed that the number of knots need not be provided in the constructor as it can be deduced by the degree and number of control points. Again, B-Spline surfaces should look easy once curves are mastered!

2.4.3 Run-time checks

The arrays passed as parameters in the constructors or in the functions returning the control points are never checked for being of the correct size; thus it is the responsibility of the user to ensure this. In fact, C++ does not provide any way of performing such checks, since a single pointer to the first element of the array is passed.

On the other hand, checks *are* made for conformance to various restrictions outlined in the theory:

- That the control points of a B-Spline curve suffice for the required degree.
- That the interval into which a knot is to be inserted is within the appropriate limits (these are of course updated to conform with the indices actually used).
- That the inserted knot(s) retain the knot sequence non-decreasing.
- That any knot to be deleted is within the appropriate limits.
- That the drawing algorithm chosen for B-Splines is applicable (drawing as piecewise Bézier curves is only implemented for quadratic and cubic B-Splines).

In case the user provided wrong arguments, a warning message is issued and, if possible, corrective action is undertaken (for example, drawing using the alternative method).

2.4.4 Calculation of triangles

In many instances in Chapter 1 quantities are calculated using triangular arrangements of numbers or points. This is the case predominantly with de Casteljau triangles, used throughout for 'interpolation steps' (cf. the subdivision of a Bézier curve, the de Boor points – although it was not clearly shown there, and the multiple knot insertion algorithm); it is also the case for calculation of differences [cf. (1.24)].

These arrangements have two characteristics in common:

- The elements of each column are calculated using two adjacent elements of the column at the left, by taking either a weighted sum of them or their difference.
- Not all these points are actually needed; the useful ones are on one side of the triangle.

The first point suggests that the elements in the triangle should be calculated columnwise, from left to right. The second one allows to use only a one-dimensional array large enough to hold the initial elements, and at each step to replace the elements no longer needed by the new ones; after a number of steps (at most) equal to the number of initial elements, the array will hold the desired numbers/points.

For example, to calculate the differences of a sequence of numbers using the scheme (1.24), we can use a single vector of length n which we initialise with the x_i. At each step $r = 1,2,\ldots,n$ we produce $\Delta^r x_0$ by running the vector from bottom to the r^{th} element and replacing x_i by $x_i - x_{i-1}$; after n steps, the vector contains all differences $\Delta^r x_0$, $r = 0,1,2,\ldots,n$. Similar technique applies to the other cases, and variations may only arise from the side of the triangle that we want to keep; see, for example, the implementation of multiple knot insertion in `GBSpline2d` in Appendix C.

2.4.5 Additions since the Design

However complete, the Design of the classes could not predict all their characteristics; some inevitably came out during their Implementation.

An addition useful for the programmer but with no impact to the user of the classes is the incorporation of some auxiliary functions in the class declarations, either as private member functions or as `friend` functions of the classes. This allowed access to the internal structure of the graphical entities and often saved a lot of complexity that would be required, for instance, for copying the control points of the curve to use them. Such are mainly the functions implementing the various drawing methods – the one finally used simply makes the choice between them.

A useful `public` addition was made to the 3D drawable classes only, concerning conversion of the primitives to the corresponding `BRep`. This class is essentially used for shading the surfaces; thus it should be useful to have the `BRep` representing the surface available, so that additional functionality provided by this class, such as Gouraud or Phong shading can be applied directly to it. This facility is not provided as a standard conversion operator, for two reasons: firstly this construct did not seem to work properly when tested, and secondly not all the drawing methods may be used for this conversion. More on this subject follow in the section on Limitations.

2.4.6 Limitations

The classes built to implement curves and surfaces apply all the theory developed in Chapter 1 and satisfy the greatest part of the Requirements set in Section 2.2 of this

document. The few limitations that exist do not restrict in any way the freedom of the user because alternative solutions exist. These limitations are presented next.

- *B-Spline curves and surfaces of degree higher than three* (for surfaces this applies to parameters t and u separately) *cannot be drawn as piecewise Bézier curves/surfaces*.

  In fact the necessary formulae were not given in the theory (page 21), because they are very complex for higher degrees; so, it is natural that this case is not implemented. That said, in practice mainly quadratic and cubic B-Spline curves and surfaces are used and thus this limitation has little practical effect.

  If the user attempts to draw a B-Spline curve/surface of degree higher than three with this method, a warning is issued and the de Boor algorithm is used instead, which can treat arbitrary B-Splines.

- *Bézier surfaces can only be shaded using the differences method*. (But they can be drawn as wireframes using either the de Casteljau or the differences method.)

  This is solely a limitation of the current implementation. As mentioned before, shading is implemented using the `BRep` class provided in GLOOP; this class uses the boundary representation of a complete 3D object to perform the back surface culling of the polygons comprising it and to sort them from back to front with respect to the camera. The differences method produces immediately a mesh representing the surface and it is easy to convert this to its boundary representation. On the contrary, the de Casteljau algorithm is recursive and thus it is not trivial to assemble all the points generated in a list, throw away duplicate points (necessary for the boundary representation to be valid) and keep references to which points form which polygon. This would possibly involve some suitable data structures for the list of points and for the references to the vertices of the polygons. It should be said that efforts have been made to implement all this with existing GLOOP constructs but they have not been successful.

  If the user attempts to shade a Bézier curve with the de Casteljau method, a warning message is issued and the differences method is used instead.

- All the same, *B-Spline surfaces cannot be shaded as piecewise Bézier surfaces*. (But they can be drawn as wireframes either as piecewise Bézier surfaces or with the de Boor algorithm.)

  The reason is the same as for shading Bézier surfaces with the de Casteljau algorithm: the boundary representation of the surface is difficult to form using this approach. Of the two algorithms that could be used for shading the Bézier sub-surfaces, the de Casteljau one is out of the question after what is mentioned in the last point. The differences method could be used relatively easily: one would have to keep the `BRep`s of the sub-surfaces in an array, process each separately and sort them (outside of the `BRep` class) from back to front. The reason that all this is not implemented is that it would not actually worth it. The main quality of the de Casteljau algorithm is that it is adaptive to the curvature of the surface, and this would justify any effort to implement it for shading Bézier surfaces and then use it for B-Spline ones; this quality is not possessed by the differences method. Given

that the de Boor method produces analogous results and is applicable to arbitrary B-Spline surfaces, whereas finding the Bézier control points is restricted to quadratic or cubic ones, it is clear that the differences method is redundant in this case.

If the user attempts to shade a B-Spline curve as a piecewise Bézier surface, a warning message is issued and the de Boor method is used instead.

2.5 Testing

Testing is a very important part of software development because it reveals defaults in the implementation that could not otherwise be spotted. The programmer usually has in mind the most general case and possibly some exceptions when (s)he is writing code; only proper testing pushes the software to its limits and reveals its imperfections.

A software system should be tested in stages, starting from individual functions, passing from complete modules and finally reaching the complete system. The testing stages and procedures followed for this Project are outlined next.

2.5.1 Unit testing

This stage of testing had to ensure that all functions of the classes constructed work as expected. For this, small programs treating Bézier and B-Spline curves and surfaces were written, which used as many of the defined functions as possible. The general pattern followed was:

- First, testing of the functions for 'reasonable', regular data. This ensured that they work fine with various sets of control points, drawing parameters, etc.
- Then, testing of the functions with abnormal or simply not so common data; for example zero control points, zero degree, knots repeated many times, insertion of knot in the first interval, etc. This part was very important because it revealed many exceptional cases (mentioned, with hindsight, in the theoretical part as restrictions) and led to the incorporation of many run-time checks in the programs.

Testing was particularly extensive for the classes implementing curves in 2D. In this case it was easy to figure out test sets of points, re-compilation and drawing times were reasonably short and also the class hierarchy was considerably simpler than the one for the 3D classes.

On the other hand, for the 3D classes it was inherently more difficult to find control points that would produce a reasonable surface; moreover several other parameters, such as the camera position, affect the display of the surface and they were often difficult to control. Most important of all, when more advanced classes of GLOOP were involved, for example the `BRep` class in the shading of surfaces, unexpected errors came upon which

were due, as some investigation revealed, to imperfections of these classes and not of our code (fortunately most of these have been rectified, thanks to the co-operation of Simon Arridge). This is not to say, however, that the 3D classes are not properly tested. Additionally, the fact that their code comes directly from the one of the 2D classes – suitably adapted – can convince us that it should be almost equally robust as this.

2.5.2 System testing

This Project did not involve the production of an application program; it rather aimed to develop a programming tool, a set of classes implementing curves and surfaces. Because of this, the only meaning we can give to 'system testing' in this case is testing of the classes at work; it is, in great part, equivalent to module testing, and this is why all intermediate stages after unit testing are omitted in this document.

Again, this stage of testing was performed by means of sample programs, this time more complicated than for the previous stage.

In 2D, not much more could or in fact needed to be done. Test programs assured good interoperability with other GLOOP features, when applicable, tried more complex shapes etc. All test programs have finished successfully.

In 3D, a more interesting program was constructed, which draws a teapot as 32 distinct cubic Bézier surfaces. It has a full set of options to specify the drawing method, drawing parameters and lighting model. It obviously handles large amounts of data and demonstrates the Bézier classes in use, verifying that they work adequately. Examples of its use can be found in the next chapter.

This program also revealed a bug that we have not been able to correct, not even to attribute to some part of existing or newly written code: when the teapot is shaded with the differences method and some specific values for the evaluation step for t and u, the program hangs. Similar behaviour has not been observed for any other surface tested. Possible explanations include: some mistake in copy constructors or assignment operators of some of the many classes involved, thus producing faults when objects are initialised or assigned; or, some bug in the part of the package that encodes the drawing for the external drawing program thus producing an incompatible code for this specific case.

Except for this, the classes constructed work as expected in all the cases tested.

3. RESULTS

Most applications in Computer Graphics are computationally very intensive, and thus the choice of the most suitable algorithm for each specific case can have great impact on the performance of the application. There is always a trade-off between the visual detail expected and the required drawing speed. This is even more the case with the graphical entities studied for this Project, as they treat freeform shapes which are inherently difficult to represent accurately using simpler constructs such as straight lines. Moreover, multiple algorithms were implemented to draw them, each with its own qualities.

This chapter presents the outcome of this Project in practice. Mainly through a series of Figures, it compares the various algorithms implemented and illustrates the functionality provided. This comparison concerns mostly the visual characteristics of the algorithms though reference is also made to their performance.

3.1 Visual characteristics

3.1.1 Curves

Drawing of curves is reasonably fast in any case and so the user can choose the drawing parameters so as to produce smooth curves, without severe time overhead. Thus we should not really compare the various methods implemented for visual performance – they can all be equally accurate.

Bézier curves

Figure 3-1 shows some example control points and the Bézier curve they generate. This is a 9^{th} degree Bézier curve and would hardly be used in practice: it is rather unintuitive as it does not follow its control polygon reasonably well.

The only interesting topic on Bézier curves is subdivision. In Figure 3-2 is demonstrated the first subdivision step for the de Casteljau algorithm on this curve. It can be seen that the left and right control polygons converge quickly to the curve.

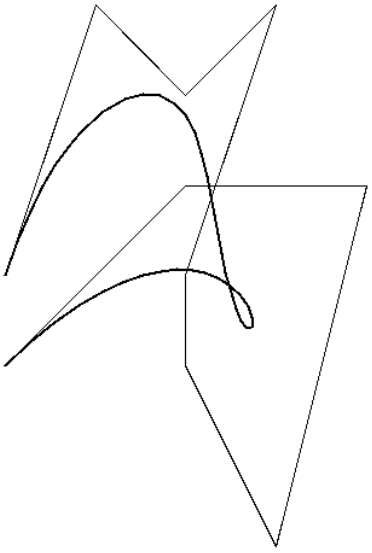

Figure 3-1: An example Bézier curve.

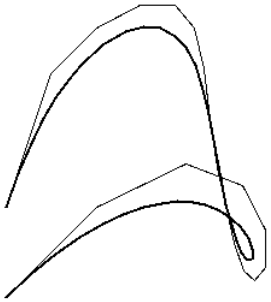

Figure 3-2: Subdivision of the Bézier curve.

B-Spline curves

The same control points used for the Bézier curve above give rise to the cubic B-Spline curve (with uniform knot sequence) shown in Figure 3-3. This one is far more predictable from its control points.

The effect of different parametrisations is presented in Figure 3-4. This B-Spline has several multiple knots apart from the end ones, thus it presents the cusps shown, and its last three knots are not equal, thus it ends before its last control point.

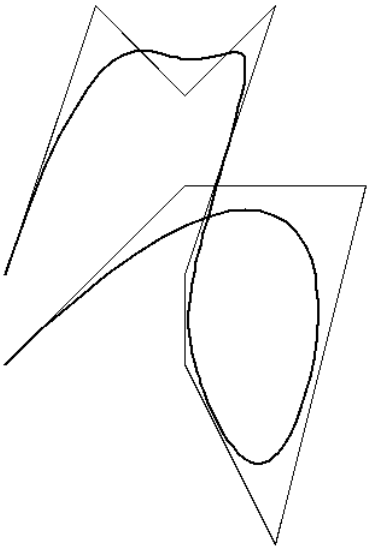

Figure 3-3: Cubic B-Spline curve.

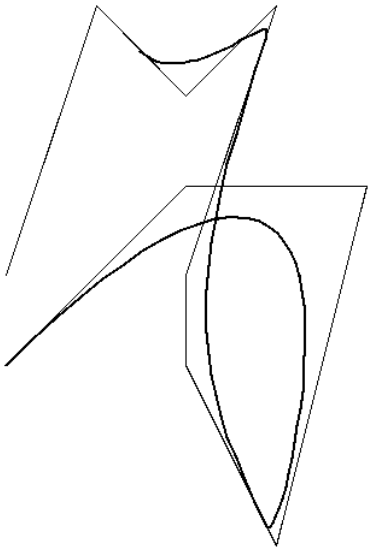

Figure 3-4: Non-uniform knot sequence.

In Figure 3-5 is shown the result of knot insertion: the control polygon is seen to converge to the B-Spline curve quickly.

Finally, Figure 3-6 shows a cubic B-Spline curve interpolating the same set of control points used so far. The result is somehow awkward because uniform parametrisation is used; other parametrisations would yield much better results.

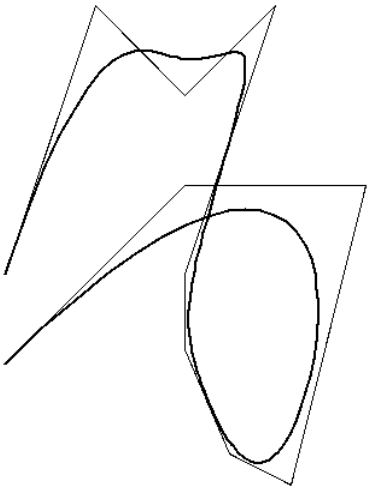

Figure 3-5: Knot insertion.

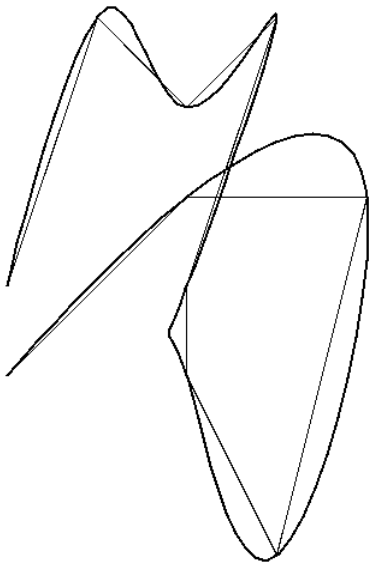

Figure 3-6: Cubic B-Spline interpolation.

We close the presentation of B-Spline curves with an example that clearly demonstrates their flexibility: The letter **r** in Figure 3-7 is *not* made from an interpolating B-Spline; it is a simple cubic B-Spline with 46 control points and triple knots everywhere.[1] In this case, the effect of multiple knots has been used constructively to produce an interesting shape.

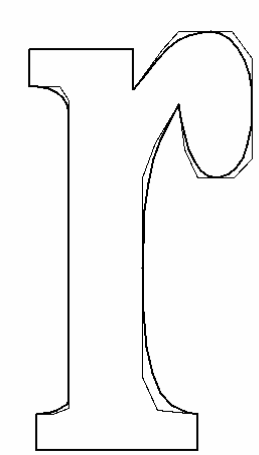

Figure 3-7: An interesting B-Spline curve.

[1] The data for this drawing comes from [1].

3.1.2 Surfaces

For surfaces, things are not so straightforward as they were for curves. The computational complexity of all the algorithms is higher and usually a compromise between drawing speed and accuracy is made. So in this case, the choice of the suitable algorithm may be important for the outcome.

Bézier surfaces

Figure 3-8 and Figure 3-9 contrast the two different methods for drawing Bézier curves. The outcome is similar in both cases (the de Casteljau algorithm used a tolerance level of 0.1 and the differences method used evaluation step 0.1 for t and u), but it is clear that the differences method does some unnecessary work in areas where the surface is rather flat; the de Casteljau method 'concentrates' where it really needs to.

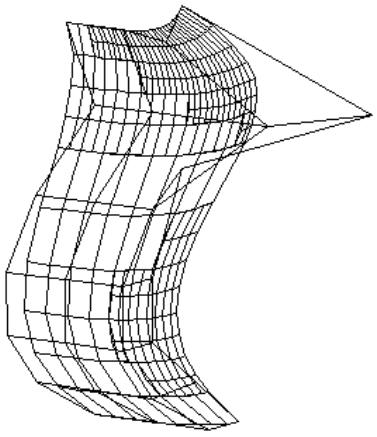

Figure 3-8: Bézier surface using the de Casteljau algorithm.

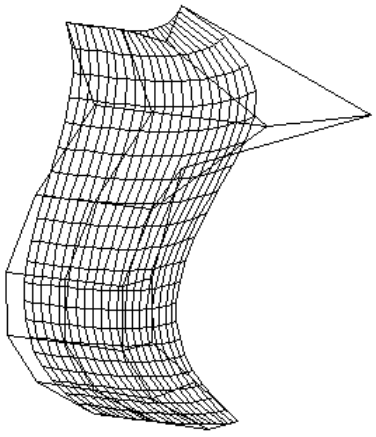

Figure 3-9: Bézier surface using differences.

The adaptability of the de Casteljau algorithm can sometimes cause problem. This is demonstrated in Figure 3-10, where some polygons can be seen not to join correctly because different recursion depths were applied to them. This is extremely obvious if coarse flatness tests are applied such as in our case (tolerance level 0.5).

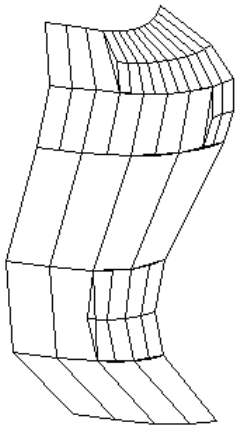

Figure 3-10: Inefficiency of the de Casteljau algorithm.

In Figure 3-11 is shown the outcome of the Teapot program contained in Appendix D. On the left, the teapot is drawn as a wireframe using the de Casteljau algorithm with tolerance level 0.1. On the right, it is shaded using the differences method and evaluation steps 0.05 for both t and u; this value is rather small but was used to produce a smooth outcome. For more details on these pictures, refer to Appendix D.

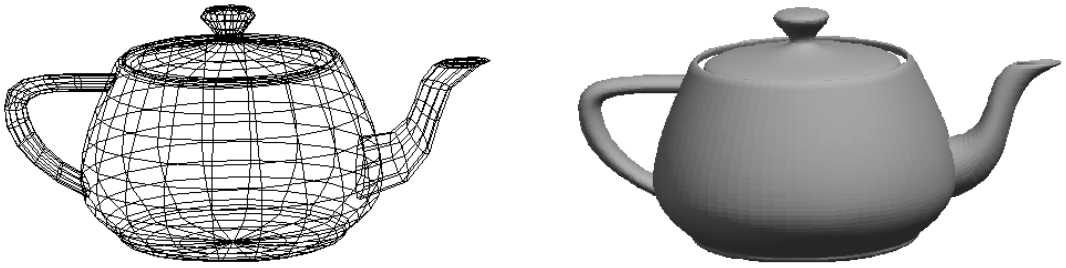

Figure 3-11: The famous teapot.

B-Spline surfaces

For B-Spline surfaces we do not demonstrate again the topics mentioned in the sub-section on curves; the drawings would be too cluttered to be intelligible. We simply show, in Figure 3-12, the bi-cubic B-Spline surface resulting from the same control points as our previous Bézier surface, and in Figure 3-13, interpolation of these points with a bi-cubic B-Spline surface. As for curves, uniform parametrisation is responsible for the rather awkward result in some places. The wireframe in Figure 3-12 is drawn using the de Boor method and shows the inefficiency of a fixed evaluation step for the whole surface. On the contrary, the surface in Figure 3-13 is even steeper, but thanks to the de Casteljau algorithm used for the Bézier sub-surfaces, the result is acceptably smooth everywhere.

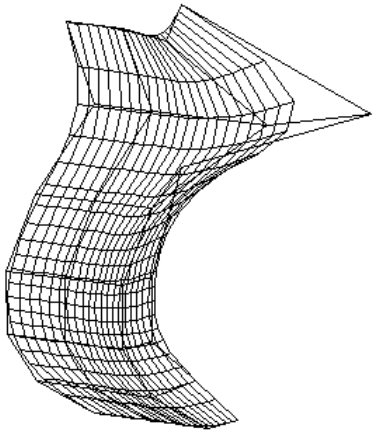

Figure 3-12: Bi-cubic B-Spline surface.

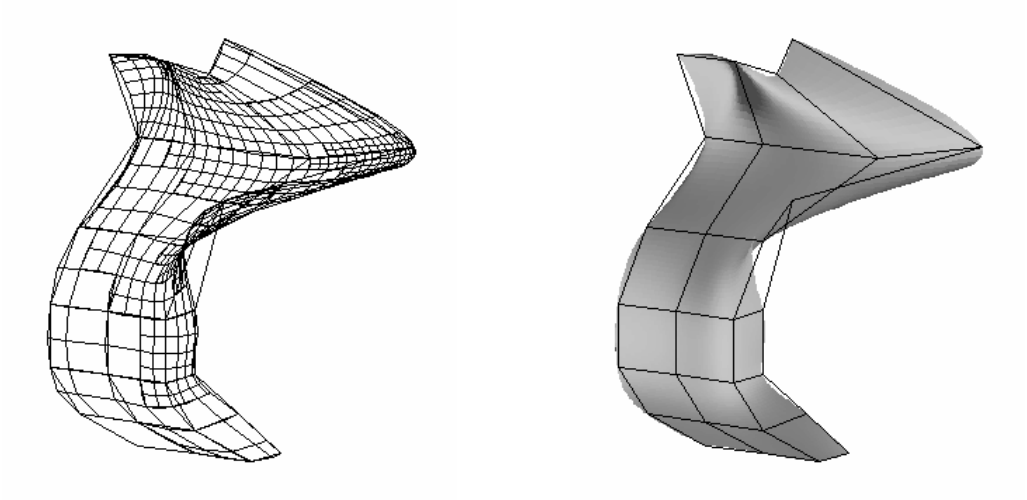

Figure 3-13: Interpolating B-Spline surface.

3.2 Performance

Having presented the visual qualities of the algorithms implemented, it would also be useful to compare them in terms of drawing speed, obviously for relatively equivalent visual outcome.

Unfortunately, this was not easy in this case. First of all, the actual drawing is carried out by an external program which receives the data through Unix pipes; this mechanism introduces inevitable delays, whose proportion in the time to display the drawings cannot be measured. Moreover, the classes written carry out only the computation directly related to calculating points on the curves/surfaces, whereas those related to the actual drawing of lines or filling of polygons are performed externally and again consume an unknown proportion of the drawing time. Formal testing of the algorithms *could* have been done, using the Unix utility `prof` / `gprof`, which analyses the time spent separately in each function called during the execution of a program. However, it has not been possible to make it work, under either of the platforms used for the development of the software (see below).

Therefore, the only comparison of the algorithms could be done by timing the drawing of a specific curve or surface with each of the ones concerned (possibly consecutive times in a loop for the results to have some statistical value), using parameters that yield visually close outcomes. This approach may not demonstrate the value of each algorithm on its own, but is nevertheless a useful measure of their performance in the environment into which they are set to work.

Tests concerning curves have been carried out on a Pentium® PC at 133MHz running Linux 1.2.13, whereas those concerning surfaces have been carried out on a Sun SparcStation running SunOS (the 3D GLOOP libraries have not been ported to this version of Linux successfully). It should be noted that drawing speed on the Pentium

machine was generally higher than on the SparcStation for the 2D examples, which were executed on both for a simple comparison of architectures.

3.2.1 Curves

For Bézier curves, all tests carried out showed that the differences method outperforms the de Casteljau algorithm in their current implementations. This can be traced to the fact that the de Casteljau algorithm executes the complex flatness tests at each recursive step, whereas the differences method only needs additions after some initial costly calculations.

For B-Spline curves, the de Boor method for calculating points on the curve was found to be the slowest of all; this is not surprising as for each point it evaluates a different de Casteljau triangle, and it requires many points to produce a smooth curve. When drawing the B-Spline curve as Bézier curve segments, the differences method was again faster, although this time the de Casteljau algorithm was closer, possibly because other calculations were involved as well (computation of the Bézier control points).

3.2.2 Surfaces

In the case of surfaces, performance tests mainly showed that the bottleneck in the drawing procedure is the display of the lines or polygons themselves, and not the calculations involved in the algorithms: because so many lines or polygons have to be drawn, the proportion of the computation in the whole time required is rather small. It would be interesting to note, at this point, that testing has shown that drawing of simple wireframes is a lot *slower* than drawing the same polygons filled with some colour, in the current implementation of the external drawing program. For example, the teapot at the right of Figure 3-11 required almost the double time to be drawn as a wireframe (not shown here) than shaded, as shown!

Other testing carried out for Bézier and B-Spline surfaces confirmed that the results for curves apply to surfaces, too. This time, however, depending on the shape of the surface, the de Casteljau algorithm can produce a fairly acceptable result with a higher tolerance level and thus be faster than the differences method. For this its use may be preferable, especially for previewing purposes.

CONCLUSIONS – FURTHER WORK

This Project aimed at studying Bézier and B-Spline curves and surfaces and at implementing them in C++ using GLOOP to provide the background of the work.

In the time scale available, coverage of the relevant theory was broad enough to include the most important topics regarding these two classes of curves and surfaces; it was also deep enough to allow for proper understanding of the concepts involved, which led to their relatively untroubled implementation. But it was often the inverse case, too: difficulties that arose during the implementation of the algorithms neatly presented in theory, or during the testing of the classes built, revealed subtle points that only a practical approach could expose.

It need not be emphasised that this is even more true as far as the implementation itself is concerned. This Project was a very good way of practising C++ in the context of a medium-scale software project, as the classes written did not only have to work correctly but also to co-operate with existing software. This was at the same time a great advantage, because the study of existing code has proved beneficial in the development process.

As far as this Report is concerned, effort has been made so that the presentation of the work is self-contained. The length and scope of this document only allowed to discuss those concepts directly related to the work of this Project. Some related interesting topics not falling within these strict limits were left out, but references are provided for further study.

The software constructed aimed at an efficient implementation of the algorithms, well integrated into GLOOP, and it is believed to have succeeded in this. The design of the classes provides the programmer with all the required functionality and ensures good interoperability with the package. The few existing limitations are overcome by other features, thus resulting to a complete and useable package.

Anyone who would like to study the code and possibly modify or extend it should inevitably understand the relevant theory and have acquaintance with GLOOP. This is more to perceive its internal structure than because the use of GLOOP features make it incomprehensible. Apart from the implementation of shading which depends on other classes strongly, one would need just a structure to model points in 2D and 3D and some function to draw straight lines to make use of most parts of the code without significant modifications.

All that said, the software is not perfect – it could not possibly be. Some additions that could be incorporated in the classes built are the following:

- Generalisation of the algorithm for drawing B-Spline curves and surfaces as piecewise Bézier ones. Although preliminary research has shown that this not a

trivial task, the benefit mainly of using the de Casteljau algorithm for B-Spline surfaces would be considerable.

- Implementation of shading of surfaces with all the methods provided for their drawing as wireframes. The drive behind this is to provide wider choice to the user, which can be valuable depending on the specific application.
- Provision of conversion operators to other GLOOP features; for example a B-Spline curve could be converted to a `GPolyPoint2d` holding its de Boor points. A premature form of such a conversion is provided for surfaces to the corresponding `BRep`, using only the methods performing shading of the surfaces. Such modifications are straightforward in some cases but have been unsuccessful in others and would probably require some restructuring of the classes. None has been implemented, in order to preserve consistency across the classes written.
- Construction of the graphical entities, especially the surfaces, directly from files of standard format. The classes for surfaces so far accept only the rectangular array of control points as such and it would be helpful to have it made out of some text file containing, for example, the boundary representation of the control points. (See the Teapot program in Appendix D.)

Apart from the above improvements that could be applied to the existing software, the field of CAGD is rich in concepts that could be studied and implemented to provide yet more flexible ways of defining curves and surfaces. These include:

- *Different parametrisations for B-Spline curves and surfaces.*

  In Chapter 1 this subject is merely referred to as being 'very important', and the classes constructed treat the most simple case of uniform parametrisation. The references provided on this subject (page 23) discuss other ways of parametrisation that take into account the geometry of the control points of the curve and are thus more likely to produce 'better' results.

  Appropriate implementation of these would probably involve an abstract base class to model the characteristics of a knot sequence and derived classes to implement specific parametrisations. These new classes could fit into the existing structure of classes for B-Splines simply by providing the knot sequence as an array of floating point numbers.

- *Rational curves and surfaces.*

  Rational Bézier curves are again specified by a set of control points with the help of the Bernstein polynomials, and additionally this time of a set of weights w_i, $i = 0,1,2,\ldots,n$. The formula of an n^{th} degree rational Bézier curve is

$$\mathbf{x}^n(t) = \sum_{i=0}^{n} w_i B_i^n(t)\,\mathbf{p}_i \Bigg/ \sum_{i=0}^{n} w_i B_i^n(t)\,,$$

  and its similarity to simple Bézier curves is imminent. The weights involved add further freedom in the shape of the curve. From rational Bézier curves can be formed (non-uniform) rational B-Splines, the so called NURBS, which are a very

flexible tool for describing curves and often used in practice. Additionally, rational surfaces can be defined based on rational curves, just as it was done for simple surfaces. Our familiar Bézier and B-Spline curves and surfaces are in fact special cases of their rational counterparts, with all the weights equal.

Implementation of rational curves (and surfaces) could not be done without the creation of new classes. The weights now present in the formulae require rewriting of the code to incorporate them. Fortunately, much of the existing code could be adapted with minor changes to the rational case since the concepts are basically the same. Once these are implemented, simple Bézier and B-Spline curves and surfaces could be reduced to derived classes with the weights set equal. This approach could not have been followed for this Project as the time available only permitted study of the initial concepts.

- *Triangular patches for Bézier surfaces.*

  As noted on page 28, Bézier surfaces can be represented with triangular patches instead of the rectangular ones produced by the tensor product approach. This form can represent more complex shapes, not inherently 'rectangular', more easily than the traditional one.

  Implementation of triangular Bézier patches would require the writing of new classes, because the mathematics are quite different in this case. Again, however, the concepts do not differ a lot and the existing code could be used as a starting point.

REFERENCES

[1] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design*, 3rd edition, Academic Press, 1993.

[2] M. Slater, *Computer Graphics and Virtual Environments* (Chapter 13), draft, 1996.

[3] J. Hoschek, D. Lasser, *Fundamentals of Computer Aided Geometric Design*, A K Peters, 1993.

[4] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[5] G. Salmon, *Analytic Geometry of three dimensions*, Vol. 1, 7th edition, Longmans, Green and Co. Ltd., 1928.

[6] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics, Principles and Practice*, 2nd edition in C, Addison Wesley, 1996.

[7] I. Sommerville, *Software Engineering*, 5th edition, Addison Wesley, 1995.

[8] B. Stroustrup, *The C++ Programming Language*, 2nd edition, Addison Wesley, 1991.

[9] S. Oualline, *Practical C++ Programming*, 1st edition, O'Reilly & Associates, 1995.

[10] S. Arridge, *GLOOP: 2d Graphics User Manual* and *GLOOP in 3d: User Manual*, Department of Computer Science, University College London.

APPENDIX A: MANUAL PAGES

In this Appendix are given printouts of the manual pages describing the classes constructed for this Project. This form of reference information was chosen because it does not lack completeness, compared with a written reference guide, while in the same time it is readily accessible electronically when the classes are being used.

NAME

GBezier2d

TYPE

Geometric primitive class.

DESCRIPTION

GBezier2d is the geometric primitive for the class of Bezier curves of any degree in two dimensions. Its drawable counterpart is Bezier2d. It is made of an array of GPoint2ds which are the control points of the curve.

INHERITANCE INFORMATION

Derived From

GPrimitive2d

Base For

Bezier2d

CONSTRUCTORS

GBezier2d();
creates an empty Bezier curve.

GBezier2d(int no_pts, const float* x, const float* y);
creates a Bezier curve with no_pts control points specified by their coordinates.

GBezier2d(int no_pts, const GPoint2d* pts);
is similar to the second except that the points are provided as GPoint2ds.

GBezier2d(const GPolyPoint2d& polypoint);
creates a Bezier curve with control points specified as a GPolyPoint2d. In this case the number of control points is provided by the polypoint itself.

GBezier2d(const GBezier2d& bezier);
The copy constructor. It is necessary because the class manipulates dynamically allocated storage.

ACCESS FUNCTIONS

int no_pts();
returns the number of control points of the Bezier curve.

int deg();
returns the degree of the Bezier curve (always no_pts() - 1).

void control_pts(GPoint2d* points);
returns the control points of the Bezier curve in points (which must be of the appropriate size).

MEMBER FUNCTIONS

GRectangle2d minmax_box();
returns the minmax box of the Bezier curve as a GRectangle2d. It can be useful for checking intersection with the curve.

GPolyPoint2d convex_hull();
returns the convex hull of the Bezier curve as a GPolyPoint2d. It can be useful for checking intersection with the curve.

void subdivide(float t, GBezier2d& bez_left, GBezier2d& bez_right);
subdivides the curve into two Bezier curves, left and right, at parameter value t. Mainly used in the implementation of the de Casteljau algorithm for drawing Bezier curves, but may have other uses.

int is_flat(float tolerance);
checks if the Bezier curve is flat within the given tolerance. Also used in the implementation of the de Casteljau algorithm.

void transform(const Trans2d& trans);
transforms the Bezier curve according to trans, as usual for all primitives in Gloop.

CONVERSION OPERATORS

A Bezier curve cannot be converted to any other primitive defined in Gloop.

OVERLOADED OPERATORS

GBezier2d& operator=(const GBezier2d& bezier)
The assignement operator for GBezier2ds. As the copy constructor, it is necessary because the class manipulates dynamically allocated storage.

SEE ALSO

GPoint2d, **GPolyPoint2d**, **Trans2d**, **Bezier2d**.

AUTHOR

Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS

No known bugs.

NAME

Bezier2d

TYPE

Drawable primitive class.

DESCRIPTION

Bezier2d is the drawable primitive for the class of Bezier curves of any degree in two dimensions. It is the drawable counterpart of GBezier2d. A Bezier2d consists of two sub-objects of types GBezier2d and Attributes.

INHERITANCE INFORMATION

Derived From

Primitive2d (and hence Attributes)
GBezier2d

CONSTRUCTORS

For each way of specifying a Bezier2d there are two constructor functions. The first form constructs a primitive with default attribute values; the second with specified attribute values. Only the first form is described below. The parameter list will be the same for the second form except that an additional parameter of type 'const Attributes &' is required at the end. For example, an empty Bezier curve with default attributes can be constructed with
Bezier2d bez();
and a Bezier curve with specified attributes a can be constructed with
Bezier2d bez(a);

Bezier2d();

creates an empty Bezier curve.

Bezier2d(int no_pts, const float* x, const float* y);

creates a Bezier curve with no_pts control points specified by their coordinates.

Bezier2d(int no_pts, const GPoint2d* pts);

is similar to the second except that the points are provided as GPoint2ds.

Bezier2d(const GPolyPoint2d& polypoint);

creates a Bezier curve with control points specified as a GPolyPoint2d. In this case the number of control points is provided by the polypoint itself.

Bezier2d(const Bezier2d& bezier);

The copy constructor. In is necessary because the default (memberwise) copy constructor did not work well with derived classes.

MEMBER FUNCTIONS

Bezier2d inherits the access and control functions of its base classes. Special member functions deal with the drawing of the Bezier curves.

void set_drawing_method(Bezier_algorithm alg);

specifies the algorithm to be used for drawing the Bezier curve. Two methods for drawing Bezier curves are implemented, based on the de Casteljau algorithm and on the use of differences; the choice that will give the best result may depend on the specific application. The variable alg is of an enumeration type defined for this purpose:
enum Bezier_algorithm {DE_CASTELJAU, DIFFERENCES};
The default drawing algorithm is the differences method.

void set_tolerance(float tol);

sets the tolerance for the flatness tests in the de Casteljau algorithm. Smaller values produce smoother curves. This setting is used only when the drawing method is DE_CASTELJAU. The default tolerance is 0.01

void set_eval_step(float h);

sets the step at which the function representing the curve is evaluated. Smaller values produce smoother curves. This setting is used only when the drawing method is DIFFERENCES. The default evaluation step is 0.01. NOTE: It is advisable to use values that have integer reciprocal, for example 0.1 or 0.3333 but not 0.3; otherwise the curve will not end at its last control point. (The reciprocal of the evaluation step is the number of steps needed to run the range of t from 0 to 1. This behaviour is inherent in the algorithm.) Moreover, evaluation step greater than 0.5 will have awkward results.

void draw(View2d& view);

draws a Bezier curve with the specified method and drawing parameters (or the default ones if they are not specified).

void draw_transformed(View2d& view, const Trans2d& trans);

draws the Bezier curve as before but transformed according to trans (without affenting the original curve).

void transform(const Trans2d& trans);

transforms the Bezier curve according to trans, as usual for all primitives in Gloop.

CONVERSION OPERATORS

A Bezier curve cannot be converted to any other primitive defined in Gloop. A Bezier2d may be used in the place of a GBezier2d or of Attributes as allowed by inheritance.

SEE ALSO

Attributes, **View2d**, **Trans2d**, **Bezier2d**.

AUTHOR

Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS

No known bugs.

NAME

GBSpline2d

TYPE

Geometric primitive class.

DESCRIPTION

GBSpline2d is the geometric primitive for the class of B-Spline curves of any degree and containing any number of control points, in two dimensions. Its drawable counterpart is BSpline2d. (See its manual page for restrictions on drawing arbitrary B-Splines.) It is made of an array of GPoint2ds which are the control points of the curve, and of an array of floats which are the knots of the curve.

INHERITANCE INFORMATION

Derived From

GPrimitive2d

Base For

BSpline2d

CONSTRUCTORS

For each way of specifying a GBSPline2d (except for an empty GBSpline) there are two constructor functions. The first form constructs a primitive with default knot sequence, i.e. uniformly spaced and with the degree first and last knots equal; the second with specified knot sequence. Only the first form is described below. The parameter list will be the same for the second form except that an additional parameter of type 'const float *' is required at the end. For example, a cubic B-Spline curve with five control points stored in the array pts and default knot sequence can be constructed with
BSPline2d bez(3, 5, pts);
and a B-Spline curve with the same characteristics and specified knot sequence knots can be constructed with
BSPline2d bez(3, 5, pts, knots);
For the second form, the number of required knots is described in the appropriate access function. Also the knots must be in non-decreasing order or the construction is aborted.

GBSpline2d();

creates an empty B-Spline curve.

GBSpline2d(int deg, int no_pts, const float* x, const float* y);

creates a B-Spline curve of degree deg with no_pts control points specified by their coordinates.

GBSpline2d(int deg, int no_pts, const GPoint2d* pts);

is similar to the second except that the points are provided as GPoint2ds.

GBSpline2d(int deg, const GPolyPoint2d& polypoint);

creates a B-Spline curve of degree deg with control points specified as a GPolyPoint2d. In this case the number of control points is provided by the polypoint itself.

GBSpline2d(const GBSpline2d& bspline);

The copy constructor. It is necessary because the class manipulates dynamically allocated storage.

ACCESS FUNCTIONS

int deg();

returns the degree of the B-Spline curve.

int no_pts();

returns the number of control points of the B-Spline curve.

void control_pts(GPoint2d* points);

returns the control points of the B-Spline curve in points (which must be of the appropriate size).

int no_knts();

returns the number of knots of the B-Spline curve (always no_pts() - 1 + deg()).

void knot_sequence(float* knot_seq);

returns the knots of the curve in knot_seq (which must be of the appropriate size).

MEMBER FUNCTIONS

void insert_knot(int index, const float knot_value);

inserts a new knot with value knot_value in the interval starting with the knot indexed index, while retaining the shape of the curve, and updates the control points and knot sequence of the curve. Insertion is only possible in intervals starting with index between deg() - 1 and no_pts() - 2 inclusive; also the resulting knot sequence must still be non-decreasing. If any of these conditions is not satisfied the insertion is aborted.

void insert_many_knots(int index, int no_knts, const float* knot_values);

inserts all no_knts knots contained in knot_values in the same interval starting with the knot indexed index, while retaining the shape of the curve, and updates the control points and knot sequence of the curve. The same conditions as for single insertion must be satisfied.

void delete_knot(int index);
deletes the knot with given index, which must be between deg() and no_pts() - 1 inclusive. When removing a newly inserted knot the result is exactly the curve before the insertion; in any other case the resulting curve may be unpredictable!

void transform(const Trans2d& trans);
transforms the B-Spline curve according to trans, as usual for all primitives in Gloop.

void interpolate(const int no_pts, const float* x, const float* y, const GPoint2d r1, const GPoint2d r2);
void interpolate(const int no_pts, const GPoint2d* pts, const GPoint2d r1, const GPoint2d r2);
void interpolate(const GPolyPoint& polypoint, const GPoint2d r1, const GPoint2d r2);
construct a cubic B-Spline curve to interpolate the given set of points (in any of the above forms). The B-Spline constructed has default (uniform) knot sequence. There are also forms of the above functions in which the user specifies the knot sequence, such as
void interpolate(const int no_pts, const float* x, const float* y, const float* knots, const GPoint2d r1, const GPoint2d r2);
The number of knots must be no_pts + 4 (the resulting B-Spline has no_pts + 2 control points). The two arbitrary points supplied, r1 and r2, are used as the second and the last but one control point respectively.

CONVERSION OPERATORS
A B-Spline curve cannot be converted to any other primitive defined in Gloop.

OVERLOADED OPERATORS
GBSpline2d& operator=(const GBSpline2d& bezier)
The assignement operator for GBSpline2ds. As the copy constructor, it is necessary because the class manipulates dynamically allocated storage.

SEE ALSO
GPoint2d, **GPolyPoint2d**, **Trans2d**, **BSpline2d**.

AUTHOR
Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS
No known bugs.

NAME

BSPline2d

TYPE

Drawable primitive class.

DESCRIPTION

BSPline2d is the drawable primitive for the class of B-Spline curves in two dimensions. It is the drawable counterpart of GBSPline2d. A BSPline2d consists of two sub-objects of types GBSPline2d and Attributes.

INHERITANCE INFORMATION

Derived From

Primitive2d (and hence Attributes)
GBSPline2d

CONSTRUCTORS

For each way of specifying a BSPline2d (except for empty ones) there are four constructor functions. The first form constructs a primitive with default attribute values and default (uniform) knot sequence; the second with specified attribute values and default knot sequence; the third with default attribute values and specified knot sequence; and the fourth with specified attribute values and specified knot sequence. Only the first form is described below. The parameter list will be the same for the other forms, with the following additions: for the second form of a parameter of type 'const Attributes&' at the end; for the third form of a parameter of type 'const float*' at the end; and for the fourth form of two parameters of types 'const float*' and 'const Attributes&' at the end. For example, a cubic B-Spline curve with five control points stored in the array pts and default knot sequence can be constructed with
BSPline2d bez(3, 5, pts);
and a B-Spline curve with the same characteristics, specified knot sequence knots and drawing attributes a can be constructed with
BSPline2d bez(3, 5, pts, knots, a);

BSPline2d();
creates an empty B-Spline curve.

BSPline2d(int no_pts, const float* x, const float* y);
creates a B-Spline curve with no_pts control points specified by their coordinates.

BSPline2d(int no_pts, const GPoint2d* pts);
is similar to the second except that the points are provided as GPoint2ds.

BSPline2d(const GPolyPoint2d& polypoint);
creates a B-Spline curve with control points specified as a GPolyPoint2d. In this case the number of control points is provided by the polypoint itself.

BSPline2d(const BSPline2d &bezier);
The copy constructor. In is necessary because the default (memberwise) copy constructor did not work well with derived classes.

MEMBER FUNCTIONS

BSPline2d inherits the access and control functions of its base classes. Special member functions deal with the drawing of the B-Spline curves.

void set_drawing_method(BSpline_algorithm alg);
specifies the algorithm to be used for drawing the B-Spline curve. Two methods for drawing B-Spline curves are implemented, as piecewise Bezier curves and using the de Boor formula; the choice that will give the best result may depend on the specific application. The variable alg is of an enumeration type defined for this purpose:
enum BSpline_algorithm {AS_BEZIER, DE_BOOR};
The default drawing algorithm is the de Boor method. Drawing of B-Splines as piecewise Bezier curves is only implemented for quadratic and cubic B-Splines; for other degrees, the de Boor method is always used. NOTE: The de Boor method may attempt division by zero and thus produce an error with some knot sequences containing multiple knots other than the end knots. This should not be common with the knot sequences used in practice.

void set_Bezier_drawing_method(Bezier_algorithm alg);
sets the algorithm used to draw the Bezier curve segments. This setting is used only when AS_BEZIER is chosen as the drawing method of the B-Spline. For the default value, refer to the manual page for Bezier2d.

void set_decas_tolerance(const float tol);
sets the tolerance for the flatness tests in the de Casteljau algorithm. This setting is used only when the drawing method for the B-Spline is AS_BEZIER and the method for the curve segments is DE_CASTELJAU. For the default value, refer to the manual page for Bezier2d.

void set_diff_eval_step(float h);
sets the step at which the functions representing the Bezier curve segments are evaluated. This setting is used only when the drawing method for the B-Spline is AS_BEZIER and the method for the curve

segments is DIFFERENCES. For the default value and an important note on advisable values for the evaluation step refer to the manual page for Bezier2d.

void set_eval_step(float h);
sets the step at which the function representing the B-Spline curve is evaluated using the de Boor method. Smaller values (but not too smooth) produce smoother curves. This setting is only used when the drawing method for the B-Spline is DE_BOOR. The default value is 0.01. NOTE: Values greater than 0.5 will have awkward results.

void draw(View2d& view);
draws a B-Spline curve with the specified method and drawing parameters (or the default ones if they are not specified).

void draw_transformed(View2d& view, const Trans2d& trans);
draws the B-Spline curve as before but transformed according to trans (without affenting the original curve).

void transform(const Trans2d& trans);
transforms the B-Spline curve according to trans, as usual for all primitives in Gloop.

CONVERSION OPERATORS
A B-Spline curve cannot be converted to any other primitive defined in Gloop. A BSPline2d may be used in the place of a GBSPline2d or of Attributes as allowed by inheritance.

SEE ALSO
Attributes, **View2d**, **Trans2d**, **BSPline2d**.

AUTHOR
Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS
No known bugs.

NAME

GBezier3d

TYPE

Geometric primitive class.

DESCRIPTION

GBezier3d is the geometric primitive for the class of Bezier surfaces of any degree in three dimensions. Its drawable counterpart is Bezier3d. It is made of a GPoint3dArray which holds the control points of the surface.

INHERITANCE INFORMATION

Derived From

GObject3d

Base For

Bezier3d

CONSTRUCTORS

GBezier3d();

creates an empty Bezier surface.

GBezier3d(const GPoint3dArray& points);

creates a Bezier surface with control points specified in points. The number of rows and columns of the array (and thus the degrees of the surface for t and u respectively) is provided by the array itself.

GBezier3d(const GBezier3d& bezier);

The copy constructor.

ACCESS FUNCTIONS

int no_pts_t();
int no_pts_u();

return the number of control points of the Bezier surface for parameters t and u.

int deg_t();
int deg_u()

return the degree of the Bezier surface for t and u (always no_pts() - 1 for each parameter).

void control_pts(GPoint3dArray points);

returns the control points of the Bezier surface in points.

MEMBER FUNCTIONS

Box3d bet_bounding_box();

returns the bounding (or minmax) box of the Bezier surface as a Box3d.

GPoint3d centre();

returns the 'centre' of the surface as the coordinatewise average of its control points.

void subdivide(float t, float u, GBezier3d& top_left,
GBezier3d& top_right, GBezier3d& bottom_left,
GBezier3d& bottom_right);

subdivides the surface into four Bezier surfaces, at parameter values t and u. Mainly used in the implementation of the de Casteljau algorithm for drawing Bezier surfaces, but may have other uses.

int is_flat(float tolerance);

checks if the Bezier surface is flat within the given tolerance. Also used in the implementation of the de Casteljau algorithm.

void transform(const Trans3d& trans);

transforms the Bezier surface according to trans, as usual for all primitives in Gloop.

CONVERSION OPERATORS

A Bezier surface cannot be converted to any other primitive defined in Gloop.

OVERLOADED OPERATORS

GBezier3d& operator=(const GBezier3d& bezier)

The assignement operator for GBezier3ds.

SEE ALSO

GPoint3dArray, **Box3d**, **Trans3d**, **Bezier3d**.

AUTHOR

Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS

No known bugs.

NAME

Bezier3d

TYPE

Drawable primitive class.

DESCRIPTION

Bezier3d is the drawable primitive for the class of Bezier surfaces of any degree in three dimensions. It is the drawable counterpart of GBezier3d. A Bezier3d consists of two sub-objects of types GBezier3d and Attributes.

INHERITANCE INFORMATION

Derived From

GBezier3d
SimpleObject3d
Attributes

CONSTRUCTORS

For each way of specifying a Bezier3d there are two constructor functions. The first form constructs a primitive with default attribute values; the second with specified attribute values. Only the first form is described below. The parameter list will be the same for the second form except that an additional parameter of type 'const Attributes &' is required at the end. For example, an empty Bezier surface with default attributes can be constructed with
Bezier3d bez();
and a Bezier surface with specified attributes a can be constructed with
Bezier3d bez(a);

Bezier3d();

creates an empty Bezier surface.

Bezier3d(const GPoint3dArray& points);

constructs a Bezier surface with the points provided as control points. Information on the number of rows and columns of the array is provided be the GPoint3dArray class. The surface may have different degrees for t and u (different number of points for each direction).

Bezier3d(const Bezier3d& bezier);

The copy constructor. In is necessary because the default (memberwise) copy constructor did not work well with derived classes.

MEMBER FUNCTIONS

Bezier3d inherits the access and control functions of its base classes. Special member functions deal with the drawing of the Bezier surfaces.

void set_drawing_method(Bezier_algorithm alg);

specifies the algorithm to be used for drawing the Bezier surface. Two methods for drawing Bezier surfaces are implemented, based on the de Casteljau algorithm and on the use of differences; the choice that will give the best result may depend on the specific application. The variable alg is of an enumeration type defined for this purpose:
enum Bezier_algorithm {DE_CASTELJAU, DIFFERENCES};
The default drawing algorithm is the differences method.

void set_tolerance(float tol);

sets the tolerance for the flatness tests in the de Casteljau algorithm. Smaller values produce smoother surfaces. This setting is used only when the drawing method is DE_CASTELJAU. The default tolerance is 0.05

void set_eval_step_t(float h_t);
void set_eval_step_u(float h_u);

set the step at which the function representing the surface is evaluated for parameters t and u respectively. Smaller values produce smoother surfaces. This setting is used only when the drawing method is DIFFERENCES. The default evaluation step is 0.05 for both parameters. NOTE: It is advisable to use values that have integer reciprocal, for example 0.1 or 0.3333 but not 0.3; otherwise the curve will not end at its last control point. (The reciprocal of the evaluation step is the number of steps needed to run the range of t from 0 to 1. This behaviour is inherent in the algorithm.) Moreover, evaluation steps greater than 0.5 will have awkward results.

void draw(View3d& view);

draws a Bezier surface as a wireframe with the specified method and drawing parameters (or the default ones if they are not specified).

void draw_transformed(View3d& view, const Trans3d& trans);

draws the Bezier surface as before but transformed according to trans (without affenting the original surface).

void draw(View3d& view, Camera& cam, Lighting_model& model);

draws the Bezier surface using the camera and lighting model provided. This is the function to use for shading of Bezier surfaces [requires set_area_action(FillArea)]. Only the differences method can be used for this at present.

void transform(const Trans3d& trans);
transforms the Bezier surface according to trans, as usual for all primitives in Gloop.

CONVERSIONS

void convert_to_BRep_diff(BRep& brep);
converts the Bezier surface to the corresponding BRep using the differences method and the evaluation steps specified.

SEE ALSO

Attributes, **View3d**, **Trans3d**, **Bezier3d**, **Camera**, **Lighting_model**, **BRep**.

AUTHOR

Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS

Rendering of Bezier surfaces using a camera and lighting model can lead to segmentation fault for some values of the evaluation steps, the camera position etc. These do not occur with the simple drawing function.

NAME

GBSpline3d

TYPE

Geometric primitive class.

DESCRIPTION

GBSpline3d is the geometric primitive for the class of B-Spline surfaces of any degree and containing any number of control points, in three dimensions. Its drawable counterpart is BSpline3d. (See its manual page for restrictions on drawing arbitrary B-Splines.) It is made of a GPoint3dArray which holds the control points of the surface, and of two arrays of floats which are the knot sequences of the surface for parameters t and u.

INHERITANCE INFORMATION

Derived From

GObject3d

Base For

BSpline3d

CONSTRUCTORS

For each way of specifying a GBSPline3d (except for an empty GBSpline) there are two constructor functions. The first form constructs a primitive with default knot sequences, i.e. uniformly spaced and with the degree first and last knots equal; the second with specified knot sequence. Only the first form is described below. The parameter list will be the same for the second form except that two additional parameters of type 'const float*' are required at the end. For example, a bi-cubic B-Spline surface with control points stored in the array pts and default knot sequences can be constructed with
BSPline3d bez(3, 3, pts);
and a B-Spline surface with the same characteristics and specified knot sequences knots_t and knots_u can be constructed with
BSPline3d bez(3, 3, pts, knots_t, knots_u);
For the second form, the number of required knots in described in the appropriate access function. Also the knots must be in non-decreasing order or the construction is aborted.

GBSpline3d();

creates an empty B-Spline surface.

GBezier3d(const GPoint3dArray& points);

creates a B-Spline surface with control points specified in points. The number of rows and columns of the array is provided by the array itself.

GBSpline3d(const GBSpline3d& bspline);

The copy constructor.

ACCESS FUNCTIONS

int deg_t();
int deg_u();

return the degrees of the B-Spline surface for parameters t and u respectively.

int no_pts_t();
int no_pts_u();

return the number of control points of the B-Spline surface for t and u (equivalently, the number of rows and columns of the array of control points).

void control_pts(GPoint3dArray points);

returns the control points of the B-Spline surface in points.

int no_knts_t();
int no_knts_u();

return the number of knots of the B-Spline surface (always no_pts() - 1 + deg() for t and u respectively).

void knot_sequence_t(float* knot_seq);
void knot_sequence_u(float* knot_seq);

return the knot sequences of the surface in knot_seq (which must be of the appropriate size) for parameters t and u separately.

MEMBER FUNCTIONS

Box3d bet_bounding_box();

returns the bounding box of the B-Spline surface as a Box3d.

GPoint3d centre();

returns the 'centre' of the surface as the coordinatewise average of its control points.

void insert_knot_t(int index, const float knot_value);
void insert_knot_u(int index, const float knot_value);

insert a new knot with value knot_value in the interval starting with the knot indexed index, while retaining the shape of the surface, and update the control points and knot sequence of the surface. Insertion is only possible in intervals starting with index between deg() - 1 and no_pts() - 2 inclusive; also the resulting knot sequence must still be non-decreasing. If any of these conditions is not satisfied the insertion is aborted.

void insert_many_knots_t(int index, int no_knts, const float* knot_values);
void insert_many_knots_u(int index, int no_knts, const float* knot_values);
insert all no_knts knots contained in knot_values in the same interval starting with the knot indexed index, while retaining the shape of the surface, and update the control points and knot sequence of the surface. The same conditions as for single insertion must be satisfied.

void delete_knot_t(int index);
void delete_knot_t(int index);
delete the knot with given index, which must be between deg() and no_pts() - 1 inclusive. When removing a newly inserted knot the result is exactly the surface before the insertion; in any other case the resulting surface may be unpredictable!

void transform(const Trans3d& trans);
transforms the B-Spline surface according to trans, as usual for all primitives in Gloop.

void interpolate(const GPoint3dArray& pts, const GPoint3d* r1, const GPoint3d* r2, const GPoint3d* c1, const GPoint3d* c2, const GPoint3d r1c1, const GPoint3d r1c2, const GPoint3d r2c1, const GPoint3d r2c2);
void interpolate(const GPoint3dArray& pts, const float* knots_t, const float* knots_u, const GPoint3d* r1, const GPoint3d* r2, const GPoint3d* c1, const GPoint3d* c2, const GPoint3d r1c1, const GPoint3d r1c2, const GPoint3d r2c1, const GPoint3d r2c2);
construct a bi-cubic B-Spline surface to interpolate the given set of points. The B-Spline constructed with the first form has default (uniform) knot sequence. The number of knots for the second form must be no_pts + 4 for t and u respectively (the resulting B-Spline has (no_pts_t + 2) by (no_pts_u + 2) control points). The arbitrary points supplied in r1 and r2 are used as the second and the last but one control point respectively when interpolating each column. Those supplied in c1 and c2 are used as the second and the last but one control point respectively when interpolating each row. r1c1 and r2c1 are used when interpolating c1. r1c2 and r2c2 are used when interpolating c2.

CONVERSION OPERATORS
A GSpline3d cannot be converted to any other primitive defined in Gloop.

OVERLOADED OPERATORS
GBSpline3d& operator=(const GBSpline3d& bezier)
The assignement operator for GBSpline3ds.

SEE ALSO
GPoint3dArray, **Box3d**, **Trans3d**, **BSpline3d**.

AUTHOR
Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS
No known bugs.

NAME

BSPline3d

TYPE

Drawable primitive class.

DESCRIPTION

BSPline3d is the drawable primitive for the class of B-Spline surfaces in two dimensions. It is the drawable counterpart of GBSPline3d. A BSPline3d consists of two sub-objects of types GBSPline3d and Attributes.

INHERITANCE INFORMATION

Derived From

GBSPline3d
SimpleObject3d
Attributes

CONSTRUCTORS

For each way of specifying a BSPline3d (except for empty ones) there are four constructor functions. The first form constructs a primitive with default attribute values and default (uniform) knot sequences for both t and u; the second with specified attribute values and default knot sequences; the third with default attribute values and specified knot sequences; and the fourth with specified attribute values and specified knot sequences. Only the first form is described below. The parameter list will be the same for the other forms, with the following additions: for the second form of a parameter of type 'const Attributes&' at the end; for the third form of two parameters of type 'const float*' at the end holding the knot sequences for t and u respectively; and for the fourth form of two parameters of type 'const float*' and one of type 'const Attributes&' at the end. For example, a bicubic B-Spline surface with control points stored in the array pts and default knot sequence can be constructed with
BSPline3d bez(3, 3, pts);
and a B-Spline surface with the same characteristics, specified knot sequences knots_t and knots_u and drawing attributes a can be constructed with
BSPline3d bez(3, 3, pts, knots_t, knots_u, a);

BSPline3d();

creates an empty B-Spline surface.

BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points);

creates a B-Spline surface with control points given in points and degrees deg_t for t and deg_u for u. Information on the number of rows and columns of the array is provided be the GPoint3dArray class. The surface may have different degrees and different number of points for t and u.

BSPline3d(const BSPline3d &bezier);

The copy constructor. In is necessary because the default (memberwise) copy constructor did not work well with derived classes.

MEMBER FUNCTIONS

BSPline3d inherits the access and control functions of its base classes. Special member functions deal with the drawing of the B-Spline surfaces.

void set_drawing_method(BSpline_algorithm alg);

specifies the algorithm to be used for drawing the B-Spline surface. Two methods for drawing B-Spline surfaces are implemented, as piecewise Bezier surfaces and using the de Boor formula; the choice that will give the best result may depend on the specific application. The variable alg is of an enumeration type defined for this purpose:
enum BSpline_algorithm {AS_BEZIER, DE_BOOR};
The default drawing algorithm is the de Boor method. Drawing of B-Splines as piecewise Bezier surfaces is only implemented for quadratic and cubic B-Splines; for other degrees, the de Boor method is always used. NOTE: The de Boor method may attempt division by zero and thus produce an error with some knot sequences containing multiple knots other than the end knots. This should not be common with the knot sequences used in practice.

void set_Bezier_drawing_method(Bezier_algorithm alg);

sets the algorithm used to draw the Bezier surface segments. This setting is used only when AS_BEZIER is chosen as the drawing method of the B-Spline. For the default value, refer to the manual page for Bezier3d.

void set_decas_tolerance(const float tol);

sets the tolerance for the flatness tests in the de Casteljau algorithm. This setting is used only when the drawing method for the B-Spline is AS_BEZIER and the method for the sub-surfaces is DE_CASTELJAU. For the default value, refer to the manual page for Bezier3d.

void set_diff_eval_step_t(float h_t);
void set_diff_eval_step_u(float h_u);

set the steps at which the functions representing the Bezier surface segments are evaluated, for parameters t and u separately. This setting is used only when the drawing method for the B-Spline is AS_BEZIER and the method for the surface segments is DIFFERENCES. For the default values and an

important note on advisable values for the evaluation steps, refer to the manual page for Bezier3d.

void set_eval_step_t(float h_t);
void set_eval_step_u(float h_u);
sets the step at which the function representing the B-Spline surface is evaluated using the de Boor method, for parameters t and u separately. Smaller values produce smoother surfaces. This setting is only used when the drawing method for the B-Spline is DE_BOOR. The default values are 0.05. NOTE: Values greater than 0.5 will have awkward results.

void draw(View3d& view);
draws a B-Spline surface with the specified method and drawing parameters (or the default ones if they are not specified).

void draw_transformed(View3d& view, const Trans3d& trans);
draws the B-Spline surface as before but transformed according to trans (without affenting the original surface).

void draw(View3d& view, Camera& cam, Lighting_model& model);
draws the B-Spline surface using the camera and lighting model provided. This is the function to use for shading of S-Spline surfaces [requires set_area_action(FillArea)]. Only the de Boor method can be used for this at present.

void transform(const Trans3d& trans);
transforms the B-Spline surface according to trans, as usual for all primitives in Gloop.

CONVERSIONS
void convert_to_BRep_deboor(BRep& brep);
converts the B-Spline surface to the corresponding BRep using the de Boor method and the evaluation steps specified.

SEE ALSO
Attributes, **View3d**, **Trans3d**, **BSPline3d**, **Camera**, **Lighting_model**, **BRep**.

AUTHOR
Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS
No known bugs.

NAME

GPoint3dArray

TYPE

Auxiliary class.

DESCRIPTION

GPoint3dArray is a class representing a rectangular array of GPoint3ds. Such arrays hold the control points of GBezier3d and GBSpline3d so that the user of these classes does not have to manipulate 'GPoint3d**' directly. The class provides many facilities useful in the context of its use.

CONSTRUCTORS

GPoint3dArray();
creates an empty array of GPoint3ds.

GPoint3dArray(const int nr, const int nc);
creates an empty GPoint3dArray array of nr rows and nc columns.

GPoint3dArray(const GPoint3dArray& array);
The copy constructor. It is necessary because the class manipulates dynamically allocated storage.

ACCESS FUNCTIONS

int no_rows();
int no_cols();
return the number of rows and columns of the array respectively.

MEMBER FUNCTIONS

void get_cell(int i, int j, GPoint3d& point);
returns the value of cell (i, j) into point.

void set_cell(int i, int j, GPoint3d point);
sets cell (i, j) equal to point.

void get_row(int i, GPoint3d* row_points);
void get_col(int j, GPoint3d* col_points);
return the whole row (or column) of the array. (The user must ensure that the return variables are of the correct size.)

void set_row(int i, GPoint3d* row_points);
void set_col(int j, GPoint3d* col_points);
set the points in the whole row (or column) of the array equal to the points given. (The user must ensure that the variables are of the correct size.)

float x(int i, int j);
float y(int i, int j);
float z(int i, int j);
return the x, y and z coordinates of cell (i, j) of the array. [No 'set' counterpart of these is defined.]

int read(char* fname);
reads the GPoint3dArray from the file with name fname. Returns 1 for successful termination or 0 if an error occured during reading. The file must contain the number of rows, the number of columns and then the coordinates of the points stored by rows.

convert_to_BRep(BRep& brep);
returns in brep a BRep corresponding to the array of points. The BRep constructed has default attributes.

OVERLOADED OPERATORS

GPoint3dArray& operator=(const GPoint3dArray& array)
The assignement operator for GPoint3dArrays. As the copy constructor, it is necessary because the class manipulates dynamically allocated storage.

SEE ALSO

GBezier3d, **GBSpline3d**.

AUTHOR

Nikolaos Platis
MSc in Information Technology 1995 - 1996
UCL Department of Computer Science

BUGS

No known bugs.

APPENDIX B: HEADER FILES

The header files of the classes constructed are presented in this Appendix, separately from the accompanying code. This facilitates the use of header files as a reference source and enhances information hiding which is supposed to be provided by the class mechanism.

```
/* File tridiagonal.h
 *
 * Contains the declarations of the necessary functions
 * to solve tridiagonal systems, for cubic B-Spline interpolation.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


void l_u_analysis(int n, const float* alpha, const float* beta,
                  const float* gamma, float* low, float* up);
// Performs the LU analysis for a tridiagonal matrix (n x n)
// with lower diagonal 'alpha', diagonal 'beta' and upper diagonal 'gamma'.
// 'low' is the lower diagonal of L (it has diagonal of ones), and
// 'up' is the diagonal of U (is has upper diagonal gamma).


void solve_system(int n, const float* low, const float* up, const float* gamma,
                  const float* d, float* x);
// Solves the system Ax=d, where A (an n x n matrix) has been analysed
// in L and U by the previous function (see there for details).
// The solution is in x.
```

```
/* File operator2d.h
 *
 * Contains the (inline) definitions of the basic operators,
 * overloaded for GPoint2d.
 *
 * It is included in GCurve2d.cc and Curve2d.cc
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


inline GPoint2d operator+(const GPoint2d& point1, const GPoint2d& point2)
{
    // Addition operator for GPoint2d.

    return GPoint2d( point1.x() + point2.x(), point1.y() + point2.y() );
}


inline GPoint2d operator-(const GPoint2d& point1, const GPoint2d& point2)
{
    // Subtraction operator for GPoint2d.

    return GPoint2d( point1.x() - point2.x(), point1.y() - point2.y() );
}


inline GPoint2d operator*(const float t, const GPoint2d& point)
{
    // Scalar product for GPoint2d.

    return GPoint2d( t * point.x(), t * point.y() );
}
```

```
/* File GCurve2d.h
 *
 * Contains the declaration of classes GBezier2d and GBSpline2d, for the
 * representation of (the geometric primitives of) Bezier and B-Spline curves
 * in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


class GBezier2d : public GPrimitive2d  {
    private:
        int no_points;              // Number of control points.
        GPoint2d* control_points;   // Control points.

    public:
        // Constructors are provided to form the control polygon of the
        // Bezier curve from arrays of x and y coordinates, from an array
        // containing the points, or from a polypoint. Thanks to inheritance,
        // it can also be formed from a polyline or a polygon.
        // WARNING: The degree of the Bezier curve constructed is (no_points-1).
        GBezier2d();
        GBezier2d(int no_pts, const float* x, const float* y);
        GBezier2d(int no_pts, const GPoint2d* pts);
        GBezier2d(const GPolyPoint2d& polypoint);
        GBezier2d(const GBezier2d& bezier);    // Copy constructor.
        ~GBezier2d();

        GBezier2d& operator=(const GBezier2d& bezier);   // Assignement oper.

        // Member functions provide all the elements of the Bezier curve,
        // subdivide it and check it for flatness.
        // They also cater for its geometric transformations.
        inline int deg() const;
        inline int no_pts() const;
        void control_pts(GPoint2d* points) const;
        GPolyPoint2d convex_hull() const;
        GRectangle2d minmax_box() const;
        void subdivide(float t,
                       GBezier2d& bez_left, GBezier2d& bez_right) const;
        int is_flat(float tolerance) const;

        void transform(const Trans2d& trans);
};


/*****************************************************************************/


class GBSpline2d : public GPrimitive2d  {
    private:
        int degree;                 // Degree of the B-Spline curve.
        int no_points;              // Number of control points.
        GPoint2d* control_points;   // Control points of the B-Spline curve.
        int no_knots;       // Number of knots (obtained by degree & no_points).
        float* knots;       // Knots of the B-Spline curve.

    public:
        // The control points of the B-Spline curve may be given as arrays of
        // x and y coordinates, as an array of points, or as a polypoint.
        // The number of knots is always (no_points - 1 + degree).
        // If knots are not given, they are taken uniformly spaced with the
        // (degree) first knots and the (degree) last knots equal.
        // WARNING: All indices run from 0 to n-1 (n as appropriate).
        GBSpline2d();
        GBSpline2d(int deg, int no_pts, const float* x, const float* y);
        GBSpline2d(int deg, int no_pts, const float* x, const float* y,
                   const float* knot_sequence);
        GBSpline2d(int deg, int no_pts, const GPoint2d* pts);
        GBSpline2d(int deg, int no_pts, const GPoint2d* pts,
                   const float* knot_sequence);
        GBSpline2d(int deg, const GPolyPoint2d& polypoint);
        GBSpline2d(int deg, const GPolyPoint2d& polypoint,
                   const float* knot_sequence);
        GBSpline2d(const GBSpline2d& bspline);   // Copy constructor.
        ~GBSpline2d();

        GBSpline2d& operator=(const GBSpline2d& bspline);   // Assignement oper.
        // Member functions providing the elements of the B-Spline curve.
        inline int deg() const;
        inline int no_pts() const;
        void control_pts(GPoint2d* points) const;
        inline int no_knts() const;
        void knot_sequence(float* knot_seq) const;

        // Insertion of knot_value in the interval with lower bound index
        // 'index'. (Boehm's knot insertion algorithm.)
        void insert_knot(int index, const float knot_value);

        // Insertion of multiple knots in the same interval (Oslo algorithm).
        void insert_many_knots(int index,
                               int no_knts, const float* knot_values);

        // Deletion of the knot indexed knot_index.
        void delete_knot(int index);

        // Transformation of the B-Spline curve.
        void transform(const Trans2d& trans);

        // Interpolation with cubic (only) B-Splines.
        // The number of knots needed is (no_pts + 4) [the resulting B-Spline
        // has (no_pts + 2) control points]. It is up to the programmer to
        // ensure that the first and the last three are equal.
        // Again the default is uniform knot sequence.
        // r1 and r2 are the arbitrary points needed.
        void interpolate(const int no_pts, const float* x, const float* y,
```

```
                    const GPoint2d r1, const GPoint2d r2);
        void interpolate(const int no_pts, const float* x, const float* y,
                    const float* knots,
                    const GPoint2d r1, const GPoint2d r2);
        void interpolate(const int no_pts, const GPoint2d* pts,
                    const GPoint2d r1, const GPoint2d r2);
        void interpolate(const int no_pts, const GPoint2d* pts,
                    const float* knots,
                    const GPoint2d r1, const GPoint2d r2);
        void interpolate(const GPolyPoint2d& polypoint,
                    const GPoint2d r1, const GPoint2d r2);
        void interpolate(const GPolyPoint2d& polypoint,
                    const float* knots,
                    const GPoint2d r1, const GPoint2d r2);

        // An auxiliary function for the drawing of B-Splines.
        // Computes the de Boor point for parameter value t in the interval
        // with lower end index 'index'.
        friend GPoint2d deBoor_point(const GBSpline2d& bspline,
                                     float t, int index);
};
```

```
/* File Curve2d.h
 *
 * Contains the declaration of classes Bezier2d and BSpline2d, for the
 * representation of (the drawable primitives of) Bezier and B-Spline curves
 * in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


enum Bezier_algorithm { DE_CASTELJAU, DIFFERENCES };

class Bezier2d : public Primitive2d, public GBezier2d  {
    private:
        Bezier_algorithm method;
        float tolerance;   // Tolerance for flatness tests in the de Casteljau
                           // drawing method (used when method == DE_CASTELJAU).
        float eval_step;   // Evaluation step h for the differences method
                           // (used when method == DIFFERENCES).
        // Default values: method = DIFFERENCES,
        //                 tolerance = 0.01,
        //                 eval_step = 0.01.

        // The two alternative drawing methods.
        void draw_decas(View2d& view) const;
        void draw_diff(View2d& view) const;

    public:
        // Constructors are equivalent to the ones of GBezier2d. The forms
        // without Attribute parameter initialise to default attributes.
        Bezier2d();
        Bezier2d(const Attributes& attributes);
        Bezier2d(const GBezier2d& bezier);
        Bezier2d(const GBezier2d& bezier, const Attributes& attributes);
        Bezier2d(int no_pts, const float* x, const float* y);
        Bezier2d(int no_pts, const float* x, const float* y,
                 const Attributes& attributes);
        Bezier2d(int no_pts, const GPoint2d* pts);
        Bezier2d(int no_pts, const GPoint2d* pts, const Attributes& attributes);
        Bezier2d(const GPolyPoint2d& polypoint);
        Bezier2d(const GPolyPoint2d& polypoint, const Attributes& attributes);
        Bezier2d(const Bezier2d& bezier);   // Copy constructor.

        // Functions to define drawing method and relevant parameters.
        inline void set_drawing_method(const Bezier_algorithm alg);
        inline void set_tolerance(const float tol);
        inline void set_eval_step(const float h);

        // Transformation and drawing of the Bezier curve.
        // Other member functions are inherited from GBezier2d.
        void transform(const Trans2d& trans);
        void draw(View2d& view) const;
        void draw_transformed(View2d& view, const Trans2d& trans) const;
};


/*****************************************************************************/


enum BSpline_algorithm { DE_BOOR, AS_BEZIER };
// Conversion to Bezier curve segments is only possible for quadratic and cubic
// B-Splines. For higher degrees the de Boor method is always used. Also this
// method is not applicable to B-Splines with multiple knots with multiplicity
// equal to (deg) other than the end ones.

class BSpline2d : public Primitive2d, public GBSpline2d  {
    private:
        BSpline_algorithm method;
        Bezier_algorithm Bezier_method;   // Used when method == AS_BEZIER.
        float decas_tolerance;   // Refer to the relevant parameters of the
        float diff_eval_step;    // Bezier2d class (when method == AS_BEZIER).
        float eval_step;   // Evaluation step for the de Boor algorithm; t is
                           // supposed in [0, 1]. (Used when method == DE_BOOR)
        // Default values: method = DE_BOOR,
        //                 eval_step = 0.01,
        //                 the defaults for AS_BEZIER drawing as in Bezier2d.

        // The two alterative drawing methods.
        void draw_bez(View2d& view) const;
        void draw_deboor(View2d& view) const;

    public:
        // Constructors are equivalent to the ones of GBezier2d. The forms
        // without Attribute parameter initialise to default attributes.
        BSpline2d();
        BSpline2d(const Attributes& attributes);
        BSpline2d(const GBSpline2d& bspline);
        BSpline2d(const GBSpline2d& bspline, const Attributes& attributes);
        BSpline2d(int deg, int no_pts, const float* x, const float* y);
        BSpline2d(int deg, int no_pts, const float* x, const float* y,
                  const Attributes& attributes);
        BSpline2d(int deg, int no_pts, const float* x, const float* y,
                  const float* knot_sequence);
        BSpline2d(int deg, int no_pts, const float* x, const float* y,
                  const float* knot_sequence, const Attributes& attributes);
        BSpline2d(int deg, int no_pts, const GPoint2d* pts);
        BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                  const Attributes& attributes);
        BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                  const float* knot_sequence);
        BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                  const float* knot_sequence, const Attributes& attributes);
        BSpline2d(int deg, const GPolyPoint2d& polypoint);
        BSpline2d(int deg, const GPolyPoint2d& polypoint,
                  const Attributes& attributes);
        BSpline2d(int deg, const GPolyPoint2d& polypoint,
```

```
                const float* knot_sequence);
        BSpline2d(int deg, const GPolyPoint2d& polypoint,
                const float* knot_sequence, const Attributes& attributes);
        BSpline2d(const BSpline2d& bspline);   // Copy constructor.

        // Functions to set the drawing method and relevant parameters.
        inline void set_drawing_method(BSpline_algorithm alg);
        inline void set_Bezier_drawing_method(Bezier_algorithm alg);
        inline void set_decas_tolerance(const float tol);
        inline void set_diff_eval_step(const float h);
        inline void set_eval_step(const float h);

        // Transformation and drawing of the B-Spline curve.
        void transform(const Trans2d& trans);
        void draw(View2d& view) const;
        void draw_transformed(View2d& v, const Trans2d& trans) const;
};
```

```
/* File GSurface3d.h
 *
 * Contains the declaration of classes GBezier3d and GBSpline3d, for the
 * representation of (the geometric primitives of) Bezier and B-Spline surfaces
 * in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


class GPoint3d;
class BRep;

class GPoint3dArray  {
    private:
        int n_rows, n_columns;
        GPoint3d** points;

    public:
        GPoint3dArray();
        GPoint3dArray(const int nr, const int nc);
        GPoint3dArray(const GPoint3dArray& array);   // Copy constructor.
        ~GPoint3dArray();

        GPoint3dArray& operator=(const GPoint3dArray& array);   // Assignement
        void convert_to_BRep(BRep& brep) const;   // Conversion to BRep.

        inline int no_rows() const;
        inline int no_cols() const;
        inline void get_cell(int i, int j, GPoint3d& point) const;
        inline void set_cell(int i, int j, GPoint3d point);
        inline float x(int i, int j) const;
        inline float y(int i, int j) const;   // Return coordinates
        inline float z(int i, int j) const;   // of cell(i, j).
        void get_row(int i, GPoint3d* row_points) const;
        void get_col(int j, GPoint3d* col_points) const;
        void set_row(int i, GPoint3d* row_points);
        void set_col(int j, GPoint3d* col_points);
        int read(char* fname);   // Read GPoint3dArray from file.
};


/*****************************************************************************/


class GBezier3d : public virtual GObject3d  {
    private:
        int no_points_t;             // Number of control points for t.
        int no_points_u;             // Number of control points for u.
        GPoint3dArray control_points;   // Control points.

    public:
        // The only constructor provided accepts the control points as an array
        // of GPoint3d (GPoint3dArray). This class also provides information on
        // the number of rows (corresponding to parameter t) and columns
        // (corresponding to parameter u).
        GBezier3d();
        GBezier3d(const GPoint3dArray& points);
        GBezier3d(const GBezier3d& bezier);    // Copy constructor.
        // No destructor necessary here.

        GBezier3d& operator=(const GBezier3d& bezier);   // Assignement oper.

        // Member functions provide all the elements of the Bezier surface,
        // subdivide it and check it for flatness.
        // They also cater for its geometric transformations.
        // No convex hull provided for surfaces (too complicated).
        inline int deg_t() const;
        inline int deg_u() const;
        inline int no_pts_t() const;
        inline int no_pts_u() const;
        void control_pts(GPoint3dArray& points) const;
        Box3d get_bounding_box() const;
        GPoint3d centre() const;
        void subdivide(float t, float u,
                       GBezier3d& top_left, GBezier3d& top_right,
                       GBezier3d& bottom_left, GBezier3d& bottom_right) const;
        int is_flat(float tolerance) const;

        void transform(const Trans3d& trans);
};


/*****************************************************************************/


class GBSpline3d : public virtual GObject3d  {
    private:
        int degree_t;                  // Degree of the B-Spline surface for t.
        int degree_u;                  // Degree of the B-Spline surface for u.
        int no_points_t;               // Number of control points for t.
        int no_points_u;               // Number of control points for u.
        GPoint3dArray control_points;  // Control points of the B-Spline surface
        int no_knots_t;                // Number of knots for t.
        int no_knots_u;                // Number of knots for u.
        float* knots_t;                // Knots for t of the B-Spline surface.
        float* knots_u;                // Knots for u of the B-Spline surface.

    public:
        // The control points of the B-Spline surface can only be given as a
        // rectangular array of points (GPoint3dArray).
        // The number of knots is always (no_pts - 1 + deg) for t and u.
        // If knots are not given, they are taken uniformly spaced with the
        // (deg) first knots and the (deg) last knots equal.
        // WARNING: All indices run from 0 to n-1 (n as appropriate).
        GBSpline3d();
```

```
GBSpline3d(int deg_t, int deg_u, const GPoint3dArray& points);
GBSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
           const float* knot_seq_t, const float* knot_seq_u);
GBSpline3d(const GBSpline3d& bspline);   // Copy constructor.
~GBSpline3d();

GBSpline3d& operator=(const GBSpline3d& bspline);   // Assignement oper.
// Member functions providing the elements of the B-Spline surface.
inline int deg_t() const;
inline int deg_u() const;
inline int no_pts_t() const;
inline int no_pts_u() const;
void control_pts(GPoint3dArray& points) const;
inline int no_knts_t() const;
inline int no_knts_u() const;
void knot_sequence_t(float* knot_seq) const;
void knot_sequence_u(float* knot_seq) const;

// Insertion of knot_value in the interval with lower bound index
// 'index'. (Boehm's knot insertion algorithm.)
void insert_knot_t(int index, const float knot_value);
void insert_knot_u(int index, const float knot_value);

// Multiple knot insertion (Oslo algorithm).
void insert_many_knots_t(int index,
                         int no_knts, const float* knot_values);
void insert_many_knots_u(int index,
                         int no_knts, const float* knot_values);

// Deletion of the knot indexed knot_index.
void delete_knot_t(int knot_index);
void delete_knot_u(int knot_index);

// Functions inherited as virtual.
Box3d get_bounding_box() const;
GPoint3d centre() const;

// Transformation of the B-Spline surface.
void transform(const Trans3d& trans);

// Interpolation with cubic (only) B-Spline surfaces.
// The number of knots needed for t and u respectively is (no_pts + 4)
// [the resulting B-Spline has (no_pts + 2) control points]. It is up to
// the programmer to ensure that the first and the last three are equal.
// r1, r2, c1, c2 are the arrays of arbitrary points needed, and
// r1c1, r1c2, r2c1, r2c2 are four more arbitrary points needed.
void interpolate(const GPoint3dArray& pts,
                 const GPoint3d* r1, const GPoint3d* r2,
                 const GPoint3d* c1, const GPoint3d* c2,
                 const GPoint3d r1c1, const GPoint3d r1c2,
                 const GPoint3d r2c1, const GPoint3d r2c2);
void interpolate(const GPoint3dArray& pts,
                 const float* knots_t, const float* knots_u,
                 const GPoint3d* r1, const GPoint3d* r2,
                 const GPoint3d* c1, const GPoint3d* c2,
                 const GPoint3d r1c1, const GPoint3d r1c2,
                 const GPoint3d r2c1, const GPoint3d r2c2);
};
```

```
/* File Surface3d.h
 *
 * Contains the declaration of classes Bezier3d and BSpline3d, for the
 * representation of (the drawable primitives of) Bezier and B-Spline surfaces
 * in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


enum Bezier_algorithm { DE_CASTELJAU, DIFFERENCES };

class Bezier3d : public GBezier3d, public SimpleObject3d, public Attributes  {
    private:
        Bezier_algorithm method;
        float tolerance;   // Tolerance for flatness tests in the de Casteljau
                           // algorithm (used when method == DE_CASTELJAU).
        float eval_step_t, eval_step_u;
        // Evaluation steps (for t and u) used when method == DIFFERENCES.
        // Default values: method = DIFFERENCES,
        //                 tolerance = 0.05,
        //                 eval_step_t = eval_step_u = 0.05.

        // The functions that carry out the drawing methods.
        void draw_decas(View3d& view) const;
        void draw_diff(View3d& view) const;
        void draw_diff(View3d& view, Camera& cam, Lighting_model& model) const;

    public:
        // Constructors are equivalent to the ones of GBezier3d. The forms
        // without Attribute parameter initialise to default attributes.
        Bezier3d();
        Bezier3d(const GBezier3d& bezier);
        Bezier3d(const Attributes& attributes);
        Bezier3d(const GBezier3d& bezier, const Attributes& attributes);
        Bezier3d(const GPoint3dArray& points);
        Bezier3d(const GPoint3dArray& points, const Attributes& attributes);
        Bezier3d(const Bezier3d& bezier);   // Copy constructor.

        // Conversion to BRep, using the differences method.
        void convert_to_BRep_diff(BRep& brep) const;

        // Functions to define drawing method and relevant parameters.
        inline void set_drawing_method(const Bezier_algorithm alg);
        inline void set_tolerance(const float tol);
        inline void set_eval_step_t(const float h_t);
        inline void set_eval_step_u(const float h_u);

        // Transformation and drawing of the Bezier surface.
        // Other member functions are inherited from GBezier3d.
        void transform(const Trans3d& trans);
        void draw(View3d& view) const;
        void draw(View3d& view, Camera& cam, Lighting_model& model) const;
        void draw_transformed(View3d& view, const Trans3d& trans) const;
};


/*****************************************************************************/


enum BSpline_algorithm { DE_BOOR, AS_BEZIER };

class BSpline3d : public GBSpline3d, public SimpleObject3d, public Attributes  {
    private:
        BSpline_algorithm method;
        Bezier_algorithm Bezier_method;   // Used when method == AS_BEZIER.
        float decas_tolerance;
        float diff_eval_step_t, diff_eval_step_u;
        // These refer to the relevant parameters of the Bezier3d class.
        float eval_step_t, eval_step_u;
        // Evaluation steps (for t and u) used when method == DE_BOOR.
        // Default values: method = DE_BOOR,
        //                 eval_step_t = eval_step_u = 0.05,
        //                 the defaults for AS_BEZIER drawing as in Bezier2d.

        // The functions that do all the work for the drawing.
        void draw_bez(View3d& view) const;
        void draw_deboor(View3d& view) const;
        void draw_deboor(View3d& view, Camera& cam,
                         Lighting_model& model) const;

    public:
        // Constructors are equivalent to the ones of GBezier3d. The forms
        // without Attribute parameter initialise to default attributes.
        BSpline3d();
        BSpline3d(const GBSpline3d& bspline);
        BSpline3d(const Attributes& attributes);
        BSpline3d(const GBSpline3d& bspline, const Attributes& attributes);
        BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points);
        BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                  const Attributes& attributes);
        BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                  const float* knot_seq_t, const float* knot_seq_u);
        BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                  const float* knot_seq_t, const float* knot_seq_u,
                  const Attributes& attributes);
        BSpline3d(const BSpline3d& bspline);   // Copy constructor.

        // Conversion to BRep, using the de Boor method.
        void convert_to_BRep_deboor(BRep& brep) const;

        // Functions to set the drawing method and relevant parameters.
        inline void set_drawing_method(BSpline_algorithm alg);
        inline void set_Bezier_drawing_method(Bezier_algorithm alg);
        inline void set_decas_tolerance(const float tol);
        inline void set_diff_eval_step_t(const float h_t);
```

```
        inline void set_diff_eval_step_u(const float h_u);
        inline void set_eval_step_t(const float h_t);
        inline void set_eval_step_u(const float h_u);

        // Transformation and drawing of the B-Spline surface.
        void transform(const Trans3d& trans);
        void draw(View3d& view) const;
        void draw(View3d& view, Camera& cam, Lighting_model& model) const;
        void draw_transformed(View3d& view, const Trans3d& trans) const;
};
```

APPENDIX C: PROGRAM LISTINGS

This final Appendix contains the listings of the classes implementing Bézier and B-Spline curves and surfaces. The order of presentation is the same as for the respective header files in Appendix B.

Credits

Farin [1] provided a very simple implementation of several algorithms needed for this Project; they were all expanded and adapted to fit in the classes constructed. His book has also been helpful for the implementation of some obscure parts of the differences algorithm. More information can be found inside each function concerned.

The `GPoint3dArray` class evolved from a basic class for two-dimensional arrays in the course notes on *Introduction to Programming* by Ben Bacarisse.

```
/* File tridiagonal.cc
 *
 * Implements header file tridiagonal.h, containing the necessary functions
 * to solve tridiagonal systems, for cubic B-Spline interpolation.
 *
 * Compilation: By itself with g++.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


void l_u_analysis(int n, const float* alpha, const float* beta,
                  const float* gamma, float* low, float* up)
{
    // Performs the LU analysis for a tridiagonal matrix (n x n)
    // with lower diagonal alpha, diagonal beta and upper diagonal gamma.
    // low is the lower diagonal of L (it has diagonal of ones), and
    // up is the diagonal of U (is has upper diagonal gamma).
    // Main idea from Farin, Curves and Surfaces for CAGD, section 9.6

    up[0]=beta[0];
    for(int i = 1; i < n; i++)  {
        low[i] = alpha[i] / up[i-1];
        up[i] = beta[i] - low[i]*gamma[i-1];
    }
}


void solve_system(int n, const float* low, const float* up, const float* gamma,
                  const float* d, float* x)
{
    // Solves the system Ax=d, where A (an n x n matrix) has been analysed
    // in L and U by the previous function (see there for details).
    // The solution is in x.
    // Main idea from Farin, Curves and Surfaces for CAGD, section 9.6

    // From LUx=d, set Ux=y and solve Ly=d first (forward substitution).
    float* y = new float[n];
    y[0] = d[0];
    for (int i = 1; i < n; i++)
        y[i] = d[i] - low[i]*y[i-1];

    // Then solve Ux=y (backward substitution).
    x[n-1] = y[n-1] / up[n-1];
    for (int j = n-2; j >= 0; j--)
        x[j] = (y[j] - gamma[j]*x[j+1]) / up[j];

    delete[] y;
}
```

```
/* File GCurve2d.cc
 *
 * Implements header file GCurve2d.h containing classes GBezier2d and
 * GBSpline2d, for the representation of (the geometric primitives of) Bezier
 * and B-Spline curves in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


#include "GPrimitive2d.h"
#include "Trans2d.h"
#include "GCurve2d.h"
#include "operator2d.h"
#include "tridiagonal.h"
#include <math.h>
#include <iostream.h>


GBezier2d::GBezier2d()
{
    // Constructor for a null Bezier curve.

    no_points = 0;
    control_points = 0;   // Null pointer.
}


GBezier2d::GBezier2d(int no_pts, const float* x, const float* y)
{
    // Constructor with vertices of control polygon specified by coordinates.

    no_points = no_pts;

    if (no_points > 0)  {
        control_points = new GPoint2d[no_points];
        for (int i = 0; i < no_points; i++)
            control_points[i] = GPoint2d(x[i], y[i]);
    }
}


GBezier2d::GBezier2d(int no_pts, const GPoint2d* pts)
{
    // Constructor for vertices of control polygon given as points.

    no_points = no_pts;

    if (no_points > 0)  {
        control_points = new GPoint2d[no_points];
        for (int i = 0; i < no_points; i++)
            control_points[i] = pts[i];
    }
}


GBezier2d::GBezier2d(const GPolyPoint2d& polypoint)
{
    // Constructor for control polygon specified as polypoint.

    no_points = polypoint.no_pts();

    if (no_points > 0)  {
        control_points = new GPoint2d[no_points];
        polypoint.points(control_points);
    }
}


GBezier2d::GBezier2d(const GBezier2d& bezier)
{
    // Copy contstructor.
    // cf. Stroustrup, section 7.6

    no_points = bezier.no_points;

    if (no_points > 0)  {
        control_points = new GPoint2d[no_points];
        bezier.control_pts(control_points);
    }
}


GBezier2d::~GBezier2d()
{
    // Destructor for GBezier2d.

    if (no_points > 0)
        delete[] control_points;
}


GBezier2d& GBezier2d::operator=(const GBezier2d& bezier)
{
    // Assignement operator for GBezier2d.
    // cf. Stroustrup, section 7.6

    if (this != &bezier)  {
        if (no_points > 0)
            delete[] control_points;

        no_points = bezier.no_points;

        if (no_points > 0)  {
            control_points = new GPoint2d[no_points];
            bezier.control_pts(control_points);
```

```
        }
    }
    return *this;
}


int GBezier2d::deg() const
{
    // Returns the degree of the Bezier curve.
    // A curve of degree n requires n+1 control points.

    return (no_points - 1);
}


int GBezier2d::no_pts() const
{
    // Returns the number of control points.

    return no_points;
}


void GBezier2d::control_pts(GPoint2d* points) const
{
    // points returns a pointer to the array of the control points of the curve.

    for (int i = 0; i < no_points; i++)
        points[i] = control_points[i];
}


GPolyPoint2d GBezier2d::convex_hull() const
{
    // Returns the convex hull of the Bezier curve, as a polypoint.
    // This is algorithm 8.1 - 8.2 in Edelsbrunner, 'Algorithms in Combinatorial
    // Geometry'.

    if (no_points == 0)
        cerr << "Trying to find the convex hull of an empty Bezier curve."
             << endl << endl;

    // Check for three or less points in the control polygon (trivial case).
    if (no_points <= 3)
        return GPolyPoint2d(no_points, control_points);

    int i, j;   // Used in for loops below.

    // Copy control polygon to use it.
    GPoint2d* points = new GPoint2d[no_points];
    for (i = 0; i < no_points; i++)
        points[i] = control_points[i];

    // Sort control points on their x coordinate. Selection sort is used.
    for (i = 0; i < no_points-1; i++)  {
        // Find minimum of remaining elements.
        int index_min = i;
        GPoint2d min = points[i];
        for (j = i+1; j < no_points; j++)
            if (points[j].x() < min.x())  {
                index_min = j;
                min = points[j];
            }

        // Exchange it with current element.
        points[index_min] = points[i];
        points[i] = min;
    }

    // This function checks if pt is to the right of the line
    // from points[j1] to points[j2].
    int is_outside(const GPoint2d& pt, GPoint2d* points, int j1, int j2);

    GPoint2d* hull = new GPoint2d[no_points];
    // At most no_points points in the convex hull.
    int no_points_hull = 3;   // Number of points actually in convex hull.

    // The first three points are inserted counterclockwise.
    for (i = 0; i < 3; i++)
        hull[i] = points[i];
    if (is_outside(hull[2], hull, 0, 1))  {   // Points are clockwise.
        GPoint2d temp = hull[0];              // Exchange two to turn them
        hull[0] = hull[1];                    // counterclockwise.
        hull[1] = temp;
    }

    // Main loop. The hull is considered stored in circular list.
    for (i = 3; i < no_points; i++)  {
        // Find 'top' point.
        j = no_points_hull - 1;
        int j_next = (j+1) % no_points_hull;
        while (is_outside(points[i], hull, j, j_next))  {
            j = j_next;
            j_next = (j+1) % no_points_hull;
        }
        const int index_top = j;

        // Find 'bottom' point.
        j = no_points_hull - 1;
        int j_prev = (j-1) % no_points_hull;
        while (! is_outside(points[i], hull, j, j_prev))  {
            j = j_prev;
            j_prev = (j-1) % no_points_hull;
        }
        const int index_bottom = j;

        // Update convex hull. Insert the points so that the last point in the
        // array is the last one added.
```

```
        GPoint2d* new_hull = new GPoint2d[no_points];
        int no_points_new_hull = 0;
        if (index_top < index_bottom)
            for (j = index_top; j <= index_bottom; j++)  {
                new_hull[no_points_new_hull] = hull[j];
                no_points_new_hull++;
            }
        else  {
            for (j = index_top; j < no_points_hull; j++)  {
                new_hull[no_points_new_hull] = hull[j];
                no_points_new_hull++;
            }
            for (j = 0; j <= index_bottom; j++)  {
                new_hull[no_points_new_hull] = hull[j];
                no_points_new_hull++;
            }
        }
        new_hull[no_points_new_hull] = points[i];
        no_points_new_hull++;

        // Copy new_hull back to hull, to go on to the next step.
        for (j = 0; j < no_points_new_hull; j++)
            hull[j] = new_hull[j];
        no_points_hull = no_points_new_hull;

        delete[] new_hull;
    }

    GPolyPoint2d h(no_points_hull, hull);
    delete[] points;
    delete[] hull;

    return h;
}


int is_outside(const GPoint2d& pt, GPoint2d* points, int j1, int j2)
{
    // Checks if the point pt is 'outside' (to the right) of the line with
    // ends points[j1], points[j2].

    float x1 = points[j1].x(),    y1 = points[j1].y(),
          x2 = points[j2].x(),    y2 = points[j2].y(),
          x = pt.x(),             y = pt.y();

    if ( ((y-y1)*(x2-x1) - (x-x1)*(y2-y1)) >= 0 )
        return ( (x2 > x1) ? 1 : 0 );
    else
        return ( (x2 > x1) ? 0 : 1 );
}


GRectangle2d GBezier2d::minmax_box() const
{
    // Returns the minmax box of the Bezier curve, as a rectangle.
    // Reports an error for an empty Bezier curve.

    if (no_points == 0)
        cerr << "Trying to find the minmax box of an empty Bezier curve."
             << endl << endl;

    float xmin = control_points[0].x(), xmax = control_points[0].x(),
          ymin = control_points[0].y(), ymax = control_points[0].y();

    for (int i = 1; i < no_points; i++)  {
        float current_x = control_points[i].x();
        if (current_x < xmin)
            xmin = current_x;
        else if (current_x > xmax)
            xmax = current_x;

        float current_y = control_points[i].y();
        if (current_y < ymin)
            ymin = current_y;
        else if (current_y > ymax)
            ymax = current_y;
    }

    return GRectangle2d(xmin, ymin, xmax, ymax);
}


void GBezier2d::subdivide(float t,
                          GBezier2d& bez_left, GBezier2d& bez_right) const
{
    // Subdivides the Bezier curve at parameter value t.
    // Main idea from Farin, Curves and Surfaces for CAGD, section 4.9

    GPoint2d* right = new GPoint2d[no_points];  // Temporary storage of the
    GPoint2d* left = new GPoint2d[no_points];   // new points.

    const float t1 = 1 - t;
    int r, i;   // Used in the for loops below.

    // Get the right half first. Each step of the de Casteljau algorithm as
    // implemented below overwrites all but the rightmost control point with
    // the new ones; so at the end we get the right half polygon.

    for (i = 0; i < no_points; i++)
        right[i] = control_points[i];   // Initialise right half.

    for (r = 1; r < no_points; r++)
        for (i = 0; i < no_points - r; i++)
            right[i] = t1*right[i] + t*right[i+1];   // De Casteljau step.

    bez_right = GBezier2d(no_points, right);

    // Then get the left half. Essentially reverse the order of the control
```

```
    // points to keep the leftmost control point at each step.
    // The left half control points are then in reverse order; it does not
    // matter since Bezier curves are symmetric.

    for (i = 0; i < no_points; i++)
        left[i] = control_points[no_points - i - 1];   // Initialise left half.

    for (r = 1; r < no_points; r++)
        for (i = 0; i < no_points - r; i++)
            left[i] = t*left[i] + t1*left[i+1];   // De Casteljau step.

    bez_left = GBezier2d(no_points, left);

    delete[] right;
    delete[] left;
}


int GBezier2d::is_flat(float tolerance) const
{
    // Checks if the Bezier curve is flat (in 2d a straight line) with
    // given tolerance. Essentially checks the distance of every control point
    // from the line through the two end control points.
    // Main idea from Farin, Curves and Surfaces for CAGD, section 4.9

    const float x1 = control_points[0].x(),
                y1 = control_points[0].y(),
                x2 = control_points[no_points - 1].x(),
                y2 = control_points[no_points - 1].y();   // The end points.

    const float length = sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
    // Length of line between the two end points.

    if (length < 0.0001)  {   // Consider the two end points coincide.
        for (int i = 1; i < no_points-1; i++)  {
            const float xi = control_points[i].x(),
                        yi = control_points[i].y();
            const float dist = sqrt((x2 - xi)*(x2 - xi) + (y2 - yi)*(y2 - yi));
            if (dist > tolerance)
                return 0;
        }
        return 1;
    }
    else  {
        const float tmp = x1*y2 - x2*y1;
        for (int i = 1; i < no_points-1; i++)  {
            const float xi = control_points[i].x(),
                        yi = control_points[i].y();
            const float det = xi*(y1 - y2) - yi*(x1 - x2) + tmp;
            const float dist = det / length;
            if (fabs(dist) > tolerance)
                return 0;
        }
        return 1;
    }
}


void GBezier2d::transform(const Trans2d& trans)
{
    // Transforms the Bezier curve. Exploits the fact that it is invariant
    // under affine maps, and all basic transformations are affine maps; so to
    // transform the curve we only need to transform its control points.
    // Exception: Projection is *not* an affine map.

    for (int i = 0; i < no_points; i++)
        control_points[i].transform(trans);
}


/******************************************************************************/


GBSpline2d::GBSpline2d()
{
    // Constructor for a null B-Spline curve.

    degree = 0;
    no_points = 0;
    control_points = 0;   // Null pointer.
    no_knots = 0;
    knots = 0;   // Null pointer.
}


GBSpline2d::GBSpline2d(int deg, int no_pts, const float* x, const float* y)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates and default knot sequence.

    degree = deg;
    no_points = no_pts;
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        if (degree > no_points-1)  {
            degree = no_points - 1;
            no_knots = no_points - 1 + degree;
            cerr << "Not enough control points. A B-Spline of degree "
                 << degree << " will be constructed." << endl << endl;
        }

        int i;   // Used in for loops below.

        control_points = new GPoint2d[no_points];
        for (i = 0; i < no_points; i++)
            control_points[i] = GPoint2d(x[i], y[i]);
```

```
        knots = new float[no_knots];
        for (i = 0; i < degree; i++)
            knots[i] = 0;
        for (i = degree; i < no_points-1; i++)
            knots[i] = i - degree + 1;
        for (i = no_points-1; i < no_knots; i++)
            knots[i] = no_points - degree;
    }
}


GBSpline2d::GBSpline2d(int deg, int no_pts, const float* x, const float* y,
                       const float* knot_sequence)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates and given knot sequence.

    degree = deg;
    no_points = no_pts;
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knots, knot_sequence))  {
            cerr << "Knot sequence must be non-decreasing. "
                 << "Construction of B-Spline failed." << endl << endl;
        }

        else  {
            if (degree > no_points-1)  {
                degree = no_points - 1;
                no_knots = no_points - 1 + degree;
                cerr << "Not enough control points. A B-Spline of degree "
                     << degree << " will be constructed." << endl << endl;
            }

            int i;   // Used in for loops below.

            control_points = new GPoint2d[no_points];
            for (i = 0; i < no_points; i++)
                control_points[i] = GPoint2d(x[i], y[i]);

            knots = new float[no_knots];
            for (i = 0; i < no_knots; i++)
                knots[i] = knot_sequence[i];
        }
    }
}


GBSpline2d::GBSpline2d(int deg, int no_pts, const GPoint2d* pts)
{
    // Constructor for a B-Spline curve with control points given as points
    // and default knot sequence.

    degree = deg;
    no_points = no_pts;
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        if (degree > no_points-1)  {
            degree = no_points - 1;
            no_knots = no_points - 1 + degree;
            cerr << "Not enough control points. A B-Spline of degree "
                 << degree << " will be constructed." << endl << endl;
        }

        int i;   // Used in for loops below.

        control_points = new GPoint2d[no_points];
        for (i = 0; i < no_points; i++)
            control_points[i] = pts[i];

        knots = new float[no_knots];
        for (i = 0; i < degree; i++)
            knots[i] = 0;
        for (i = degree; i < no_points-1; i++)
            knots[i] = i - degree + 1;
        for (i = no_points-1; i < no_knots; i++)
            knots[i] = no_points - degree;
    }
}


GBSpline2d::GBSpline2d(int deg, int no_pts, const GPoint2d* pts,
                       const float* knot_sequence)
{
    // Constructor for a B-Spline curve with control points given as points
    // and given knot sequence.

    degree = deg;
    no_points = no_pts;
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knots, knot_sequence))  {
            cerr << "Knot sequence must be non-decreasing. "
                 << "Construction of B-Spline failed." << endl << endl;
        }

        else  {
            if (degree > no_points-1)  {
                degree = no_points - 1;
                no_knots = no_points - 1 + degree;
                cerr << "Not enough control points. A B-Spline of degree "
```

```
                    << degree << " will be constructed." << endl << endl;
            }

            int i;   // Used in for loops below.

            control_points = new GPoint2d[no_points];
            for (i = 0; i < no_points; i++)
                control_points[i] = pts[i];

            knots = new float[no_knots];
            for (i = 0; i < no_knots; i++)
                knots[i] = knot_sequence[i];
        }
    }
}


GBSpline2d::GBSpline2d(int deg, const GPolyPoint2d& polypoint)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint and default knot sequence.

    degree = deg;
    no_points = polypoint.no_pts();
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        if (degree > no_points-1)  {
            degree = no_points - 1;
            no_knots = no_points - 1 + degree;
            cerr << "Not enough control points. A B-Spline of degree "
                 << degree << " will be constructed." << endl << endl;
        }

        int i;   // Used in for loops below.

        control_points = new GPoint2d[no_points];
        polypoint.points(control_points);

        knots = new float[no_knots];
        for (i = 0; i < degree; i++)
            knots[i] = 0;
        for (i = degree; i < no_points-1; i++)
            knots[i] = i - degree + 1;
        for (i = no_points-1; i < no_knots; i++)
            knots[i] = no_points - degree;
    }
}


GBSpline2d::GBSpline2d(int deg, const GPolyPoint2d& polypoint,
                       const float* knot_sequence)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint and given knot sequence.

    degree = deg;
    no_points = polypoint.no_pts();
    no_knots = no_points - 1 + degree;

    if ((no_points > 0)  &&  (degree > 0))  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knots, knot_sequence))  {
            cerr << "Knot sequence must be non-decreasing. "
                 << "Construction of B-Spline failed." << endl << endl;
        }

        else  {
            if (degree > no_points-1)  {
                degree = no_points - 1;
                no_knots = no_points - 1 + degree;
                cerr << "Not enough control points. A B-Spline of degree "
                     << degree << " will be constructed." << endl << endl;
            }

            int i;   // Used in for loops below.

            control_points = new GPoint2d[no_points];
            polypoint.points(control_points);

            knots = new float[no_knots];
            for (i = 0; i < no_knots; i++)
                knots[i] = knot_sequence[i];
        }
    }
}


GBSpline2d::GBSpline2d(const GBSpline2d& bspline)
{
    // Copy constructor for B-Spline curves.
    // cf. Stroustrup, section 7.6

    degree = bspline.degree;
    no_points = bspline.no_points;
    no_knots = bspline.no_knots;

    if ((no_points > 0)  &&  (degree > 0))  {
        control_points = new GPoint2d[no_points];
        bspline.control_pts(control_points);

        knots = new float[no_knots];
        bspline.knot_sequence(knots);
    }
}
```

```
GBSpline2d::~GBSpline2d()
{
    // Destructor function for GBSpline2d.

    if (no_points > 0)
        delete[] control_points;
    if (no_knots > 0)
        delete[] knots;
}


GBSpline2d& GBSpline2d::operator=(const GBSpline2d& bspline)
{
    // Assignement operator for GBSpline2d.
    // cf. Stroustrup, section 7.6

    if (this != &bspline)  {
        if (no_points > 0)
            delete[] control_points;
        if (no_knots > 0)
            delete[] knots;

        degree = bspline.degree;
        no_points = bspline.no_points;
        no_knots = bspline.no_knots;

        if ((no_points > 0)  &&  (degree > 0))  {
            control_points = new GPoint2d[no_points];
            bspline.control_pts(control_points);

            knots = new float[no_knots];
            bspline.knot_sequence(knots);
        }
    }
    return *this;
}


int GBSpline2d::deg() const
{
    // Returns the degree of the B-Spline curve.

    return degree;
}


int GBSpline2d::no_pts() const
{
    // Returns the number of control points of the B-Spline curve.

    return no_points;
}
```

```
void GBSpline2d::control_pts(GPoint2d* points) const
{
    // points is a pointer to the array of control points of the curve.

    for (int i = 0; i < no_points; i++)
        points[i] = control_points[i];
}


int GBSpline2d::no_knts() const
{
    // Returns the number of knots in the B-Spline curve.

    return no_knots;
}


void GBSpline2d::knot_sequence(float* knot_seq) const
{
    // knot_seq is a pointer to the array of knots of the B-Spline curve.

    for (int i = 0; i < no_knots; i++)
        knot_seq[i] = knots[i];
}


void GBSpline2d::insert_knot(int index, const float knot_value)
{
    // Insertion of knot_value in the interval with lower bound index
    // 'index'. We use Boehm's knot insertion algorithm.

    // Check the insertion is in allowed interval. Allowed intervals start at
    // t[i], with i = degree-1, ..., no_points-2.
    if ((index < degree-1)  ||  (index > no_points-2))
        cerr << "Insertion in this interval not allowed. "
             << "Knot insertion aborted." << endl << endl;

    // Check for allowed knot value.
    else if ((knot_value < knots[index])  ||  (knot_value > knots[index+1]))
        cerr << "New knot value must be between knot values at interval ends."
             << endl << "Knot insertion aborted." << endl << endl;

    else  {
        no_points += 1;
        GPoint2d* new_points = new GPoint2d[no_points];

        int j;   // Used in for loops below.

        for (j = 0; j < index-degree+2; j++)
            new_points[j] = control_points[j];

        for (j = index-degree+2; j < index+2; j++)  {
            const float coeff = (knots[j+degree-1] - knot_value) /
                                (knots[j+degree-1] - knots[j-1]);
```

```
            new_points[j] = coeff*control_points[j-1] +
                                                    (1-coeff)*control_points[j];
        }

        for (j = index+2; j < no_points; j++)
            new_points[j] = control_points[j-1];

        delete[] control_points;   // Safe because of the checks above.
        control_points = new_points;

        // Tidy up the knot sequence.
        no_knots += 1;
        float* new_knots = new float[no_knots];

        for (j = 0; j <= index; j++)
            new_knots[j] = knots[j];
        new_knots[index+1] = knot_value;
        for (j = index+2; j < no_knots; j++)
            new_knots[j] = knots[j-1];

        delete[] knots;   // Safe because of the checks above.
        knots = new_knots;
    }
}


void GBSpline2d::insert_many_knots(int index,
                                   int no_knts, const float* knot_values)
{
    // Insertion of multiple knots in the same interval (Oslo algorithm).
    // Only applicable for insertion of (degree) or (degree+1) knots.

    int error = 0;   // Set to 1 if an error occurs.

    // Check for applicability.
    if ((no_knts > degree+1)  ||  (no_knts < degree))  {
        cerr << "Only " << degree << " or " << degree+1 << " knots can be "
             << "inserted simultaneously." << endl
             << "Inserting them sequentially instead." << endl << endl;
        error = 1;
        for (int i = 0; i < no_knts; i++)
            insert_knot(index+i, knot_values[i]);
    }

    // Check the insertion is in allowed interval. Allowed intervals start at
    // t[i], with i = degree-1, ..., no_points-2.
    if (!error)
        if ((index < degree-1)  ||  (index > no_points-2))  {
            cerr << "Insertion in this interval not allowed. "
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }

    // Check for allowed knot values.

    if (!error)
        for (int i = 0; (i < no_knts  &&  !error); i++)
            if ((knot_values[i] < knots[index])  ||
                (knot_values[i] > knots[index+1]))  {
                cerr << "New knot values must be between knot values "
                     << "at interval ends." << endl
                     << "Knot insertion aborted." << endl << endl;
                error = 1;
            }
    if (!error)  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knts, knot_values))  {
            cerr << "New knot values must be non-decreasing." << endl
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }
    }

    if (!error)  {   // Proceed.
        no_points += no_knts;   // New number of points.
        GPoint2d* new_points = new GPoint2d[no_points];

        int j, k;   // Used in for loops below.

        for (j = 0; j < index-degree+2; j++)
            new_points[j] = control_points[j];

        GPoint2d pts[degree+1];
        // Points used in the de Casteljau triangles.

        // Calculate first (degree) new points, in increasing order.
        for (j = 0; j <= degree; j++)
            pts[j] = control_points[j+index-degree+1];

        for (j = 0; j < degree; j++)
            for (k = degree; k >= j+1; k--)  {
                const float coeff= (knots[k-j+index] - knot_values[j]) /
                                   (knots[k-j+index] - knots[k-j+index-degree]);
                pts[k] = coeff*pts[k-1] + (1-coeff)*pts[k];
            }

        // Insert them in new_points.
        for (j = index-degree+2; j < index+2; j++)
            new_points[j] = pts[j-index+degree-1];

        // Calculate remaining new points, in decreasing order.
        for (j = 0; j <= degree; j++)
            pts[j] = control_points[j+index-degree+1];

        // s is 0 or 1 according to no_knts.
        const int s = 1-no_knts+degree;
        for (j = 0; j < degree-s; j++)
            for (k = s; k <= degree-j-1; k++)  {
```

```
                const float coeff= (knots[k+j+index+1] -
                                                knot_values[no_knts-1-j]) /
                                    (knots[k+j+index+1] -
                                                knots[k+j+index+1-degree]);
                pts[k] = coeff*pts[k] + (1-coeff)*pts[k+1];
            }

        // Insert them in new_points.
        for (j = index+2; j < index+no_knts+1; j++)
            new_points[j] = pts[j-index-2+s];

        // Insert remaining control points in new_points.
        for (j = index+no_knts+1; j < no_points; j++)
            new_points[j] = control_points[j-no_knts];

        delete[] control_points;
        control_points = new_points;

        // Tidy up the knot sequence.
        no_knots += no_knts;
        float* new_knots = new float[no_knots];

        for (j = 0; j <= index; j++)
            new_knots[j] = knots[j];
        for (j = index+1; j < index+1+no_knts; j++)
            new_knots[j] = knot_values[j-index-1];
        for (j = index+no_knts+1; j < no_knots; j++)
            new_knots[j] = knots[j-no_knts];

        delete[] knots;
        knots = new_knots;
    }
}


void GBSpline2d::delete_knot(int index)
{
    // Deletion of the knot indexed 'index'.
    // Only knots with index degree, ..., no_points-2 may be deleted.

    // Check for allowed knot to be deleted.
    if ((index < degree) || (index > no_points-1))
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    // Check if it is the first of three equal knots.
    else if ((knots[index] == knots[index+1])  &&
             (knots[index] == knots[index+2]))
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    else  {
        no_points -= 1;
        GPoint2d* new_points = new GPoint2d[no_points];

        int j;   // Used in for loops below.

        for (j = no_points-1; j > index-1; j--)
            new_points[j] = control_points[j+1];

        for (j = index-1; j > index-degree; j--)  {
            const float coeff = (knots[j+degree+1] - knots[j]) /
                                (knots[j+degree+1] - knots[index]);
            new_points[j] = coeff*control_points[j+1]+(1-coeff)*new_points[j+1];
        }

        for (j = index-degree; j >= 0; j--)
            new_points[j] = control_points[j];

        delete[] control_points;
        control_points = new_points;

        // Tidy up the knot sequence.
        no_knots -= 1;
        float* new_knots = new float[no_knots];

        for (j = 0; j < index; j++)
            new_knots[j] = knots[j];
        for (j = index; j < no_knots; j++)
            new_knots[j] = knots[j+1];

        delete[] knots;
        knots = new_knots;
    }
}


void GBSpline2d::transform(const Trans2d& trans)
{
    // Transformation of the B-Spline curve. As for Bezier curves,
    // we only need to transform the control polygon.

    for (int i = 0; i < no_points; i++)
        control_points[i].transform(trans);
}


void GBSpline2d::interpolate(const int no_pts, const float* x, const float* y,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the no_pts points given by coordinates with a cubic B-Spline
    // The knots are taken uniformly spaced.
    // r1 and r2 are the two arbitrary points needed.

    int i;   // Used in for loops below.

    if (no_pts > 1)  {
        // Elements of the interpolating B-Spline.
```

```
        const int no_pts_bspl = no_pts + 2;
        const int deg = 3;
        const int no_knots = no_pts_bspl - 1 + deg;

        // Produce uniform knot sequence.
        float* knots = new float[no_knots];
        for (i = 0; i < deg; i++)
            knots[i] = 0;
        for (i = deg; i < no_pts_bspl-1; i++)
            knots[i] = i - deg + 1;
        for (i = no_pts_bspl-1; i < no_knots; i++)
            knots[i] = no_pts_bspl - deg;

        interpolate(no_pts, x, y, knots, r1, r2);

        delete[] knots;
    }
    else
        *this = GBSpline2d();   // An empty one.
}


void GBSpline2d::interpolate(const int no_pts, const float* x, const float* y,
                             const float* knots,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the no_pts points given by coordinates with a cubic B-Spline
    // The knots are provided.
    // r1 and r2 are the two arbitrary points needed.

    if (no_pts > 1)  {
        // Elements of the interpolating B-Spline.
        const int no_pts_bspl = no_pts + 2;
        const int deg = 3;

        int i;   // Used in for loops below.

        // Set up the matrix of the system.
        float* alpha = new float[no_pts_bspl];
        float* beta = new float[no_pts_bspl];
        float* gamma = new float[no_pts_bspl];

        alpha[1] = 0;
        beta[0] = 1;
        beta[1] = 1;
        gamma[0] = 0;
        gamma[1] = 0;

        for (i = 1; i < no_pts-1; i++)  {
            const float temp1 = (knots[i+3]-knots[i+2]) / (knots[i+3]-knots[i]);
            const float temp2 = (knots[i+2]-knots[i+1]) /
                                                    (knots[i+4]-knots[i+1]);

            alpha[i+1] = temp1*(knots[i+3]-knots[i+2]);
            beta[i+1] = temp1*(knots[i+2]-knots[i]) +
                                                    temp2*(knots[i+4]-knots[i+2]);
            gamma[i+1] = temp2*(knots[i+2]-knots[i+1]);
        }

        alpha[no_pts_bspl - 2] = 0;
        alpha[no_pts_bspl - 1] = 0;
        beta[no_pts_bspl - 2] = 1;
        beta[no_pts_bspl - 1] = 1;
        gamma[no_pts_bspl - 2] = 0;

        // Analyse the matrix of the system into L and U once.
        float* low = new float[no_pts_bspl];
        float* up = new float[no_pts_bspl];
        l_u_analysis(no_pts_bspl, alpha, beta, gamma, low, up);

        float* rhs = new float[no_pts_bspl];   // Right-hand side of the systems
        float* solution_x = new float[no_pts_bspl];   // Solutions of the
        float* solution_y = new float[no_pts_bspl];   // systems.

        // Form right-hand side of system for x.
        rhs[0] = x[0];
        rhs[1] = r1.x();

        for (i = 1; i < no_pts-1; i++)
            rhs[i+1] = x[i] * (knots[i+3] - knots[i+1]);

        rhs[no_pts_bspl - 2] = r2.x();
        rhs[no_pts_bspl - 1] = x[no_pts - 1];

        solve_system(no_pts_bspl, low, up, gamma, rhs, solution_x);

        // Form right-hand side of system for y.
        rhs[0] = y[0];
        rhs[1] = r1.y();

        for (i = 1; i < no_pts-1; i++)
            rhs[i+1] = y[i] * (knots[i+3] - knots[i+1]);

        rhs[no_pts_bspl - 2] = r2.y();
        rhs[no_pts_bspl - 1] = y[no_pts - 1];

        solve_system(no_pts_bspl, low, up, gamma, rhs, solution_y);

        *this = GBSpline2d(deg, no_pts_bspl, solution_x, solution_y, knots);

        delete[] alpha;
        delete[] beta;
        delete[] gamma;
        delete[] rhs;
        delete[] solution_x;
        delete[] solution_y;
}
```

```
    else
        *this = GBSpline2d();   // An empty one.
}


void GBSpline2d::interpolate(const int no_pts, const GPoint2d* pts,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the no_pts points in pts with a cubic B-Spline.
    // The knots are uniformly spaced.
    // r1 and r2 are the two arbitrary points needed.

    if (no_pts > 1)  {
        float* x = new float[no_pts];
        float* y = new float[no_pts];

        for (int i = 0; i < no_pts; i++)  {
            x[i] = pts[i].x();
            y[i] = pts[i].y();
        }

        interpolate(no_pts, x, y, r1, r2);

        delete[] x;
        delete[] y;
    }
    else
       *this = GBSpline2d();   // An empty one.
}


void GBSpline2d::interpolate(const int no_pts, const GPoint2d* pts,
                             const float* knots,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the no_pts points in pts with a cubic B-Spline.
    // The knots are provided.
    // r1 and r2 are the two arbitrary points needed.

    if (no_pts > 1)  {
        float* x = new float[no_pts];
        float* y = new float[no_pts];

        for (int i = 0; i < no_pts; i++)  {
            x[i] = pts[i].x();
            y[i] = pts[i].y();
        }

        interpolate(no_pts, x, y, knots, r1, r2);

        delete[] x;
        delete[] y;
    }
    else
        *this = GBSpline2d();   // An empty one.
}


void GBSpline2d::interpolate(const GPolyPoint2d& polypoint,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the points in polypoint with a cubic B-Spline.
    // The knots are uniformly spaced.
    // r1 and r2 are the two arbitrary points needed.

    const int no_pts = polypoint.no_pts();

    if (no_pts > 1)  {
        GPoint2d* pts = new GPoint2d[no_pts];
        polypoint.points(pts);

        float* x = new float[no_pts];
        float* y = new float[no_pts];

        for (int i = 0; i < no_pts; i++)  {
            x[i] = pts[i].x();
            y[i] = pts[i].y();
        }

        interpolate(no_pts, x, y, r1, r2);

        delete[] pts;
        delete[] x;
        delete[] y;
    }
    else
        *this = GBSpline2d();   // An empty one.
}


void GBSpline2d::interpolate(const GPolyPoint2d& polypoint,
                             const float* knots,
                             const GPoint2d r1, const GPoint2d r2)
{
    // Interpolates the points in polypoint with a cubic B-Spline.
    // The knots are provided.
    // r1 and r2 are the two arbitrary points needed.

    const int no_pts = polypoint.no_pts();

    if (no_pts > 1)  {
        GPoint2d* pts = new GPoint2d[no_pts];
        polypoint.points(pts);

        float* x = new float[no_pts];
        float* y = new float[no_pts];

        for (int i = 0; i < no_pts; i++)  {
```

```
            x[i] = pts[i].x();
            y[i] = pts[i].y();
        }

        interpolate(no_pts, x, y, knots, r1, r2);

        delete[] pts;
        delete[] x;
        delete[] y;
    }
    else
        *this = GBSpline2d();   // An empty one.
}


int valid_knots(int no_knots, const float* knots)
{
    // Checks if the sequence of knots is non-decreasing to be valid.

    for (int i = 0; i < no_knots-1; i++)
        if (knots[i] > knots[i+1])
            return 0;
    return 1;
}
```

```
/* File Curve2d.cc
 *
 * Implements header file Curve2d.h containing classes Bezier2d and
 * BSpline2d, for the representation of (the drawable primitives of) Bezier
 * and B-Spline curves in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


#include "GPrimitive2d.h"
#include "graphics.h"
#include "Attributes.h"
#include "View2d.h"
#include "Object2d.h"
#include "Primitive2d.h"
#include "Trans2d.h"
#include "GCurve2d.h"
#include "Curve2d.h"
#include "operator.h"
#include <math.h>


Bezier2d::Bezier2d()
{
    // Constructor for a null Bezier curve.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const Attributes& attributes)
    : Primitive2d(attributes)
{
    // Constructor for a null Bezier curve with only attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const GBezier2d& bezier)
    : GBezier2d(bezier)
{
    // Constructor for a Bezier curve from a GBezier2d
    // with no attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const GBezier2d& bezier, const Attributes& attributes)
    : Primitive2d(attributes), GBezier2d(bezier)
{
    // Constructor for a Bezier curve from a GBezier2d
    // with attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(int no_pts, const float* x, const float* y)
    : GBezier2d(no_pts, x, y)
{
    // Constructor with vertices of control polygon specified by coordinates,
    // and no attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(int no_pts, const float* x, const float* y,
                   const Attributes& attributes)
    : Primitive2d(attributes), GBezier2d(no_pts, x, y)
{
    // Constructor with vertices of control polygon specified by coordinates,
    // and attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(int no_pts, const GPoint2d* pts)
    : GBezier2d(no_pts, pts)
{
    // Constructor for vertices of control polygon given as points,
    // and no attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(int no_pts, const GPoint2d* pts,
```

```
                    const Attributes& attributes)
    : Primitive2d(attributes), GBezier2d(no_pts, pts)
{
    // Constructor for vertices of control polygon given as points,
    // and attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const GPolyPoint2d& polypoint)
    : GBezier2d(polypoint)
{
    // Constructor for control polygon specified as polypoint,
    // and no attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const GPolyPoint2d& polypoint, const Attributes& attributes)
    : Primitive2d(attributes), GBezier2d(polypoint)
{
    // Constructor for control polygon specified as polypoint,
    // and attributes specified.

    method = DIFFERENCES;
    tolerance = 0.01;
    eval_step = 0.01;
}


Bezier2d::Bezier2d(const Bezier2d& bezier)
{
    // Copy contstructor. See comments in Primitive2d.cc.

    *this = bezier;
}


void Bezier2d::set_drawing_method(const Bezier_algorithm alg)
{
    // Set the method used for drawing of the Bezier curve.

    method = alg;
}


void Bezier2d::set_tolerance(const float tol)
{
    // Set the tolerance for flatness tests in the de Casteljau algorithm.

    tolerance = tol;
}


void Bezier2d::set_eval_step(const float h)
{
    // Set the evaluation step h for the differences method.

    eval_step = h;
}


void Bezier2d::transform(const Trans2d& trans)
{
    // Transformation function for Bezier curves.

    GBezier2d::transform(trans);
}


void Bezier2d::draw(View2d& view) const
{
    // Drawing of the Bezier curve.

    const int no_points = no_pts();

    if (no_points > 0)  {

        if (no_points == 1)  {   // A single point.
            GPoint2d* points = new GPoint2d[no_points];
            control_pts(points);
            Point2d pt(points[0], *this);   // *this for the attributes.
            pt.draw(view);
            delete[] points;
        }
        else if (no_points == 2)  {   // Draw a simple line.
            GPoint2d* points = new GPoint2d[no_points];
            control_pts(points);
            Line2d line(points[0], points[1], *this);
            line.draw(view);
            delete[] points;
        }
        else if (no_points >= 3)  {

            // Set drawing attributes.
            view.set_line_style(inquire_line_style());
            view.set_line_width(inquire_line_width());
            ColourType type;
            float red, green, blue, index;
            inquire_colour(type, red, green, blue, index);
            if (type == ComponentColour)
                view.set_colour(red, green, blue);
```

```
            else
                view.set_colour(index);

            if (method == DE_CASTELJAU)
                draw_decas(view);
            else if (method == DIFFERENCES)
                draw_diff(view);
            else
                cerr << "Unknown Bezier drawing method." << endl << endl;
        }
    }
}


void Bezier2d::draw_decas(View2d& view) const
{
    // Actually draws the Bezier curve, using the de Casteljau algorithm.

    const float t = 0.5;   // Parameter value for subdivision.

    if (is_flat(tolerance))  {
        const int no_points = no_pts();
        GPoint2d* control_points = new GPoint2d[no_points];
        control_pts(control_points);

        Line2d line(control_points[0], control_points[no_points - 1], *this);
        // *this for the attributes.
      line.draw(view);

      delete[] control_points;
    }
    else  {
        Bezier2d bez_left(*this),
                 bez_right(*this);
        subdivide(t, bez_left, bez_right);

        bez_left.draw_decas(view);
        bez_right.draw_decas(view);
    }
}


void Bezier2d::draw_diff(View2d& view) const
{
    // Actually draws the Bezier curve, using forward differences.

    // Functions to calculate differences of points and floats.
    void differences(const GPoint2d* const points, GPoint2d* const diff,
                     int no_points);
    void differences(const float* const array, float* const diff,
                     int no_elements);

    int i, j, r;   // Used in for loops below.

    const float h = eval_step;   // Just for abbreviation.

    // Keep the elements of the curve handy.
    const int degree = deg();
    const int no_points = no_pts();
    GPoint2d* control_points = new GPoint2d[no_points];
    control_pts(control_points);

    // Calculate the coefficients of the monomial form of the Bezier curve.

    GPoint2d* coeff = new GPoint2d[no_points];
    differences(control_points, coeff, no_points);
    float i_factorial = 1;
    float n_r = 1;
    // coeff[0] already holds control_points[0] as needed.
    for (i = 1; i < no_points; i++)  {
        i_factorial = i_factorial * i;
        n_r = n_r * (degree - i + 1);

        coeff[i] =  n_r / i_factorial * coeff[i];
    }

    // Calculate delta^r of f[0].

    GPoint2d* df0 = new GPoint2d[no_points];   // Array of differences.
    df0[0] = control_points[0];
    for (r = 1; r < no_points; r++)
        df0[r] = GPoint2d(0, 0);   // Initialisation.

    float* h_power = new float[no_points];   // All the powers of h.
    h_power[0] = 1;
    for (i = 1; i < no_points; i++)
        h_power[i] = h * h_power[i-1];

    float* d0 = new float[no_points];   // Array of delta^r of 0^i.
    for (i = 1; i < no_points; i++)  {
        for (j = 0; j <= i; j++)
            d0[j] = pow(j, i);
        differences(d0, d0, i+1);

        for (r = 1; r <= i; r++)
            df0[r] = df0[r] + d0[r] * h_power[i] * coeff[i];
    }

    // Find points of the Bezier curve.
    GPoint2d start, end;   // End points of the lines to be drawn.
    start = df0[0];
    const int no_steps = int((1.0 / h) + 0.5);   // Round up.
    for (i = 0; i < no_steps; i++)  {
        for (r = 0; r < no_points-1; r++)   // Calculate delta^r of f(t+h).
            df0[r] = df0[r] + df0[r+1];

        end = df0[0];
        Line2d line(start, end, *this);   // *this for the attributes.
```

```
        line.draw(view);

        start = end;
    }

    delete[] control_points;
    delete[] coeff;
    delete[] df0;
    delete[] h_power;
    delete[] d0;
}


void differences(const GPoint2d* const points, GPoint2d* const diff,
                 int no_points)
{
    // Calculates all forward differences
    // delta^r of points[0], r = 1,2,...,no_points-1.

    for (int j = 0; j < no_points; j++)
        diff[j] = points[j];

    for (int r = 1; r < no_points; r++)
        for (int i = no_points-1; i > r-1; i--)
            diff[i] = diff[i] - diff[i-1];
}


void differences(const float* const array, float* const diff,
                 int no_elements)
{
    // Calculates all forward differences
    // delta^r of array[0], r = 1,2,...,no_elements-1.

    for (int j = 0; j < no_elements; j++)
        diff[j] = array[j];

    for (int r = 1; r < no_elements; r++)
        for (int i = no_elements-1; i > r-1; i--)
            diff[i] = diff[i] - diff[i-1];
}


void Bezier2d::draw_transformed(View2d& view, const Trans2d& trans) const
{
    // Draws the Bezier curve transformed, without affecting it.

    Bezier2d bezier = *this;
    bezier.transform(trans);
    bezier.draw(view);
}


/*****************************************************************************/
```

```
BSpline2d::BSpline2d()
{
    // Constructor for a null B-Spline curve.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(const Attributes& attributes)
    : Primitive2d(attributes)
{
    // Constructor for a null B-Spline curve with only attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(const GBSpline2d& bspline)
    : GBSpline2d(bspline)
{
    // Constructor for a B-Spline curve given as a GBSpline2d,
    // with no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(const GBSpline2d& bspline, const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(bspline)
{
    // Constructor for a B-Spline curve given as a GBSpline2d,
    // and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}
```

```
BSpline2d::BSpline2d(int deg, int no_pts, const float* x, const float* y)
    : GBSpline2d(deg, no_pts, x, y)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates, default knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const float* x, const float* y,
                     const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, no_pts, x, y)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates, default knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const float* x, const float* y,
                     const float* knot_sequence)
    : GBSpline2d(deg, no_pts, x, y, knot_sequence)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates, given knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const float* x, const float* y,
                     const float* knot_sequence, const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, no_pts, x, y, knot_sequence)
{
    // Constructor for a B-Spline curve with control points specified by
    // coordinates, given knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const GPoint2d* pts)
    : GBSpline2d(deg, no_pts, pts)
{
    // Constructor for a B-Spline curve with control points given as points,
    // default knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                     const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, no_pts, pts)
{
    // Constructor for a B-Spline curve with control points given as points,
    // default knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                     const float* knot_sequence)
    : GBSpline2d(deg, no_pts, pts, knot_sequence)
{
    // Constructor for a B-Spline curve with control points given as points,
    // given knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, int no_pts, const GPoint2d* pts,
                     const float* knot_sequence, const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, no_pts, pts, knot_sequence)
{
```

```
    // Constructor for a B-Spline curve with control points given as points,
    // given knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, const GPolyPoint2d& polypoint)
    : GBSpline2d(deg, polypoint)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint, default knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, const GPolyPoint2d& polypoint,
                     const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, polypoint)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint, default knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, const GPolyPoint2d& polypoint,
                     const float* knot_sequence)
    : GBSpline2d(deg, polypoint, knot_sequence)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint, given knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(int deg, const GPolyPoint2d& polypoint,
                          const float* knot_sequence, const Attributes& attributes)
    : Primitive2d(attributes), GBSpline2d(deg, polypoint, knot_sequence)
{
    // Constructor for a B-Spline curve with control points given as a
    // polypoint, given knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.01;
    diff_eval_step = 0.01;
    eval_step = 0.01;
}


BSpline2d::BSpline2d(const BSpline2d& bspline)
{
    // Copy constructor for B-Spline curves.
    // See comments in Primitive2d.cc

    *this = bspline;
}


void BSpline2d::set_drawing_method(BSpline_algorithm alg)
{
    // Sets the drawing method for the B-Spline curve.

    method = alg;
}


void BSpline2d::set_Bezier_drawing_method(Bezier_algorithm alg)
{
    // Sets the drawing method for the Bezier curve segments.

    Bezier_method = alg;
}


void BSpline2d::set_decas_tolerance(const float tol)
{
    // Sets the tolerance for the de Castlejau algorithm when applied to
    // the Bezier segments.

    decas_tolerance = tol;
}


void BSpline2d::set_diff_eval_step(const float h)
{
    // Sets the evaluation step for the differences method when applied to
    // the Bezier curve segments.
```

```
    diff_eval_step = h;
}


void BSpline2d::set_eval_step(const float h)
{
    // Sets the evaluation step for the de Boor algorithm.

    eval_step = h;
}


void BSpline2d::transform(const Trans2d& trans)
{
    // Transformation function for B-Spline curves.

    GBSpline2d::transform(trans);
}


void BSpline2d::draw(View2d& view) const
{
    // Drawing of the B-Spline curve.

    if ((no_pts() > 0)  &&  (deg() > 0))  {

        // Checks for knots of multiplicity mult, from knots[j1] to knots[j2].
        int multiple_knots(const BSpline2d& bspline, int j1, int j2, int mult);

        // Set drawing attributes.
        view.set_line_style(inquire_line_style());
        view.set_line_width(inquire_line_width());
        ColourType type;
        float red, green, blue, index;
        inquire_colour(type, red, green, blue, index);
        if (type == ComponentColour)
            view.set_colour(red, green, blue);
        else
            view.set_colour(index);

        if (multiple_knots(*this, 0, no_knts()-1, deg()+1))
            cerr << "B-Splines having knots with multiplicity higher than"
             << endl << "their degree cannot be rendered. Procedure aborted."
             << endl << endl;

        else if (method == AS_BEZIER)
            if (deg() > 3)  {
                cerr << "Rendering of B-Splines as piecewise Bezier curves"
                 << endl << "only applicable for quadratic or cubic B-Splines."
                 << endl << "Using the de Boor method instead." << endl << endl;

                draw_deboor(view);
            }
            else if (multiple_knots(*this, 1, no_knts()-2, deg()))  {
                cerr << "Rendering of B-Splines as piecewise Bezier curves"
                 << endl << "not applicable to curves having knots with"
                 << endl << "multiplicity equal to their degree, other than the"
                 << endl << "end knots. Using the de Boor method instead."
                 << endl << endl;

                draw_deboor(view);
            }
            else
                draw_bez(view);

        else if (method == DE_BOOR)
            draw_deboor(view);
        else
            cerr << "Unknown B-Spline drawing method." << endl << endl;
    }
}


int multiple_knots(const BSpline2d& bspline, int j1, int j2, int mult)
{
    // Checks for knots of multiplicity mult, from knots[j1] to knots[j2].

    float* knots = new float[bspline.no_knts()];
    bspline.knot_sequence(knots);

    int i = j1;
    while (i <= j2-mult+1)  {
        const float model = knots[i];
        int count = 1;
        i++;
        while ((i <= j2)  &&  (knots[i] == model)  &&  (count < mult))  {
          count++;
          i++;
        }
        if (count == mult)  {
            delete[] knots;
            return 1;
        }
    }

    delete[] knots;
    return 0;
}


void BSpline2d::draw_bez(View2d& view) const
{
    // Draws quadratic or cubic B-Splines by calculating the corresponding
    // control points for the Bezier curve segments.

    // Get the elements of the curve handy.
    const int degree = deg();
```

```
    const int no_points = no_pts();
    GPoint2d* points = new GPoint2d[no_points];
    control_pts(points);
    const int no_knots = no_knts();
    float* knots = new float[no_knots];
    knot_sequence(knots);

    float coeff;   // Coefficient for interpolations below.

    if (degree == 2)   // Quadratic case.
        for (int i = degree-1; i < no_points-1; i++)  {
            GPoint2d bezier_points[3];
            coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-1]);
            bezier_points[0] = coeff*points[i-1] + (1-coeff)*points[i];

            bezier_points[1] = points[i];

            coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i]);
            bezier_points[2] = coeff*points[i] + (1-coeff)*points[i+1];

            Bezier2d bezier(3, bezier_points, *this);   // *this for attributes
            bezier.set_drawing_method(Bezier_method);
            bezier.set_tolerance(decas_tolerance);
            bezier.set_eval_step(diff_eval_step);
            bezier.draw(view);
        }

    else   // Cubic case.
        for (int i = degree-1; i < no_points-1; i++)  {
            GPoint2d bezier_points[4];

            // First step.
            coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-2]);
            bezier_points[0] = coeff*points[i-2] + (1-coeff)*points[i-1];

            coeff = (knots[i+2] - knots[i]) / (knots[i+2] - knots[i-1]);
            bezier_points[1] = coeff*points[i-1] + (1-coeff)*points[i];

            coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i-1]);
            bezier_points[2] = coeff*points[i-1] + (1-coeff)*points[i];

            coeff = (knots[i+3] - knots[i+1]) / (knots[i+3] - knots[i]);
            bezier_points[3] = coeff*points[i] + (1-coeff)*points[i+1];

            // Second step.
            coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-1]);
            bezier_points[0] = coeff*bezier_points[0] +
                                                (1-coeff)*bezier_points[1];

            coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i]);
            bezier_points[3] = coeff*bezier_points[2] +
                                                (1-coeff)*bezier_points[3];

            Bezier2d bezier(4, bezier_points, *this);   // *this for attributes
            bezier.set_drawing_method(Bezier_method);
            bezier.set_tolerance(decas_tolerance);
            bezier.set_eval_step(diff_eval_step);
            bezier.draw(view);
        }

    delete[] points;
    delete[] knots;
}


void BSpline2d::draw_deboor(View2d& view) const
{
    // Draws a B-Spline curve of any degree using the de Boor formula.

    // This function calculates the points on the B-Spline.
    // 'index' is the index of the lower end of the interval.
    GPoint2d deBoor_point(const GBSpline2d& bspline, float t, int index);

    // Get the elements of the curve.
    const int degree = deg();
    const int no_points = no_pts();
    const int no_knots = no_knts();
    float* knots = new float[no_knots];
    knot_sequence(knots);

    float h = eval_step;   // Just for abbreviation.

    // The curve is only defined in the intervals starting at t[i],
    // with i = degree-1, ..., no_points-2.

    // Transform h to match the actual range of t.
    h = h * (knots[no_points-1] - knots[degree-1]);

    GPoint2d start, end;   // End points of the lines drawn.
    start = deBoor_point(*this, knots[degree-1], degree-1);
    for (int i = degree-1; i < no_points-1; i++)  {
        for (float t = knots[i]; t < knots[i+1]; t += h)  {
            end = deBoor_point(*this, t, i);
            Line2d line(start, end, *this);   // *this for the Attributes.
            line.draw(view);
            start = end;
        }
    }

    // Cater for the last point.
    end = deBoor_point(*this, knots[no_points-1], no_points-2);
    Line2d line(start, end, *this);
    line.draw(view);

    delete[] knots;
}
```

```
GPoint2d deBoor_point(const GBSpline2d& bspline, float t, int index)
{
    // Calculates the point corresponding to parameter value t,
    // t in [ knots[index], knots[index+1] ],
    // on the B-Spline curve, using the de Boor algorithm.

    // This function is declared 'friend' of class GBSpline2d, so all the
    // elements of the curve are accessible.

    const int deg = bspline.degree;

    GPoint2d* pts = new GPoint2d[deg + 1];   // de Boor points.
    for (int j = 0; j < deg + 1; j++)
        pts[j] = bspline.control_points[index - j + 1];

    for (int s = 0; s < deg; s++)
        for (int i = deg; i > s; i--)  {
            const float coeff = (bspline.knots[deg+1+index-i] - t) /
                                (bspline.knots[deg+1+index-i] -
                                                    bspline.knots[s+1+index-i]);
            pts[i] = coeff*pts[i] + (1-coeff)*pts[i-1];
        }

    GPoint2d pt = pts[deg];
    delete[] pts;
    return pt;
}


void BSpline2d::draw_transformed(View2d& view, const Trans2d& trans) const
{
    // Draws the B-Spline curve transformed, without affecting it.

    BSpline2d bspline(*this);
    bspline.transform(trans);
    bspline.draw(view);
}
```

```
/* File GSurface3d.cc
 *
 * Implements header file GCurve3d.h containing classes GBezier3d and
 * GBSpline3d, for the representation of (the geometric primitives of) Bezier
 * and B-Spline surfaces in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


#include "graphics2d.h"
#include "Atom3d.h"
#include "GObject3d.h"
#include "Vector3d.h"
#include "GPrimitive3d.h"
#include "Box3d.h"
#include "GPlane.h"
#include "Trans3d.h"
#include "Camera.h"
#include "View3d.h"
#include "Drawable3d.h"
#include "Primitive3d.h"
#include "Object3d.h"
#include "SimpleObject3d.h"
#include "DPrimitive3d.h"
#include "light.h"
#include "model.h"
#include "Pixel.h"
#include "Colour.h"
#include "BRep.h"
#include "tridiagonal.h"
#include "GSurface3d.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>


GPoint3dArray::GPoint3dArray()
{
    // Constructor for an empty GPoint3dArray.

    n_rows = 0;
    n_columns = 0;
    points = 0;   // Null pointer.
}


GPoint3dArray::GPoint3dArray(const int nr, const int nc)
{
    // Constructor for GPoint3dArray.

    n_rows = nr;
    n_columns = nc;

    if ((n_rows > 0)  &&  (n_columns > 0))  {
        points = new GPoint3d* [n_rows];
        for (int i = 0; i < n_rows; i++)
            points[i] = new GPoint3d[n_columns];
    }
}


GPoint3dArray::GPoint3dArray(const GPoint3dArray& array)
{
    // Copy constructor for GPoint3dArray.

    n_rows = array.n_rows;
    n_columns = array.n_columns;

    if ((n_rows > 0)  &&  (n_columns > 0))  {
        points = new GPoint3d* [n_rows];
        for (int i = 0; i < n_rows; i++)  {
            points[i] = new GPoint3d[n_columns];
            for (int j = 0; j < n_columns; j++)
                points[i][j] = array.points[i][j];
        }
    }
}


GPoint3dArray::~GPoint3dArray()
{
    // Destructor for GPoint3dArray.

    if (n_columns > 0)
        for (int i = 0; i < n_rows; i++)
            delete[] points[i];
    if (n_rows > 0)
        delete[] points;
}


GPoint3dArray& GPoint3dArray::operator=(const GPoint3dArray& array)
{
    // Assignement operator for GPoint3dArray.

    if (this != &array)  {
        if (n_columns > 0)
            for (int i = 0; i < n_rows; i++)
                delete[] points[i];
        if (n_rows > 0)
            delete[] points;

        n_rows = array.n_rows;
        n_columns = array.n_columns;
        if ((n_rows > 0)  &&  (n_columns > 0))  {
```

```
            points = new GPoint3d* [n_rows];
            for (int i = 0; i < n_rows; i++)  {
                points[i] = new GPoint3d[n_columns];
                for (int j = 0; j < n_columns; j++)
                    points[i][j] = array.points[i][j];
            }
        }
    }
    return *this;
}


void GPoint3dArray::convert_to_BRep(BRep& brep) const
{
    // A function to convert a GPoint3dArray to the corresponding BRep.

    if ((n_rows > 0)  &&  (n_columns > 0))  {

        const int nr = n_rows,
                  nc = n_columns;

        const int no_vert = nr * nc;
        GPoint3d* vert = new GPoint3d[no_vert];
        for (int i = 0; i < nr; i++)
            for (int j = 0; j < nc; j++)  {
                int index = i*nc + j;   // Position in the list of vertices.
                get_cell(i, j, vert[index]);
            }

        const int no_polys = (nr-1) * (nc-1);
        GSurfacePolygon3d* polys = new GSurfacePolygon3d[no_polys];
        for (int k = 0; k < no_polys; k++)  {
            const int row = k / (nc-1),   // Position of top left corner of
                      col = k % (nc-1);   // the polygon in the array.
            const int pos = row*nc + col;   // Position of the top left corner
                                            // in the list of vertices.
            int face[4];
            face[0] = pos;
            face[1] = pos + 1;
            face[2] = pos + 1 + nc;
            face[3] = pos + nc;

            polys[k] = GSurfacePolygon3d(4, face);
        }

        GPolyPoint3d gpp(no_vert, vert);
        GSurfaceList gsl(no_polys, polys);

        brep.update(gpp, gsl);

        delete[] vert;
        delete[] polys;
    }
}


int GPoint3dArray::no_rows() const
{
    // Returns the number of rows of the array.

    return n_rows;
}


int GPoint3dArray::no_cols() const
{
    // Returns the number of columns of the array.

    return n_columns;
}


void GPoint3dArray::get_cell(int i, int j, GPoint3d& point) const
{
    // Returns cell (i, j).

    point = points[i][j];
}


void GPoint3dArray::set_cell(int i, int j, GPoint3d point)
{
    // Sets cell (i, j) equal to point.

    points[i][j] = point;
}


float GPoint3dArray::x(int i, int j) const
{
   // Returns x-coordinate of cell(i, j).

   return points[i][j].x();
}


float GPoint3dArray::y(int i, int j) const
{
   // Returns y-coordinate of cell(i, j).

   return points[i][j].y();
}


float GPoint3dArray::z(int i, int j) const
{
   // Returns z-coordinate of cell(i, j).
```

```
   return points[i][j].z();
}


void GPoint3dArray::get_row(int i, GPoint3d* row_points) const
{
    // Returns row i of the array into row_points.

    for (int j = 0; j < n_columns; j++)
        row_points[j] = points[i][j];
}


void GPoint3dArray::get_col(int j, GPoint3d* col_points) const
{
    // Returns column j of the array into col_points.

    for (int i = 0; i < n_rows; i++)
        col_points[i] = points[i][j];
}


void GPoint3dArray::set_row(int i, GPoint3d* row_points)
{
    // Sets points in row i of the array equal to row_points.

    for (int j = 0; j < n_columns; j++)
        points[i][j] = row_points[j];
}


void GPoint3dArray::set_col(int j, GPoint3d* col_points)
{
    // Sets points in column j of the array equal col_points.

    for (int i = 0; i < n_rows; i++)
        points[i][j] = col_points[i];
}


int GPoint3dArray::read(char* fname)
{
    // Reads a GPoint3dArray from file named fname.
    // Returns 0 in case of an error during input, 1 otherwise.
    // File format: First should come the number of rows and columns of the
    // array; then the coordinates of the points, stored by rows.

    ifstream input_file(fname);
    if (!input_file)  {
        cerr << "Error reading file " << fname << endl << endl;
        return 0;
    }

    int nr, nc;
    input_file >> nr >> nc;
    if (input_file.eof())  {
        cerr << "Error reading file " << fname << endl << endl;
        return 0;
    }

    GPoint3dArray array(nr, nc);
    for (int i = 0; i < nr; i++)
        for (int j = 0; j < nc; j++)  {
            float x, y, z;
            input_file >> x >> y >> z;

            if (input_file.eof())  {
                cerr << "Error reading file " << fname << endl << endl;
                return 0;
            }

            array.set_cell(i, j, GPoint3d(x, y, z));
        }

    *this = array;
    return 1;
}


/*****************************************************************************/


GBezier3d::GBezier3d()
{
    // Constructor for a null Bezier surface.

    no_points_t = 0;
    no_points_u = 0;
    control_points = GPoint3dArray();   // An empty one.
}


GBezier3d::GBezier3d(const GPoint3dArray& points)
{
    // Constructor for a Bezier surface.

    no_points_t = points.no_rows();
    no_points_u = points.no_cols();
    control_points = points;
}


GBezier3d::GBezier3d(const GBezier3d& bezier)
{
    // Copy contstructor.
    // cf. Stroustrup, section 7.6

    no_points_t = bezier.no_points_t;
```

```
    no_points_u = bezier.no_points_u;
    control_points = bezier.control_points;
}


GBezier3d& GBezier3d::operator=(const GBezier3d& bezier)
{
    // Assignement operator for GBezier3d.
    // cf. Stroustrup, section 7.6

    if (this != &bezier)  {
        no_points_t = bezier.no_points_t;
        no_points_u = bezier.no_points_u;
        control_points = bezier.control_points;
    }
    return *this;
}


int GBezier3d::deg_t() const
{
    // Returns the degree of the Bezier surface with respect to parameter t.
    // A surface of degree n requires n+1 control points.

    return (no_points_t - 1);
}


int GBezier3d::deg_u() const
{
    // Returns the degree of the Bezier surface with respect to parameter u.
    // A surface of degree n requires n+1 control points.

    return (no_points_u - 1);
}


int GBezier3d::no_pts_t() const
{
    // Returns the number of control points with respect to parameter t.

    return no_points_t;
}


int GBezier3d::no_pts_u() const
{
    // Returns the number of control points with respect to parameter u.

    return no_points_u;
}


void GBezier3d::control_pts(GPoint3dArray& points) const
{
    // points returns the array of control points of the surface.

    points = control_points;
}


Box3d GBezier3d::get_bounding_box() const
{
    // Returns the minmax box of the Bezier surface, as a Box3d.
    // Reports an error for an empty Bezier surface.

    if ((no_points_t == 0)  ||  (no_points_u == 0))
        cerr << "Trying to find the bounding box of an empty Bezier surface."
             << endl << endl;

    float xmin = control_points.x(0, 0), xmax = control_points.x(0, 0),
          ymin = control_points.y(0, 0), ymax = control_points.y(0, 0),
          zmin = control_points.z(0, 0), zmax = control_points.z(0, 0);

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            float current_x = control_points.x(i, j);
            if (current_x < xmin)
                xmin = current_x;
            else if (current_x > xmax)
                xmax = current_x;

            float current_y = control_points.y(i, j);
            if (current_y < ymin)
                ymin = current_y;
            else if (current_y > ymax)
                ymax = current_y;

            float current_z = control_points.z(i, j);
            if (current_z < zmin)
                zmin = current_z;
            else if (current_z > zmax)
                zmax = current_z;
        }

    return Box3d(xmin, ymin, zmin, xmax, ymax, zmax);
}


GPoint3d GBezier3d::centre() const
{
    // Returns the centre of the Bezier surface, as the coordinatewise average
    // of its control points.
    // Returns (0, 0, 0) for an empty Bezier surface.

    if ((no_points_t == 0)  ||  (no_points_u == 0))  {
        cerr << "Trying to find the centre of an empty Bezier surface."
             << endl << endl;
```

```
        return GPoint3d(0.0, 0.0, 0.0);
    }

    float tot_x = 0,
          tot_y = 0,
          tot_z = 0;

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            tot_x += control_points.x(i, j);
            tot_y += control_points.y(i, j);
            tot_z += control_points.z(i, j);
        }

    const int no_pts = no_points_t * no_points_u;

    return GPoint3d(tot_x/no_pts, tot_y/no_pts, tot_z/no_pts);
}


void GBezier3d::subdivide(float t, float u,
                          GBezier3d& top_left, GBezier3d& top_right,
                          GBezier3d& bottom_left, GBezier3d& bottom_right) const
{
    // Subdivides the Bezier surface at parameter values (t, u).

    // The routine that actually does the subdivisions.
    void subdivide_vector(const GPoint3d* const bezier_pts, int no_pts, float t,
                          GPoint3d* const right_pts, GPoint3d* const left_pts);

    // First, subdivide each row of control points.
    GPoint3dArray right_pts(no_points_t, no_points_u);  // Temporary storage of
    GPoint3dArray left_pts(no_points_t, no_points_u);   // the new points.

    GPoint3d* control_row = new GPoint3d[no_points_u];
    GPoint3d* right_row = new GPoint3d[no_points_u];
    GPoint3d* left_row = new GPoint3d[no_points_u];

    for (int i = 0; i < no_points_t; i++)  {
        control_points.get_row(i, control_row);
        subdivide_vector(control_row, no_points_u, u, right_row, left_row);
        right_pts.set_row(i, right_row);
        left_pts.set_row(i, left_row);
    }

    delete[] control_row;
    delete[] right_row;
    delete[] left_row;

    // Then, subdivide each of the new columns.
    GPoint3dArray top_right_pts(no_points_t, no_points_u);
    GPoint3dArray top_left_pts(no_points_t, no_points_u);
    GPoint3dArray bottom_right_pts(no_points_t, no_points_u);
    GPoint3dArray bottom_left_pts(no_points_t, no_points_u);

    GPoint3d* control_col = new GPoint3d[no_points_t];
    GPoint3d* bottom_col = new GPoint3d[no_points_t];
    GPoint3d* top_col = new GPoint3d[no_points_t];

    for (int j = 0; j < no_points_u; j++)  {
        right_pts.get_col(j, control_col);
        subdivide_vector(control_col, no_points_t, t, bottom_col, top_col);
        bottom_right_pts.set_col(j, bottom_col);
        top_right_pts.set_col(j, top_col);
    }

    for (int k = 0; k < no_points_u; k++)  {
        left_pts.get_col(k, control_col);
        subdivide_vector(control_col, no_points_t, t, bottom_col, top_col);
        bottom_left_pts.set_col(k, bottom_col);
        top_left_pts.set_col(k, top_col);
    }

    delete[] control_col;
    delete[] bottom_col;
    delete[] top_col;

    // Make the four new Bezier surfaces.
    top_left = GBezier3d(top_left_pts);
    top_right = GBezier3d(top_right_pts);
    bottom_left = GBezier3d(bottom_left_pts);
    bottom_right = GBezier3d(bottom_right_pts);
}


void subdivide_vector(const GPoint3d* const bezier_pts, int no_pts, float t,
                      GPoint3d* const right_pts, GPoint3d* const left_pts)
{
    // Subdivides one vector of Bezier points (one row or column of the control
    // points of the surface) at parameter value t.
    // Essentially the same as subdivision for 2d.

    const float t1 = 1 - t;
    int r, i;   // Used in the for loops below.

    // Get the right half first. Each step of the de Casteljau algorithm as
    // implemented below overwrites all but the rightmost control point with
    // the new ones; so at the end we get the right half polygon.

    for (i = 0; i < no_pts; i++)
        right_pts[i] = bezier_pts[i];   // Initialise right half.

    for (r = 1; r < no_pts; r++)
        for (i = 0; i < no_pts - r; i++)
            // De Casteljau step.
            right_pts[i] = t1*right_pts[i] + t*right_pts[i+1];

    // Then get the left half. Essentially reverse the order of the control
```

```
    // points to keep the leftmost control point at each step.

    for (i = 0; i < no_pts; i++)
        left_pts[i] = bezier_pts[no_pts - i - 1];   // Initialise left half.

    for (r = 1; r < no_pts; r++)
        for (i = 0; i < no_pts - r; i++)
            // De Casteljau step.
            left_pts[i] = t*left_pts[i] + t1*left_pts[i+1];
}


int GBezier3d::is_flat(float tolerance) const
{
    // Checks if the Bezier surface is flat (in 3d a plane polygon) with
    // given tolerance. Essentially checks that all four sides of the polygon
    // are straight lines and that all control points lie on the same plane.

    // This function checks if the given points belong to the same line.
    int is_flat_line(int no_points, const GPoint3d* const points,
                     float tolerance);

    // Check flatness of four sides of the control polygon.
    GPoint3d* row = new GPoint3d[no_points_u];
    control_points.get_row(0, row);
    if (!is_flat_line(no_points_u, row, tolerance))
        return 0;

    control_points.get_row(no_points_t - 1, row);
    if (!is_flat_line(no_points_u, row, tolerance))
        return 0;
    delete[] row;

    GPoint3d* col = new GPoint3d[no_points_t];
    control_points.get_col(0, col);
    if (!is_flat_line(no_points_t, col, tolerance))
        return 0;

    control_points.get_col(no_points_u - 1, col);
    if (!is_flat_line(no_points_t, col, tolerance))
        return 0;
    delete[] col;

    // Now check all points for being co-planar.
    const float x1 = control_points.x(0, 0),
                y1 = control_points.y(0, 0),
                z1 = control_points.z(0, 0),
                x2 = control_points.x(0, no_points_u - 1),
                y2 = control_points.y(0, no_points_u - 1),
                z2 = control_points.z(0, no_points_u - 1),
                x3 = control_points.x(no_points_t - 1, 0),
                y3 = control_points.y(no_points_t - 1, 0),
                z3 = control_points.z(no_points_t - 1, 0);

    int i, j;   // Used in for loops below.

    const float tmp_x = y2*z3 - y3*z2,
                tmp_y = x2*z3 - x3*z2,
                tmp_z = x2*y3 - x3*y2,
                det_x = y1*(z2 - z3) - z1*(y2 - y3) + tmp_x,
                det_y = x1*(z2 - z3) - z1*(x2 - x3) + tmp_y,
                det_z = x1*(y2 - y3) - y1*(x2 - x3) + tmp_z,
                det_c = x1*tmp_x - y1*tmp_y + z1*tmp_z;
                const float surface = sqrt(det_x*det_x + det_y*det_y +
                                                                det_z*det_z);

    if (surface < 0.0001)  {   // Consider all points coincide.
        for (i = 0; i < no_points_t; i++)
            for (j = 0; j < no_points_u; j++)  {
                const float x = control_points.x(i, j),
                            y = control_points.y(i, j),
                            z = control_points.z(i, j);
                const float dist = sqrt((x - x1)*(x - x1) + (y - y1)*(y - y1) +
                                                             (z - z1)*(z - z1));
                if (dist > tolerance)
                    return 0;
            }
        return 1;
    }
    else  {
        for (i = 0; i < no_points_t; i++)
            for (j = 0; j < no_points_u; j++)  {
                const float x = control_points.x(i, j),
                            y = control_points.y(i, j),
                            z = control_points.z(i, j);
                const float det = x*det_x - y*det_y + z*det_z - det_c;
                const float dist = det / surface;
                if (fabs(dist) > tolerance)
                    return 0;
            }
        return 1;
    }
}


int is_flat_line(int no_points, const GPoint3d* const points, float tolerance)
{
    // Checks if all points belong to the same straight line, with given
    // tolerance. Essentially the same as the flatness test for Bezier curves.

    const float x1 = points[0].x(),
                y1 = points[0].y(),
                z1 = points[0].z(),
                x2 = points[no_points - 1].x(),
                y2 = points[no_points - 1].y(),
                z2 = points[no_points - 1].z();   // The end points.

    const float length = sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1) +
```

```
                                                    (z2 - z1)*(z2 - z1));
    // Length of line between the two end points.

    if (length < 0.0001)  {   // Consider the two end points coincide.
        for (int i = 1; i < no_points-1; i++)  {
            const float xi = points[i].x(),
                        yi = points[i].y(),
                        zi = points[i].z();
            const float dist = sqrt((x2 - xi)*(x2 - xi) + (y2 - yi)*(y2 - yi) +
                                                    (z2 - zi)*(z2 - zi));
            if (dist > tolerance)
                return 0;
        }
        return 1;
    }
    else  {
        const float tmp1 = x1*y2 - x2*y1,
                    tmp2 = x1*z2 - x2*z1,
                    tmp3 = y1*z2 - y2*z1;
        for (int i = 1; i < no_points-1; i++)  {
            const float xi = points[i].x(),
                        yi = points[i].y(),
                        zi = points[i].z();
            const float det = xi*tmp3 - yi*tmp2 + zi*tmp1;
            const float dist = det / length;
            if (fabs(dist) > tolerance)
                return 0;
        }
        return 1;
    }
}


void GBezier3d::transform(const Trans3d& trans)
{
    // Transforms the Bezier surface. Exploits the fact that it is invariant
    // under affine maps, and all basic transformations are affine maps; so to
    // transform the surface we only need to transform its control points.
    // Exception: Projection is *not* an affine map.

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            GPoint3d pt;
            control_points.get_cell(i, j, pt);
            pt.transform(trans);
            control_points.set_cell(i, j, pt);
      }
}


/*****************************************************************************/


GBSpline3d::GBSpline3d()
{
    // Constructor for a null B-Spline surface.

    degree_t = 0;
    degree_u = 0;
    no_points_t = 0;
    no_points_u = 0;
    control_points = GPoint3dArray();   // An empty one.
    no_knots_t = 0;
    knots_t = 0;   // Null pointer.
    knots_u = 0;   // Null pointer.
}


GBSpline3d::GBSpline3d(int deg_t, int deg_u, const GPoint3dArray& points)
{
    // Constructor for a B-Spline surface with given control points
    // and default knot sequence.

    degree_t = deg_t;
    degree_u = deg_u;
    no_points_t = points.no_rows();
    no_points_u = points.no_cols();
    no_knots_t = no_points_t - 1 + degree_t;
    no_knots_u = no_points_u - 1 + degree_u;

    if ((no_points_t > 0)  &&  (degree_t > 0)  &&
        (no_points_u > 0)  &&  (degree_u > 0))  {

        control_points = points;

        if (degree_t > no_points_t-1)  {
            degree_t = no_points_t - 1;
            cerr << "Not enough control points. A B-Spline of degree "
                 << degree_t << " for t will be constructed." << endl << endl;
        }

        if (degree_u > no_points_u-1)  {
            degree_u = no_points_u - 1;
            cerr << "Not enough control points. A B-Spline of degree "
                 << degree_u << " for u will be constructed." << endl << endl;
        }

        int i, j;   // Used in for loops below.

        knots_t = new float[no_knots_t];
        for (i = 0; i < degree_t; i++)
            knots_t[i] = 0;
        for (i = degree_t; i < no_points_t-1; i++)
            knots_t[i] = i - degree_t + 1;
        for (i = no_points_t-1; i < no_knots_t; i++)
            knots_t[i] = no_points_t - degree_t;

        knots_u = new float[no_knots_u];
```

```
        for (j = 0; j < degree_u; j++)
            knots_u[j] = 0;
        for (j = degree_u; j < no_points_u-1; j++)
            knots_u[j] = j - degree_u + 1;
        for (j = no_points_u-1; j < no_knots_u; j++)
            knots_u[j] = no_points_u - degree_u;
    }
}


GBSpline3d::GBSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                       const float* knot_seq_t, const float* knot_seq_u)
{
    // Constructor for a B-Spline surface with given control points
    // and knot sequences.

    degree_t = deg_t;
    degree_u = deg_u;
    no_points_t = points.no_rows();
    no_points_u = points.no_cols();
    no_knots_t = no_points_t - 1 + degree_t;
    no_knots_u = no_points_u - 1 + degree_u;

    if ((no_points_t > 0)  &&  (degree_t > 0)  &&
        (no_points_u > 0)  &&  (degree_u > 0))  {

        int valid_knots(int no_knots, const float* knots);

        if ( (!valid_knots(no_knots_t, knot_seq_t))  ||
             (!valid_knots(no_knots_u, knot_seq_u)) )  {
            cerr << "Knot sequences must be non-decreasing. "
                 << "Construction of B-Spline failed." << endl << endl;
        }

        else  {
            control_points = points;

            if (degree_t > no_points_t-1)  {
                degree_t = no_points_t - 1;
                no_knots_t = no_points_t - 1 + degree_t;
                cerr << "Not enough control points. A B-Spline of degree "
                     << degree_t << " for t will be constructed." << endl
                     << endl;
            }

            if (degree_u > no_points_u-1)  {
                degree_u = no_points_u - 1;
                no_knots_u = no_points_u - 1 + degree_u;
                cerr << "Not enough control points. A B-Spline of degree "
                     << degree_u << " for u will be constructed." << endl
                     << endl;
            }

            knots_t = new float[no_knots_t];
            for (int i = 0; i < no_knots_t; i++)
                knots_t[i] = knot_seq_t[i];

            knots_u = new float[no_knots_u];
            for (int j = 0; j < no_knots_u; j++)
                knots_u[j] = knot_seq_u[j];
        }
    }
}


GBSpline3d::GBSpline3d(const GBSpline3d& bspline)
{
    // Copy constructor for B-Spline surfaces.
    // cf. Stroustrup, section 7.6

    degree_t = bspline.degree_t;
    degree_u = bspline.degree_u;
    no_points_t = bspline.no_points_t;
    no_points_u = bspline.no_points_u;
    no_knots_t = bspline.no_knots_t;
    no_knots_u = bspline.no_knots_u;

    if ((no_points_t > 0)  &&  (degree_t > 0)  &&
        (no_points_u > 0)  &&  (degree_u > 0))  {

        bspline.control_pts(control_points);

        knots_t = new float[no_knots_t];
        bspline.knot_sequence_t(knots_t);

        knots_u = new float[no_knots_u];
        bspline.knot_sequence_u(knots_u);
    }
}


GBSpline3d::~GBSpline3d()
{
    // Destructor function for GBSpline3d.

    if (no_knots_t > 0)
        delete[] knots_t;
    if (no_knots_u > 0)
        delete[] knots_u;
}


GBSpline3d& GBSpline3d::operator=(const GBSpline3d& bspline)
{
    // Assignement operator for GBSpline3d.
    // cf. Stroustrup, section 7.6

    if (this != &bspline)  {
```

```
        if (no_knots_t > 0)
            delete[] knots_t;
        if (no_knots_u > 0)
            delete[] knots_u;

        degree_t = bspline.degree_t;
        degree_u = bspline.degree_u;
        no_points_t = bspline.no_points_t;
        no_points_u = bspline.no_points_u;
        no_knots_t = bspline.no_knots_t;
        no_knots_u = bspline.no_knots_u;

        if ((no_points_t > 0)  &&  (degree_t > 0)  &&
            (no_points_u > 0)  &&  (degree_u > 0))  {

            bspline.control_pts(control_points);

            knots_t = new float[no_knots_t];
            bspline.knot_sequence_t(knots_t);

            knots_u = new float[no_knots_u];
            bspline.knot_sequence_u(knots_u);
        }
    }
    return *this;
}


int GBSpline3d::deg_t() const
{
    // Returns the degree of the B-Spline surface with respect to parameter t.

    return degree_t;
}


int GBSpline3d::deg_u() const
{
    // Returns the degree of the B-Spline surface with respect to parameter u.

    return degree_u;
}


int GBSpline3d::no_pts_t() const
{
    // Returns the number of control points for t of the B-Spline surface.

    return no_points_t;
}


int GBSpline3d::no_pts_u() const
{
    // Returns the number of control points for u of the B-Spline surface.

    return no_points_u;
}


void GBSpline3d::control_pts(GPoint3dArray& points) const
{
    // points is a pointer to the array of control points of the surface.

    points = control_points;
}


int GBSpline3d::no_knts_t() const
{
    // Returns the number of knots for t in the B-Spline surface.

    return no_knots_t;
}


int GBSpline3d::no_knts_u() const
{
    // Returns the number of knots for u in the B-Spline surface.

    return no_knots_u;
}


void GBSpline3d::knot_sequence_t(float* knot_seq) const
{
    // knot_seq is a pointer to the array of knots for t of the B-Spline surface

    for (int i = 0; i < no_knots_t; i++)
        knot_seq[i] = knots_t[i];
}


void GBSpline3d::knot_sequence_u(float* knot_seq) const
{
    // knot_seq is a pointer to the array of knots for t of the B-Spline surface

    for (int j = 0; j < no_knots_u; j++)
        knot_seq[j] = knots_u[j];
}


void GBSpline3d::insert_knot_t(int index, const float knot_value)
{
    // Insertion of knot_value in the interval with lower bound index
    // 'index' for parameter t. We use Boehm's knot insertion algorithm.

    // Check the insertion is in allowed interval. Allowed intervals start at
```

```
    // t[i], with i = degree_t-1, ..., no_points_t-2.
    if ((index < degree_t-1)  ||  (index > no_points_t-2))
        cerr << "Insertion in this interval not allowed. "
             << "Knot insertion aborted." << endl << endl;

    // Check for allowed knot value.
    else if ((knot_value < knots_t[index])  ||  (knot_value > knots_t[index+1]))
        cerr << "New knot value must be between knot values at interval ends."
             << endl << "Knot insertion aborted." << endl << endl;

    else  {

        // The function that actually produces the new control points.
        void insert_knot_in_vector(GPoint3d* points, int no_points, int degree,
                                   float* knots, int index, const float knot_value,
                                   GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t + 1, no_points_u);
        GPoint3d* col = new GPoint3d[no_points_t];
        GPoint3d* new_col = new GPoint3d[no_points_t + 1];

        int j;   // Used in for loops below.

        for (j = 0; j < no_points_u; j++)  {
            control_points.get_col(j, col);
            insert_knot_in_vector(col, no_points_t, degree_t,
                                  knots_t, index, knot_value, new_col);
            new_points.set_col(j, new_col);
        }

        delete[] col;
        delete[] new_col;

        // Tidy up B-Spline surface.
        no_points_t += 1;
        control_points = new_points;

        no_knots_t += 1;
        float* new_knots_t = new float[no_knots_t];

        for (j = 0; j <= index; j++)
            new_knots_t[j] = knots_t[j];
        new_knots_t[index+1] = knot_value;
        for (j = index+2; j < no_knots_t; j++)
            new_knots_t[j] = knots_t[j-1];

        delete[] knots_t;
        knots_t = new_knots_t;
    }
}


void GBSpline3d::insert_knot_u(int index, const float knot_value)
{
    // Insertion of knot_value in the interval with lower bound index
    // 'index' for parameter u. We use Boehm's knot insertion algorithm.

    // Check the insertion is in allowed interval. Allowed intervals start at
    // u[i], with i = degree_u-1, ..., no_points_u-2.
    if ((index < degree_u-1)  ||  (index > no_points_u-2))
        cerr << "Insertion in this interval not allowed. "
             << "Knot insertion aborted." << endl << endl;

    // Check for allowed knot value.
    else if ((knot_value < knots_u[index])  ||  (knot_value > knots_u[index+1]))
        cerr << "New knot value must be between knot values at interval ends."
             << endl << "Knot insertion aborted." << endl << endl;

    else  {

        // The function that actually produces the new control points.
        void insert_knot_in_vector(GPoint3d* points, int no_points, int degree,
                                   float* knots, int index, const float knot_value,
                                   GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t, no_points_u + 1);
        GPoint3d* row = new GPoint3d[no_points_u];
        GPoint3d* new_row = new GPoint3d[no_points_u + 1];

        int i;   // Used in for loops below.

        for (i = 0; i < no_points_t; i++)  {
            control_points.get_row(i, row);
            insert_knot_in_vector(row, no_points_u, degree_u,
                                  knots_u, index, knot_value, new_row);
            new_points.set_row(i, new_row);
        }

        delete[] row;
        delete[] new_row;

        // Tidy up B-Spline surface.
        no_points_u += 1;
        control_points = new_points;

        no_knots_u += 1;
        float* new_knots_u = new float[no_knots_u];

        for (i = 0; i <= index; i++)
            new_knots_u[i] = knots_u[i];
        new_knots_u[index+1] = knot_value;
        for (i = index+2; i < no_knots_u; i++)
            new_knots_u[i] = knots_u[i-1];

        delete[] knots_u;
        knots_u = new_knots_u;
    }
}
```

```
void insert_knot_in_vector(GPoint3d* points, int no_points, int degree,
                           float* knots, int index, const float knot_value,
                           GPoint3d* new_points)
{
    // Produces new control points when knot_value is inserted in the interval
    // with lower bound index 'index'. We use Boehm's knot insertion algorithm.

    int j;   // Used in for loops below.

    for (j = 0; j < index-degree+2; j++)
        new_points[j] = points[j];

    for (j = index-degree+2; j < index+2; j++)  {
        const float coeff = (knots[j+degree-1] - knot_value) /
                            (knots[j+degree-1] - knots[j-1]);
        new_points[j] = coeff*points[j-1] + (1-coeff)*points[j];
    }

    for (j = index+2; j < no_points+1; j++)
        new_points[j] = points[j-1];
}


void GBSpline3d::insert_many_knots_t(int index,
                                     int no_knts, const float* knot_values)
{
    // Insertion of multiple knots in the same interval for t (Oslo algorithm).
    // Only applicable for insertion of (degree_t) or (degree_t+1) knots.

    int error = 0;   // Set to 1 if an error occurs.

    // Check for applicability.
    if ((no_knts > degree_t+1)  ||  (no_knts < degree_t))  {
        cerr << "Only " << degree_t << " or " << degree_t+1 << " knots can be "
             << "inserted simultaneously." << endl
             << "Inserting them sequentially instead." << endl << endl;
        error = 1;
        for (int i = 0; i < no_knts; i++)
            insert_knot_t(index+i, knot_values[i]);
    }

    // Check the insertion is in allowed interval. Allowed intervals start at
    // t[i], with i = degree_t-1, ..., no_points_t-2.
    if (!error)
        if ((index < degree_t-1)  ||  (index > no_points_t-2))  {
            cerr << "Insertion in this interval not allowed. "
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }

    // Check for allowed knot values.
    if (!error)
        for (int i = 0; (i < no_knts  &&  !error); i++)
            if ((knot_values[i] < knots_t[index])  ||
                (knot_values[i] > knots_t[index+1]))  {
                cerr << "New knot values must be between knot values "
                     << "at interval ends." << endl
                     << "Knot insertion aborted." << endl << endl;
                error = 1;
            }
    if (!error)  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knts, knot_values))  {
            cerr << "New knot values must be non-decreasing." << endl
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }
    }

    if (!error)  {   // Proceed.

        // The function that actually produces the new control points.
        void insert_many_knots_in_vector(GPoint3d* points, int no_points,
                                         int degree, float* knots, int index,
                                         int no_knts, const float* knot_values,
                                         GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t + no_knts, no_points_u);
        GPoint3d* col = new GPoint3d[no_points_t];
        GPoint3d* new_col = new GPoint3d[no_points_t + no_knts];

        int j;   // Used in for loops below.

        for (j = 0; j < no_points_u; j++)  {
            control_points.get_col(j, col);
            insert_many_knots_in_vector(col, no_points_t, degree_t, knots_t,
                                        index, no_knts, knot_values, new_col);
            new_points.set_col(j, new_col);
        }

        delete[] col;
        delete[] new_col;

        no_points_t += no_knts;
        control_points = new_points;

        // Tidy up the knot sequence.
        no_knots_t += no_knts;
        float* new_knots_t = new float[no_knots_t];

        for (j = 0; j <= index; j++)
            new_knots_t[j] = knots_t[j];
        for (j = index+1; j < index+1+no_knts; j++)
            new_knots_t[j] = knot_values[j-index-1];
        for (j = index+no_knts+1; j < no_knots_t; j++)
```

```
            new_knots_t[j] = knots_t[j-no_knts];

        delete[] knots_t;
        knots_t = new_knots_t;
    }
}


void GBSpline3d::insert_many_knots_u(int index,
                                     int no_knts, const float* knot_values)
{
    // Insertion of multiple knots in the same interval for u (Oslo algorithm).
    // Only applicable for insertion of (degree_u) or (degree_u+1) knots.

    int error = 0;   // Set to 1 if an error occurs.

    // Check for applicability.
    if ((no_knts > degree_u+1)  ||  (no_knts < degree_u))  {
        cerr << "Only " << degree_u << " or " << degree_u+1 << " knots can be "
             << "inserted simultaneously." << endl
             << "Inserting them sequentially instead." << endl << endl;
        error = 1;
        for (int j = 0; j < no_knts; j++)
            insert_knot_u(index+j, knot_values[j]);
    }

    // Check the insertion is in allowed interval. Allowed intervals start at
    // u[j], with j = degree_u-1, ..., no_points_u-2.
    if (!error)
        if ((index < degree_u-1)  ||  (index > no_points_u-2))  {
            cerr << "Insertion in this interval not allowed. "
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }

    // Check for allowed knot values.
    if (!error)
        for (int j = 0; (j < no_knts  &&  !error); j++)
            if ((knot_values[j] < knots_u[index])  ||
                (knot_values[j] > knots_u[index+1]))  {
                cerr << "New knot values must be between knot values "
                     << "at interval ends." << endl
                     << "Knot insertion aborted." << endl << endl;
                error = 1;
            }
    if (!error)  {
        int valid_knots(int no_knots, const float* knots);

        if (!valid_knots(no_knts, knot_values))  {
            cerr << "New knot values must be non-decreasing." << endl
                 << "Knot insertion aborted." << endl << endl;
            error = 1;
        }
    }

    if (!error)  {   // Proceed.

        // The function that actually produces the new control points.
        void insert_many_knots_in_vector(GPoint3d* points, int no_points,
                                         int degree, float* knots, int index,
                                         int no_knts, const float* knot_values,
                                         GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t, no_points_u + no_knts);
        GPoint3d* row = new GPoint3d[no_points_u];
        GPoint3d* new_row = new GPoint3d[no_points_u + no_knts];

        int i;   // Used in for loops below.

        for (i = 0; i < no_points_t; i++)  {
            control_points.get_row(i, row);
            insert_many_knots_in_vector(row, no_points_u, degree_u, knots_u,
                                        index, no_knts, knot_values, new_row);
            new_points.set_row(i, new_row);
        }

        delete[] row;
        delete[] new_row;

        no_points_u += no_knts;
        control_points = new_points;

        // Tidy up the knot sequence.
        no_knots_u += no_knts;
        float* new_knots_u = new float[no_knots_u];

        for (i = 0; i <= index; i++)
            new_knots_u[i] = knots_u[i];
        for (i = index+1; i < index+1+no_knts; i++)
            new_knots_u[i] = knot_values[i-index-1];
        for (i = index+no_knts+1; i < no_knots_u; i++)
            new_knots_u[i] = knots_u[i-no_knts];

        delete[] knots_u;
        knots_u = new_knots_u;
    }
}


void insert_many_knots_in_vector(GPoint3d* points, int no_points,
                                 int degree, float* knots, int index,
                                 int no_knts, const float* knot_values,
                                 GPoint3d* new_points)
{
    // Produces the new control points when multiple knots are inserted in the
    // vector of control points given. We use the Oslo algorithm.

    int j, k;   // Used in for loops below.
```

```
    for (j = 0; j < index-degree+2; j++)
        new_points[j] = points[j];

    GPoint3d pts[degree+1];
    // Points used in the de Casteljau triangles.

    // Calculate first (degree) new points, in increasing order.
    for (j = 0; j <= degree; j++)
        pts[j] = points[j+index-degree+1];

    for (j = 0; j < degree; j++)
        for (k = degree; k >= j+1; k--)  {
            const float coeff= (knots[k-j+index] - knot_values[j]) /
                               (knots[k-j+index] - knots[k-j+index-degree]);
            pts[k] = coeff*pts[k-1] + (1-coeff)*pts[k];
        }

    // Insert them in new_points.
    for (j = index-degree+2; j < index+2; j++)
        new_points[j] = pts[j-index+degree-1];

    // Calculate remaining new points, in decreasing order.
    for (j = 0; j <= degree; j++)
        pts[j] = points[j+index-degree+1];

    // s is 0 or 1 according to no_knts.
    const int s = 1-no_knts+degree;
    for (j = 0; j < degree-s; j++)
        for (k = s; k <= degree-j-1; k++)  {
            const float coeff= (knots[k+j+index+1] - knot_values[no_knts-1-j]) /
                               (knots[k+j+index+1] - knots[k+j+index+1-degree]);
            pts[k] = coeff*pts[k] + (1-coeff)*pts[k+1];
        }

    // Insert them in new_points.
    for (j = index+2; j < index+no_knts+1; j++)
        new_points[j] = pts[j-index-2+s];

    // Insert remaining control points in new_points.
    for (j = index+no_knts+1; j < no_points+no_knts; j++)
        new_points[j] = points[j-no_knts];
}


void GBSpline3d::delete_knot_t(int index)
{
    // Deletion of the knot indexed 'index' with respect to parameter t.
    // Only knots with index degree_t, ..., no_points_t-2 may be deleted.

    // Check for allowed knot to be deleted.
    if ((index < degree_t) || (index > no_points_t-1))
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    // Check if it is the first of three equal knots.
    else if ((knots_t[index] == knots_t[index+1])  &&
             (knots_t[index] == knots_t[index+2]))
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    else  {

        // The function that produces the new control points.
        void delete_knot_in_vector(GPoint3d* points, int no_points, int degree,
                                   float* knots, int index, GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t - 1, no_points_u);
        GPoint3d* col = new GPoint3d[no_points_t];
        GPoint3d* new_col = new GPoint3d[no_points_t - 1];

        int j;   // Used in for loops below.

        for (j = 0; j < no_points_u; j++)  {
            control_points.get_col(j, col);
            delete_knot_in_vector(col, no_points_t, degree_t,
                                  knots_t, index, new_col);
            new_points.set_col(j, new_col);
        }

        delete[] col;
        delete[] new_col;

        // Tidy up the B-Spline surface.
        no_points_t -= 1;
        control_points = new_points;

        no_knots_t -= 1;
        float* new_knots_t = new float[no_knots_t];

        for (j = 0; j < index; j++)
            new_knots_t[j] = knots_t[j];
        for (j = index; j < no_knots_t; j++)
            new_knots_t[j] = knots_t[j+1];

        delete[] knots_t;
        knots_t = new_knots_t;
    }
}


void GBSpline3d::delete_knot_u(int index)
{
    // Deletion of the knot indexed 'index' with respect to parameter u.
    // Only knots with index degree_u, ..., no_points_u-2 may be deleted.

    // Check for allowed knot to be deleted.
    if ((index < degree_u) || (index > no_points_u-1))
```

```
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    // Check if it is the first of three equal knots.
    else if ((knots_u[index] == knots_u[index+1])  &&
             (knots_u[index] == knots_u[index+2]))
        cerr << "Deletion of knot " << index << " not allowed. "
             << "Procedure aborted." << endl << endl;

    else  {

        // The function that produces the new control points.
        void delete_knot_in_vector(GPoint3d* points, int no_points, int degree,
                                   float* knots, int index, GPoint3d* new_points);

        GPoint3dArray new_points(no_points_t, no_points_u - 1);
        GPoint3d* row = new GPoint3d[no_points_u];
        GPoint3d* new_row = new GPoint3d[no_points_u - 1];

        int i;   // Used in for loops below.

        for (i = 0; i < no_points_t; i++)  {
            control_points.get_row(i, row);
            delete_knot_in_vector(row, no_points_u, degree_u,
                                  knots_u, index, new_row);
            new_points.set_row(i, new_row);
        }

        delete[] row;
        delete[] new_row;

        // Tidy up the B-Spline surface.
        no_points_u -= 1;
        control_points = new_points;

        no_knots_u -= 1;
        float* new_knots_u = new float[no_knots_u];

        for (i = 0; i < index; i++)
            new_knots_u[i] = knots_u[i];
        for (i = index; i < no_knots_u; i++)
            new_knots_u[i] = knots_u[i+1];

        delete[] knots_u;
        knots_u = new_knots_u;
    }
}


void delete_knot_in_vector(GPoint3d* points, int no_points, int degree,
                           float* knots, int index, GPoint3d* new_points)
{
    // Produces new control points when knot indexed 'index' is deleted.

    int j;   // Used in for loops below.

    for (j = no_points-2; j > index-1; j--)
        new_points[j] = points[j+1];

    for (j = index-1; j > index-degree; j--)  {
        const float coeff = (knots[j+degree+1] - knots[j]) /
                            (knots[j+degree+1] - knots[index]);
        new_points[j] = coeff*points[j+1]+(1-coeff)*new_points[j+1];
    }

    for (j = index-degree; j >= 0; j--)
        new_points[j] = points[j];
}


void GBSpline3d::transform(const Trans3d& trans)
{
    // Transformation of the B-Spline surface. As for Bezier surfaces,
    // we only need to transform the control polygon.

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            GPoint3d pt;
            control_points.get_cell(i, j, pt);
            pt.transform(trans);
            control_points.set_cell(i, j, pt);
      }
}


Box3d GBSpline3d::get_bounding_box() const
{
    // Returns the minmax box of the Bezier surface, as a Box3d.

    if ((no_points_t == 0)  ||  (no_points_u == 0))
        cerr << "Trying to find the bounding box of an empty B-Spline surface."
             << endl << endl;

    float xmin = control_points.x(0, 0), xmax = control_points.x(0, 0),
          ymin = control_points.y(0, 0), ymax = control_points.y(0, 0),
          zmin = control_points.z(0, 0), zmax = control_points.z(0, 0);

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            float current_x = control_points.x(i, j);
            if (current_x < xmin)
                xmin = current_x;
            else if (current_x > xmax)
                xmax = current_x;

            float current_y = control_points.y(i, j);
            if (current_y < ymin)
                ymin = current_y;
```

```
            else if (current_y > ymax)
                ymax = current_y;

            float current_z = control_points.z(i, j);
            if (current_z < zmin)
                zmin = current_z;
            else if (current_z > zmax)
                zmax = current_z;
        }

    return Box3d(xmin, ymin, zmin, xmax, ymax, zmax);
}


GPoint3d GBSpline3d::centre() const
{
    // Returns the centre of the B-Spline surface, as the coordinatewise average
    // of its control points.
    // Returns (0, 0, 0) for an empty B-Spline surface.

    if ((no_points_t == 0)  ||  (no_points_u == 0))  {
        cerr << "Trying to find the centre of an empty B-Spline surface."
             << endl << endl;
        return GPoint3d(0.0, 0.0, 0.0);
    }

    float tot_x = 0,
          tot_y = 0,
          tot_z = 0;

    for (int i = 0; i < no_points_t; i++)
        for (int j = 0; j < no_points_u; j++)  {
            tot_x += control_points.x(i, j);
            tot_y += control_points.y(i, j);
            tot_z += control_points.z(i, j);
        }

    const int no_pts = no_points_t * no_points_u;

    return GPoint3d(tot_x/no_pts, tot_y/no_pts, tot_z/no_pts);
}


void GBSpline3d::interpolate(const GPoint3dArray& pts,
                             const GPoint3d* r1, const GPoint3d* r2,
                             const GPoint3d* c1, const GPoint3d* c2,
                             const GPoint3d r1c1, const GPoint3d r1c2,
                             const GPoint3d r2c1, const GPoint3d r2c2)
{
    // Interpolation with cubic (for t and u) B-Spline surfaces.
    // The knots are uniformly spaced for t and u.
    // r1, r2, c1, c2, r1c1, r1c2, r2c1, r2c2 contain the arbitrary points.

    if ((pts.no_rows() > 1)  &&  (pts.no_cols() > 1))  {

        int i, j;   // Used in for loops below.

        // The elements of the interpolating B-Spline surface.
        const int no_pts_t_bspl = pts.no_rows() + 2;
        const int no_pts_u_bspl = pts.no_cols() + 2;
        const int deg_t = 3;
        const int deg_u = 3;
        const int no_knots_t = no_pts_t_bspl - 1 + deg_t;
        const int no_knots_u = no_pts_u_bspl - 1 + deg_u;

        // Construct uniform knot sequences.
        float* knots_t = new float[no_knots_t];
        for (i = 0; i < deg_t; i++)
            knots_t[i] = 0;
        for (i = deg_t; i < no_pts_t_bspl - 1; i++)
            knots_t[i] = i - deg_t + 1;
        for (i = no_pts_t_bspl - 1; i < no_knots_t; i++)
            knots_t[i] = no_pts_t_bspl - deg_t;

        float* knots_u = new float[no_knots_u];
        for (j = 0; j < deg_u; j++)
            knots_u[j] = 0;
        for (j = deg_u; j < no_pts_u_bspl - 1; j++)
            knots_u[j] = j - deg_u + 1;
        for (j = no_pts_u_bspl - 1; j < no_knots_u; j++)
            knots_u[j] = no_pts_u_bspl - deg_u;

        interpolate(pts, knots_t, knots_u, r1, r2, c1, c2,
                    r1c1, r1c2, r2c1, r2c2);

        delete[] knots_t;
        delete[] knots_u;
    }

    else  {
        *this = GBSpline3d();   // An empty one.
    }
}


void GBSpline3d::interpolate(const GPoint3dArray& pts,
                             const float* knots_t, const float* knots_u,
                             const GPoint3d* r1, const GPoint3d* r2,
                             const GPoint3d* c1, const GPoint3d* c2,
                             const GPoint3d r1c1, const GPoint3d r1c2,
                             const GPoint3d r2c1, const GPoint3d r2c2)
{
    // Interpolation with cubic (for t and u) B-Spline surfaces.
    // The knots are provided for t and u.
    // r1, r2, c1, c2, r1c1, r1c2, r2c1, r2c2 contain the arbitrary points.

    // The function that actually does the interpoalations.
    void interpolate_vector(int no_pts_bspl, const float* low,
```

```
                        const float* up, const float* gamma,
                        const GPoint3d* rhs, GPoint3d* solution);

if ((pts.no_rows() > 1)  &&  (pts.no_cols() > 1))  {

    int i, j;   // Used in for loops below.

    const int no_pts_t = pts.no_rows();
    const int no_pts_u = pts.no_cols();

    // First work for t.

    const int no_pts_t_bspl = no_pts_t + 2;
    const int deg_t = 3;

    float* alpha_t = new float[no_pts_t_bspl];
    float* beta_t = new float[no_pts_t_bspl];
    float* gamma_t = new float[no_pts_t_bspl];

    alpha_t[1] = 0;
    beta_t[0] = 1;
    beta_t[1] = 1;
    gamma_t[0] = 0;
    gamma_t[1] = 0;

    // Prepare system for LU-analysis.
    for (i = 1; i < no_pts_t-1; i++)  {
        const float temp1 = (knots_t[i+3] - knots_t[i+2]) /
                                              (knots_t[i+3] - knots_t[i]);
        const float temp2 = (knots_t[i+2] - knots_t[i+1]) /
                                                (knots_t[i+4] - knots_t[i+1]);

        alpha_t[i+1] = temp1*(knots_t[i+3] - knots_t[i+2]);
        beta_t[i+1] = temp1*(knots_t[i+2] - knots_t[i]) +
                                           temp2*(knots_t[i+4]-knots_t[i+2]);
        gamma_t[i+1] = temp2*(knots_t[i+2] - knots_t[i+1]);
    }

    alpha_t[no_pts_t_bspl - 2] = 0;
    alpha_t[no_pts_t_bspl - 1] = 0;
    beta_t[no_pts_t_bspl - 2] = 1;
    beta_t[no_pts_t_bspl - 1] = 1;
    gamma_t[no_pts_t_bspl - 2] = 0;

    // Analyse the matrix of the system into L and U once.
    float* low_t = new float[no_pts_t_bspl];
    float* up_t = new float[no_pts_t_bspl];
    l_u_analysis(no_pts_t_bspl, alpha_t, beta_t, gamma_t, low_t, up_t);

    GPoint3d* rhs_t = new GPoint3d[no_pts_t_bspl];
    GPoint3d* solution_t = new GPoint3d[no_pts_t_bspl];

    GPoint3dArray temp_pts(no_pts_t_bspl, no_pts_u);   // Intermediate pts.

    for (j = 0; j < no_pts_u; j++)  {   // Interpolate each column.

        // Form right-hand side of system.
        pts.get_cell(0, j, rhs_t[0]);
        rhs_t[1] = r1[j];

        for (i = 1; i < no_pts_t-1; i++)  {
            GPoint3d pt;
            pts.get_cell(i, j, pt);
            rhs_t[i+1] = (knots_t[i+3] - knots_t[i+1]) * pt;
      }

        rhs_t[no_pts_t_bspl - 2] = r2[j];
        pts.get_cell(no_pts_t-1, j, rhs_t[no_pts_t_bspl - 1]);

        interpolate_vector(no_pts_t_bspl, low_t, up_t, gamma_t, rhs_t,
                           solution_t);

        temp_pts.set_col(j, solution_t);
    }

    // Also we need to interpolate the two columns c1 and c2: First c1...
    GPoint3d temp_c1[no_pts_t_bspl];

    rhs_t[0] = c1[0];
    rhs_t[1] = r1c1;

    for (i = 1; i < no_pts_t-1; i++)
        rhs_t[i+1] = (knots_t[i+3] - knots_t[i+1]) * c1[i];

    rhs_t[no_pts_t_bspl - 2] = r2c1;
    rhs_t[no_pts_t_bspl - 1] = c1[no_pts_t - 1];

    interpolate_vector(no_pts_t_bspl, low_t, up_t, gamma_t, rhs_t, temp_c1);

    // ... and then c2.
    GPoint3d temp_c2[no_pts_t_bspl];

    rhs_t[0] = c2[0];
    rhs_t[1] = r1c2;

    for (i = 1; i < no_pts_t-1; i++)
        rhs_t[i+1] = (knots_t[i+3] - knots_t[i+1]) * c2[i];

    rhs_t[no_pts_t_bspl - 2] = r2c2;
    rhs_t[no_pts_t_bspl - 1] = c2[no_pts_t - 1];

    interpolate_vector(no_pts_t_bspl, low_t, up_t, gamma_t, rhs_t, temp_c2);

    delete[] alpha_t;
    delete[] beta_t;
    delete[] gamma_t;
    delete[] low_t;
    delete[] up_t;
```

```
delete[] rhs_t;
delete[] solution_t;

// Then repeat for u!

const int no_pts_u_bspl = no_pts_u + 2;
const int deg_u = 3;

float* alpha_u = new float[no_pts_u_bspl];
float* beta_u = new float[no_pts_u_bspl];
float* gamma_u = new float[no_pts_u_bspl];

alpha_u[1] = 0;
beta_u[0] = 1;
beta_u[1] = 1;
gamma_u[0] = 0;
gamma_u[1] = 0;

// Prepare system for LU-analysis.
for (j = 1; j < no_pts_u-1; j++)  {
    const float temp1 = (knots_u[j+3] - knots_u[j+2]) /
                                        (knots_u[j+3] - knots_u[j]);
    const float temp2 = (knots_u[j+2] - knots_u[j+1]) /
                                      (knots_u[j+4] - knots_u[j+1]);

    alpha_u[j+1] = temp1*(knots_u[j+3] - knots_u[j+2]);
    beta_u[j+1] = temp1*(knots_u[j+2] - knots_u[j]) +
                                    temp2*(knots_u[j+4]-knots_u[j+2]);
    gamma_u[j+1] = temp2*(knots_u[j+2] - knots_u[j+1]);
}

alpha_u[no_pts_u_bspl - 2] = 0;
alpha_u[no_pts_u_bspl - 1] = 0;
beta_u[no_pts_u_bspl - 2] = 1;
beta_u[no_pts_u_bspl - 1] = 1;
gamma_u[no_pts_u_bspl - 2] = 0;

// Analyse the matrix of the system into L and U once.
float low_u[no_pts_u_bspl];
float up_u[no_pts_u_bspl];
l_u_analysis(no_pts_u_bspl, alpha_u, beta_u, gamma_u, low_u, up_u);

GPoint3d* rhs_u = new GPoint3d[no_pts_u_bspl];
GPoint3d* solution_u = new GPoint3d[no_pts_u_bspl];

GPoint3dArray bspl_pts(no_pts_t_bspl, no_pts_u_bspl);   // Final points.

for (i = 0; i < no_pts_t_bspl; i++)  {   // Interpolate each row.

    // Form right-hand side of system.
    temp_pts.get_cell(i, 0, rhs_u[0]);
    rhs_u[1] = temp_c1[i];

    for (j = 1; j < no_pts_u-1; j++)  {
        GPoint3d pt;
        temp_pts.get_cell(i, j, pt);
        rhs_u[j+1] = (knots_u[j+3] - knots_u[j+1]) * pt;
    }

    rhs_u[no_pts_u_bspl - 2] = temp_c2[i];
    temp_pts.get_cell(i, no_pts_u-1, rhs_u[no_pts_u_bspl - 1]);

    interpolate_vector(no_pts_u_bspl, low_u, up_u, gamma_u, rhs_u,
                       solution_u);

    bspl_pts.set_row(i, solution_u);
}

delete[] alpha_u;
delete[] beta_u;
delete[] gamma_u;
delete[] low_u;
delete[] up_u;
delete[] rhs_u;
delete[] solution_u;

GBSpline3d gbspl(deg_t, deg_u, bspl_pts, knots_t, knots_u);
*this = gbspl;
}

else  {
    *this = GBSpline3d();   // An empty one.
}
}


void interpolate_vector(int no_pts_bspl, const float* low,
                        const float* up, const float* gamma,
                        const GPoint3d* rhs, GPoint3d* solution)
{
    // Performs interpolation with the elements specified.
    // The result is put in solution.

    int j;   // Used in for loops below.

    float* d = new float[no_pts_bspl];   // Right-hand side of the systems.
    float* sol_x = new float[no_pts_bspl];   // Solutions of the systems
    float* sol_y = new float[no_pts_bspl];   // for x, y and z.
    float* sol_z = new float[no_pts_bspl];

    // Solve system for x.
    for (j = 0; j < no_pts_bspl; j++)
        d[j] = rhs[j].x();

    solve_system(no_pts_bspl, low, up, gamma, d, sol_x);

    // Solve system for y.
    for (j = 0; j < no_pts_bspl; j++)
```

```
        d[j] = rhs[j].y();

    solve_system(no_pts_bspl, low, up, gamma, d, sol_y);

    // Solve system for z.
    for (j = 0; j < no_pts_bspl; j++)
        d[j] = rhs[j].z();

    solve_system(no_pts_bspl, low, up, gamma, d, sol_z);

    // Form solution.
    for (j = 0; j < no_pts_bspl; j++)  {
        GPoint3d pt(sol_x[j], sol_y[j], sol_z[j]);
        solution[j] = pt;
    }

    delete[] d;
    delete[] sol_x;
    delete[] sol_y;
    delete[] sol_z;
}


int valid_knots(int no_knots, const float* knots)
{
    // Checks if the sequence of knots is non-decreasing to be valid.

    for (int i = 0; i < no_knots-1; i++)
        if (knots[i] > knots[i+1])
            return 0;
    return 1;
}
```

```
/* File Surface3d.cc
 *
 * Implements header file Curve3d.h containing classes Bezier3d and
 * BSpline3d, for the representation of (the drawable primitives of) Bezier
 * and B-Spline surfaces in 2D.
 *
 * Author: Nikolaos Platis,
 *         MSc in Information Technology 1995-1996,
 *         Department of Computer Science, University College London.
 */


#include "graphics2d.h"
#include "Atom3d.h"
#include "GObject3d.h"
#include "Vector3d.h"
#include "GPrimitive3d.h"
#include "Trans3d.h"
#include "Box3d.h"
#include "Camera.h"
#include "View3d.h"
#include "Drawable3d.h"
#include "Primitive3d.h"
#include "Object3d.h"
#include "SimpleObject3d.h"
#include "DPrimitive3d.h"
#include "light.h"
#include "model.h"
#include "Pixel.h"
#include "Colour.h"
#include "BRep.h"
#include "GSurface3d.h"
#include "Surface3d.h"
#include <math.h>


Bezier3d::Bezier3d()
{
    // Constructor for a null Bezier surface.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


Bezier3d::Bezier3d(const GBezier3d& bezier)
    : GBezier3d(bezier)
{
    // Constructor for a Bezier surface from a GBezier3d
    // with no attributes specified.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


Bezier3d::Bezier3d(const Attributes& attributes)
    : Attributes(attributes)
{
    // Constructor for a null Bezier surface, with attributes specified.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


Bezier3d::Bezier3d(const GBezier3d& bezier, const Attributes& attributes)
    : GBezier3d(bezier), Attributes(attributes)
{
    // Constructor for a Bezier surface from a GBezier3d
    // with attributes specified.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


Bezier3d::Bezier3d(const GPoint3dArray& points)
    : GBezier3d(points)
{
    // Constructor for a Bezier surface with given control points and
    // default attributes.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


Bezier3d::Bezier3d(const GPoint3dArray& points, const Attributes& attributes)
    : GBezier3d(points), Attributes(attributes)
{
    // Constructor for a Bezier surface with given control points
    // and given attributes.

    method = DIFFERENCES;
    tolerance = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
```

```
}


Bezier3d::Bezier3d(const Bezier3d& bezier)
{
    // Copy contstructor. See comments in Primitive3d.cc.

    *this = bezier;
}


void Bezier3d::convert_to_BRep_diff(BRep& brep) const
{
    // Converts the Bezier surface to a BRep, using forward differences.
    // No equivalent for the de Casteljau algorithm exists.

    // Functions to calculate differences of points and floats.
    void differences(const GPoint3d* const points, GPoint3d* const diff,
                     int no_points);
    void differences(const float* const array, float* const diff,
                     int no_elements);

    int i, j, k, r, c;   // Used in for loops below.
    float t, u;   // Parameter values for which we evaluate the surface.

    // Keep the elements of the surface handy.
    const int degree_t = deg_t();
    const int degree_u = deg_u();
    const int no_points_t = no_pts_t();
    const int no_points_u = no_pts_u();
    GPoint3dArray control_points(no_points_t, no_points_u);
    control_pts(control_points);

    const float h_t = eval_step_t;   // Just for abbreviation.
    const float h_u = eval_step_u;

    // Find number of points on the Bezier surface in each direction.
    int no_pts_t_bez = int((1.0 / h_t) + 0.5);   // Round up.
    int no_pts_u_bez = int((1.0 / h_u) + 0.5);

    // First work on each row.

    float* h_u_power = new float[no_points_u];   // All the powers of h, for u.
    h_u_power[0] = 1;
    for (j = 1; j < no_points_u; j++)
        h_u_power[j] = h_u * h_u_power[j-1];

    GPoint3dArray temp_pts(no_points_t, no_pts_u_bez);

    GPoint3d* row = new GPoint3d[no_points_u];
    GPoint3d* coeff_u = new GPoint3d[no_points_u];
    GPoint3d* df0_u = new GPoint3d[no_points_u];   // Array of differences.
    float* d0_u = new float[no_points_u];   // Array of delta^r of 0^i.

    for (r = 0; r < no_points_t; r++)  {

        control_points.get_row(r, row);

        // Calculate the coefficients of the monomial form for this row.

        differences(row, coeff_u, no_points_u);
        float j_factorial = 1;
        float n_k = 1;
        // coeff_u[0] already holds row[0] as needed.
        for (j = 1; j < no_points_u; j++)  {
            j_factorial = j_factorial * j;
            n_k = n_k * (degree_u - j + 1);

            coeff_u[j] =  n_k / j_factorial * coeff_u[j];
        }

        // Calculate delta^k of f[0].
        df0_u[0] = row[0];
        for (k = 1; k < no_points_u; k++)
            df0_u[k] = GPoint3d(0, 0, 0);   // Initialisation.


        for (i = 1; i < no_points_u; i++)  {
            for (j = 0; j <= i; j++)
                d0_u[j] = pow(j, i);
            differences(d0_u, d0_u, i+1);

            for (k = 1; k <= i; k++)
                df0_u[k] = GPoint3d( Vector3d(df0_u[k]) +
                                d0_u[k] * h_u_power[i] * Vector3d(coeff_u[i]) );
        }

        // Find points of the Bezier surface.
        c = 0;   // Column in the matrix where next point is to be stored.

        temp_pts.set_cell(r, c, row[0]);
        c++;

        for (u = 0; u < 1; u += h_u)  {   // u in [0, 1].
            for (k = 0; k < no_points_u-1; k++)
                // Calculate delta^k of f(t+h).
                df0_u[k] = GPoint3d(Vector3d(df0_u[k]) + Vector3d(df0_u[k+1]));

            temp_pts.set_cell(r, c, df0_u[0]);
            c++;
        }
    }

    delete[] h_u_power;
    delete[] row;
    delete[] coeff_u;
    delete[] df0_u;
    delete[] d0_u;
```

```
    // Now repeat for all the new columns.

    float h_t_power[no_points_t];   // All the powers of h, for parameter t.
    h_t_power[0] = 1;
    for (i = 1; i < no_points_t; i++)
        h_t_power[i] = h_t * h_t_power[i-1];

    GPoint3dArray bez_pts(no_pts_t_bez, no_pts_u_bez);

    GPoint3d* col = new GPoint3d[no_points_t];
    GPoint3d* coeff_t = new GPoint3d[no_points_t];
    GPoint3d* df0_t = new GPoint3d[no_points_t];   // Array of differences.
    float* d0_t = new float[no_points_t];   // Array of delta^r of 0^i.

    for (c = 0; c < no_pts_u_bez; c++)  {
        temp_pts.get_col(c, col);

        // Calculate the coefficients of the monomial form for this row.

        differences(col, coeff_t, no_points_t);
        float i_factorial = 1;
        float n_k = 1;
        // coeff_t[0] already holds col[0] as needed.
        for (i = 1; i < no_points_t; i++)  {
            i_factorial = i_factorial * i;
            n_k = n_k * (degree_t - i + 1);

            coeff_t[i] =  n_k / i_factorial * coeff_t[i];
        }

        // Calculate delta^k of f[0].
        df0_t[0] = col[0];
        for (k = 1; k < no_points_t; k++)
            df0_t[k] = GPoint3d(0, 0, 0);   // Initialisation.

        for (i = 1; i < no_points_t; i++)  {
            for (j = 0; j <= i; j++)
                d0_t[j] = pow(j, i);
            differences(d0_t, d0_t, i+1);

            for (k = 1; k <= i; k++)
                df0_t[k] = GPoint3d( Vector3d(df0_t[k]) +
                                d0_t[k] * h_t_power[i] * Vector3d(coeff_t[i]) );
        }

        // Find points of the Bezier surface.
        r = 0;   // Column in the matrix where next point is to be stored.

        bez_pts.set_cell(r, c, col[0]);
        r++;

        for (t = 0; t < 1; t += h_t)  {   // t in [0, 1].
            for (k = 0; k < no_points_t-1; k++)
                // Calculate delta^k of f(t+h).
                df0_t[k] = GPoint3d(Vector3d(df0_t[k]) + Vector3d(df0_t[k+1]));

            bez_pts.set_cell(r, c, df0_t[0]);
            r++;
        }
    }

    delete[] h_t_power;
    delete[] col;
    delete[] coeff_t;
    delete[] df0_t;
    delete[] d0_t;

    // Now form BRep.
    bez_pts.convert_to_BRep(brep);
    brep.set_attributes(*this);   // bez_pts did not hold any attributes.
}


void differences(const GPoint3d* const points, GPoint3d* const diff,
                 int no_points)
{
    // Calculates all forward differences
    // delta^r of points[0], r = 1,2,...,no_points-1.

    for (int j = 0; j < no_points; j++)
        diff[j] = points[j];

    for (int r = 1; r < no_points; r++)
        for (int i = no_points-1; i > r-1; i--)
            diff[i] = GPoint3d( Vector3d(diff[i]) - Vector3d(diff[i-1]) );
}


void differences(const float* const array, float* const diff,
                 int no_elements)
{
    // Calculates all forward differences
    // delta^r of array[0], r = 1,2,...,no_elements-1.

    for (int j = 0; j < no_elements; j++)
        diff[j] = array[j];

    for (int r = 1; r < no_elements; r++)
        for (int i = no_elements-1; i > r-1; i--)
            diff[i] = diff[i] - diff[i-1];
}


void Bezier3d::set_drawing_method(const Bezier_algorithm alg)
{
    // Sets the method for rendering the Bezier surface.
```

```
    method = alg;
}


void Bezier3d::set_tolerance(const float tol)
{
   // Sets the tolerance used for flatness tests in the de Casteljau algorithm.

   tolerance = tol;
}


void Bezier3d::set_eval_step_t(const float h_t)
{
   // Sets the evaluation step for parameter t used in the differences method.

   eval_step_t = h_t;
}


void Bezier3d::set_eval_step_u(const float h_u)
{
    // Sets the evaluation step for parameter t used in the differences method.

    eval_step_u = h_u;
}


void Bezier3d::transform(const Trans3d& trans)
{
    // Transformation function for Bezier surfaces.

    GBezier3d::transform(trans);
}


void Bezier3d::draw(View3d& view) const
{
    // Drawing of the Bezier surface as a simple wireframe.

    if ((no_pts_t() > 0)  &&  (no_pts_u() > 0))  {

        if ((no_pts_t() <= 2)  &&  (no_pts_u() <= 2))  {   // Just a polygon.
            GPoint3dArray points(no_pts_t(), no_pts_u());
            control_pts(points);

            BRep brep;
            points.convert_to_BRep(brep);
            brep.set_no_HSE(TRUE);   // No hidden surface elimination needed.
            brep.set_attributes(*this);

            Camera cam;   // A default camera, just to do the back surface cull.
            brep.back_surface_cull(cam);
            brep.sort_visible();
            brep.draw(view);
        }
        else  {
            if (method == DE_CASTELJAU)
                draw_decas(view);
            else if (method == DIFFERENCES)
                draw_diff(view);
            else
                cerr << "Unknown Bezier drawing method." << endl << endl;
        }
    }
}


void Bezier3d::draw(View3d& view, Camera& cam, Lighting_model& model) const
{
    // Rendering of the Bezier surface.

    if ((no_pts_t() > 0)  &&  (no_pts_u() > 0))  {

        if ((no_pts_t() <= 2)  &&  (no_pts_u() <= 2))  {   // Just a polygon.
            GPoint3dArray points(no_pts_t(), no_pts_u());
            control_pts(points);

            BRep brep;
            points.convert_to_BRep(brep);
            brep.set_attributes(*this);

            brep.back_surface_cull(cam);
            brep.sort_visible();
            brep.draw(view);
        }
        else  {
            if (method == DE_CASTELJAU)  {
                cerr << "Shading of Bezier surfaces only possible using the "
                     << "differences method." << endl << endl;
                draw_diff(view, cam, model);
            }
            else if (method == DIFFERENCES)
                draw_diff(view, cam, model);
            else
                cerr << "Unknown Bezier drawing method." << endl << endl;
        }
    }
}


void Bezier3d::draw_decas(View3d& view) const
{
    // Converts the Bezier surface to a GPolyPoint3d using the de Casteljau
    // algorithm. The points are added to gpp.
    // This will be then converted to a BRep for rendering.

    const float t = 0.5;   // Parameter values for subdivision.
```

```
    const float u = 0.5;

    if (is_flat(tolerance))  {
        GPoint3dArray control_points(no_pts_t(), no_pts_u());
        control_pts(control_points);
        GPoint3d vert[4];

        control_points.get_cell(0, 0, vert[0]);
        control_points.get_cell(0, no_pts_u() - 1, vert[1]);
        control_points.get_cell(no_pts_t() - 1, no_pts_u() - 1, vert[2]);
        control_points.get_cell(no_pts_t() - 1, 0, vert[3]);
        Polygon3d new_pts(4, vert, *this);   // *this for the attributes.
        new_pts.draw(view);
    }
    else  {
        Bezier3d top_left(*this),
                 top_right(*this),
                 bottom_left(*this),
                 bottom_right(*this);

        subdivide(t, u, top_left, top_right, bottom_left, bottom_right);

        top_left.draw_decas(view);
        top_right.draw_decas(view);
        bottom_left.draw_decas(view);
        bottom_right.draw_decas(view);
    }
}


void Bezier3d::draw_diff(View3d& view) const
{
    // Draws the Bezier surface as a wireframe, by displaying the according BRep

    Camera cam;   // A default camera, just for the back surface cull.

    BRep brep;
    convert_to_BRep_diff(brep);
    brep.set_no_HSE(TRUE);   // Do not perform hidden surface elimination.
    brep.back_surface_cull(cam);
    brep.sort_visible();
    brep.draw(view);
}


void Bezier3d::draw_diff(View3d& view, Camera& cam, Lighting_model& model) const
{
    // Renders the Bezier surface by displaying the according BRep.

    BRep brep;
    convert_to_BRep_diff(brep);
    brep.back_surface_cull(cam);
    brep.sort_visible();
    brep.draw(view, cam, model);
}


void Bezier3d::draw_transformed(View3d& view, const Trans3d& trans) const
{
    // Draws the Bezier surface transformed, without affecting it.

    Bezier3d bezier = *this;
    bezier.transform(trans);
    bezier.draw(view);
}


/*****************************************************************************/


BSpline3d::BSpline3d()
{
    // Constructor for a null B-Spline surface.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(const GBSpline3d& bspline)
    : GBSpline3d(bspline)
{
    // Constructor for a B-Spline surface given as a GBSpline3d,
    // with no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(const Attributes& attributes)
    : Attributes(attributes)
{
    // Constructor for a null B-Spline surface with attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
```

```
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(const GBSpline3d& bspline, const Attributes& attributes)
    : GBSpline3d(bspline), Attributes(attributes)
{
    // Constructor for a B-Spline surface given as a GBSpline3d,
    // and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points)
    : GBSpline3d(deg_t, deg_u, points)
{
    // Constructor for a B-Spline surface with given control points,
    // default knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                     const Attributes& attributes)
    : GBSpline3d(deg_t, deg_u, points), Attributes(attributes)
{
    // Constructor for a B-Spline surface with given control points,
    // default knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                     const float* knot_seq_t, const float* knot_seq_u)
    : GBSpline3d(deg_t, deg_u, points, knot_seq_t, knot_seq_u)
{
    // Constructor for a B-Spline surface with given control points,
    // default knot sequence and no attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(int deg_t, int deg_u, const GPoint3dArray& points,
                     const float* knot_seq_t, const float* knot_seq_u,
                     const Attributes& attributes)
    : GBSpline3d(deg_t, deg_u, points, knot_seq_t, knot_seq_u),
      Attributes(attributes)
{
    // Constructor for a B-Spline surface with given control points,
    // default knot sequence and attributes specified.

    method = DE_BOOR;
    Bezier_method = DIFFERENCES;
    decas_tolerance = 0.05;
    diff_eval_step_t = 0.05;
    diff_eval_step_u = 0.05;
    eval_step_t = 0.05;
    eval_step_u = 0.05;
}


BSpline3d::BSpline3d(const BSpline3d& bspline)
{
    // Copy constructor for B-Spline surfaces.
    // See comments in Primitive3d.cc

    *this = bspline;
}


void BSpline3d::convert_to_BRep_deboor(BRep& brep) const
{
    // Converts a B-Spline surface of any degree to a BRep
    // using the de Boor formula.
    // No equivalent for conversion using the Bezier control points.

    // This function calculates the points on the B-Spline.
```

```
// 'index' is the index of the lower end of the interval.
GPoint3d deBoor_point(GPoint3d* points, float* knots, int degree,
                      float t, int index);

// Get the elements of the surface.
const int degree_t = deg_t();
const int degree_u = deg_u();
const int no_points_t = no_pts_t();
const int no_points_u = no_pts_u();
const int no_knots_t = no_knts_t();
const int no_knots_u = no_knts_u();
float* knots_t = new float[no_knots_t];
knot_sequence_t(knots_t);
float* knots_u = new float[no_knots_u];
knot_sequence_u(knots_u);
GPoint3dArray points(no_points_t, no_points_u);
control_pts(points);

int i, j, r, c;   // Used in for loops below.
float t, u;   // The parameter values for which we calculate the point.

// The surface is only defined in the intervals starting at t[i],
// with i = degree-1, ..., no_points-2.

// Transform h to match the actual ranges of t and u.
const float h_t = eval_step_t *
                            (knots_t[no_points_t-1] - knots_t[degree_t-1]);
const float h_u = eval_step_u *
                            (knots_u[no_points_u-1] - knots_u[degree_u-1]);

// Calculate number of points in each direction.
// Essentially run through all parameter values once, just to enumerate them
int no_pts_u_bspl = 1;   // Count the last point, which is added directly.
for (j = degree_u - 1; j < no_points_u - 1; j++)
    for (u = knots_u[j]; u < knots_u[j+1]; u += h_u)
        no_pts_u_bspl++;

int no_pts_t_bspl = 1;   // Count the last point, which is added directly.
for (i = degree_t - 1; i < no_points_t - 1; i++)
    for (t = knots_t[i]; t < knots_t[i+1]; t += h_t)
        no_pts_t_bspl++;

// First work on each row.
GPoint3d* row = new GPoint3d[no_points_u];
GPoint3dArray temp_pts(no_points_t, no_pts_u_bspl);
for (r = 0; r < no_points_t; r++)  {
    points.get_row(r, row);

    c = 0;   // Column where next point is to be stored in the array.
    for (j = degree_u - 1; j < no_points_u - 1; j++)
        for (u = knots_u[j]; u < knots_u[j+1]; u += h_u)  {
            GPoint3d pt = deBoor_point(row, knots_u, degree_u, u, j);
            temp_pts.set_cell(r, c, pt);
            c++;
        }
    // Add the last point.
    GPoint3d pt = deBoor_point(row, knots_u, degree_u,
                               knots_u[no_points_u-1], no_points_u-2);
    temp_pts.set_cell(r, c, pt);
}
delete[] row;

// Then work on each of the new columns.
GPoint3d* col = new GPoint3d[no_points_t];
GPoint3dArray bspl_pts(no_pts_t_bspl, no_pts_u_bspl);
for (c = 0; c < no_pts_u_bspl; c++)  {
    temp_pts.get_col(c, col);

    r = 0;   // Row where next point is to be stored in the array.
    for (i = degree_t - 1; i < no_points_t - 1; i++)
        for (t = knots_t[i]; t < knots_t[i+1]; t += h_t)  {
            GPoint3d pt = deBoor_point(col, knots_t, degree_t, t, i);
            bspl_pts.set_cell(r, c, pt);
            r++;
        }
    // Add the last point.
    GPoint3d pt = deBoor_point(col, knots_t, degree_t,
                               knots_t[no_points_t-1], no_points_t-2);
    bspl_pts.set_cell(r, c, pt);

}
delete[] col;

delete[] knots_t;
delete[] knots_u;

// Now form BRep.
bspl_pts.convert_to_BRep(brep);
brep.set_attributes(*this);   // bspl_pts did not hold any attributes.
}


GPoint3d deBoor_point(GPoint3d* points, float* knots, int degree,
                      float t, int index)
{
    // Calculates the point corresponding to parameter value t,
    // t in [ knots[index], knots[index+1] ],
    // with the control points and knot sequence given,
    // using the de Boor algorithm.

    GPoint3d* pts = new GPoint3d[degree + 1];   // de Boor points.
    for (int j = 0; j < degree + 1; j++)
        pts[j] = points[index - j + 1];

    for (int s = 0; s < degree; s++)
        for (int i = degree; i > s; i--)  {
            const float coeff = (knots[degree+1+index-i] - t) /
                                (knots[degree+1+index-i] - knots[s+1+index-i]);
```

```
            pts[i] = coeff*pts[i] + (1-coeff)*pts[i-1];
        }

    GPoint3d pt = pts[degree];
    delete[] pts;
    return pt;
}


void BSpline3d::set_drawing_method(BSpline_algorithm alg)
{
    // Sets the drawing method for the B-Spline surface.

    method = alg;
}


void BSpline3d::set_Bezier_drawing_method(Bezier_algorithm alg)
{
    // Sets the drawing method for the Bezier surface segments.

    Bezier_method = alg;
}


void BSpline3d::set_decas_tolerance(const float tol)
{
    // Sets the tolerance for the de Castlejau algorithm when applied to
    // the Bezier segments.

    decas_tolerance = tol;
}


void BSpline3d::set_diff_eval_step_t(const float h_t)
{
    // Sets the evaluation step for parameter t for the differences method
    // when applied to the Bezier curve segments.

    diff_eval_step_t = h_t;
}


void BSpline3d::set_diff_eval_step_u(const float h_u)
{
    // Sets the evaluation step for parameter u for the differences method
    // when applied to the Bezier curve segments.

    diff_eval_step_u = h_u;
}


void BSpline3d::set_eval_step_t(const float h_t)
{
    // Sets the evaluation step for parameter t for the de Boor algorithm.

    eval_step_t = h_t;
}


void BSpline3d::set_eval_step_u(const float h_u)
{
    // Sets the evaluation step for parameter t for the de Boor algorithm.

    eval_step_u = h_u;
}


void BSpline3d::transform(const Trans3d& trans)
{
    // Transformation function for B-Spline surfaces.

    GBSpline3d::transform(trans);
}


void BSpline3d::draw(View3d& view) const
{
    // Drawing of the B-Spline surface as a simple wireframe.

    if ((no_pts_t() > 0)  &&  (deg_t() > 0)  &&
        (no_pts_u() > 0)  &&  (deg_u() > 0))  {

        // Checks for knots of multiplicity mult, from knots[j1] to knots[j2].
        int multiple_knots(const float* knots, int j1, int j2, int mult);

        float* knots_t = new float[no_knts_t()];
        knot_sequence_t(knots_t);
        float* knots_u = new float[no_knts_u()];
        knot_sequence_u(knots_u);

        if (multiple_knots(knots_t, 0, no_knts_t() - 1, deg_t() + 1)  ||
            multiple_knots(knots_u, 0, no_knts_u() - 1, deg_u() + 1))
            cerr << "B-Splines having knots with multiplicity higher than"
         << endl << "their degree cannot be rendered. Procedure aborted."
         << endl << endl;

        else  {
            if (method == AS_BEZIER)  {

                if ((deg_t() > 3)  ||  (deg_u() > 3))  {
                    cerr << "Rendering of B-Spline surfaces as piecewise Bezier"
                 << endl << "surfaces only applicable for quadratic or cubic"
                 << endl << "B-Splines. Using the de Boor method instead."
                 << endl << endl;
                    draw_deboor(view);
                }
```

```
                else if ( multiple_knots(knots_t, 1, no_knts_t()-2, deg_t())  ||
                          multiple_knots(knots_u, 1, no_knts_u()-2, deg_u()) ) {
                    cerr << "Rendering of B-Spline surfaces as piecewise Bezier"
                         << " surfaces not applicable to" << endl
                         << "surfaces having knots with multiplicity equal to"
                         << " their degree, other than the" << endl
                         << "end knots. Using the de Boor method instead."
                         << endl << endl;
                    draw_deboor(view);
                }

                else
                    draw_bez(view);
            }
            else if (method == DE_BOOR)
                draw_deboor(view);
            else
                cerr << "Unknown B-Spline drawing method." << endl << endl;
        }

        delete[] knots_t;
        delete[] knots_u;
    }
}


void BSpline3d::draw(View3d& view, Camera& cam, Lighting_model& model) const
{
    // Rendering of the B-Spline surface.

    if ((no_pts_t() > 0)  &&  (deg_t() > 0)  &&
        (no_pts_u() > 0)  &&  (deg_u() > 0))  {

        // Checks for knots of multiplicity mult, from knots[j1] to knots[j2].
        int multiple_knots(const float* knots, int j1, int j2, int mult);

        float* knots_t = new float[no_knts_t()];
        knot_sequence_t(knots_t);
        float* knots_u = new float[no_knts_u()];
        knot_sequence_u(knots_u);

        if (multiple_knots(knots_t, 0, no_knts_t() - 1, deg_t() + 1)  ||
            multiple_knots(knots_u, 0, no_knts_u() - 1, deg_u() + 1))
            cerr << "B-Splines having knots with multiplicity higher than"
             << endl << "their degree cannot be rendered. Procedure aborted."
             << endl << endl;

        else  {
            if (method == AS_BEZIER)  {
                cerr << "Shading of B-Spline surfaces only possible using the "
                     << "de Boor method." << endl << endl;
                draw_deboor(view, cam, model);

            }
            else if (method == DE_BOOR)
                draw_deboor(view, cam, model);
            else
                cerr << "Unknown B-Spline drawing method." << endl << endl;
        }

        delete[] knots_t;
        delete[] knots_u;
    }
}


int multiple_knots(const float* knots, int j1, int j2, int mult)
{
    // Checks for knots of multiplicity mult, from knots[j1] to knots[j2].

    int i = j1;
    while (i <= j2-mult+1)  {
        const float model = knots[i];
        int count = 1;
        i++;
        while ((i <= j2)  &&  (knots[i] == model)  &&  (count < mult))  {
          count++;
          i++;
        }
        if (count == mult)
            return 1;
     }
     return 0;
}


void BSpline3d::draw_bez(View3d& view) const
{
    // Draws quadratic or cubic B-Splines to BRep by calculating the
    // corresponding control points for the Bezier surface segments.

    // The functions that produce the Bezier control points from the given
    // B-Spline control points.
    void bspline_to_bezier_quadratic(const GPoint3d* bspl_points, float* knots,
                                     int i, GPoint3d* bez_points);
    void bspline_to_bezier_cubic(const GPoint3d* bspl_points, float* knots,
                                 int i, GPoint3d* bez_points);

    // Get the elements of the surface handy.
    const int degree_t = deg_t();
    const int degree_u = deg_u();
    const int no_points_t = no_pts_t();
    const int no_points_u = no_pts_u();
    GPoint3dArray points(no_points_t, no_points_u);
    control_pts(points);
    const int no_knots_t = no_knts_t();
    float* knots_t = new float[no_knots_t];
    knot_sequence_t(knots_t);
```

```
    const int no_knots_u = no_knts_u();
    float* knots_u = new float[no_knots_u];
    knot_sequence_u(knots_u);

    // Some auxiliary arrays of points.
    GPoint3d* row = new GPoint3d[no_points_u];
    GPoint3d* col = new GPoint3d[no_points_t];
    GPoint3d* bez_row = new GPoint3d[degree_u + 1];
    GPoint3d* bez_col = new GPoint3d[degree_t + 1];

    // First interpolate on the rows, for all possible intervals of u.
    for (int j = degree_u - 1; j < no_points_u - 1; j++)  {
        GPoint3dArray temp_pts(no_points_t, degree_u + 1);
        for (int r = 0; r < no_points_t; r++)  {
            points.get_row(r, row);
            if (degree_u == 2)
                bspline_to_bezier_quadratic(row, knots_u, j, bez_row);
            else
                bspline_to_bezier_cubic(row, knots_u, j, bez_row);
            temp_pts.set_row(r, bez_row);
        }

        // Then interpolate on the columns, for all possible intervals of t.
        for (int i = degree_t - 1; i < no_points_t - 1; i++)  {
            GPoint3dArray bez_pts(degree_t + 1, degree_u + 1);
            for (int c = 0; c < degree_u + 1; c++)  {
                temp_pts.get_col(c, col);
                if (degree_t == 2)
                    bspline_to_bezier_quadratic(col, knots_t, i, bez_col);
                else
                    bspline_to_bezier_cubic(col, knots_t, i, bez_col);
                bez_pts.set_col(c, bez_col);
            }

            // Now draw the Bezier sub-surface.
            Bezier3d bezier(bez_pts, *this);   // *this for the attributes.
            bezier.set_drawing_method(Bezier_method);
            bezier.set_tolerance(decas_tolerance);
            bezier.set_eval_step_t(diff_eval_step_t);
            bezier.set_eval_step_u(diff_eval_step_u);
            bezier.draw(view);
        }
    }

    delete[] knots_t;
    delete[] knots_u;
    delete[] row;
    delete[] col;
    delete[] bez_row;
    delete[] bez_col;
}


void bspline_to_bezier_quadratic(const GPoint3d* bspl_points, float* knots,
                                 int i, GPoint3d* bez_points)
{
    // Produces the corresponding Bezier control points for the given B-Spline
    // control points in the interval [ t[i], t[i+1] ] for the quadratic case.

    float coeff;   // Used for interpolations below.

    coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-1]);
    bez_points[0] = coeff*bspl_points[i-1] + (1-coeff)*bspl_points[i];

    bez_points[1] = bspl_points[i];

    coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i]);
    bez_points[2] = coeff*bspl_points[i] + (1-coeff)*bspl_points[i+1];
}


void bspline_to_bezier_cubic(const GPoint3d* bspl_points, float* knots,
                             int i, GPoint3d* bez_points)
{
    // Produces the corresponding Bezier control points for the given B-Spline
    // control points in the interval [ t[i], t[i+1] ] for the cubic case.

    float coeff;   // Used for interpolations below.

    // First step.
    coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-2]);
    bez_points[0] = coeff*bspl_points[i-2] + (1-coeff)*bspl_points[i-1];

    coeff = (knots[i+2] - knots[i]) / (knots[i+2] - knots[i-1]);
    bez_points[1] = coeff*bspl_points[i-1] + (1-coeff)*bspl_points[i];

    coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i-1]);
    bez_points[2] = coeff*bspl_points[i-1] + (1-coeff)*bspl_points[i];

    coeff = (knots[i+3] - knots[i+1]) / (knots[i+3] - knots[i]);
    bez_points[3] = coeff*bspl_points[i] + (1-coeff)*bspl_points[i+1];

    // Second step.
    coeff = (knots[i+1] - knots[i]) / (knots[i+1] - knots[i-1]);
    bez_points[0] = coeff*bez_points[0] + (1-coeff)*bez_points[1];

    coeff = (knots[i+2] - knots[i+1]) / (knots[i+2] - knots[i]);
    bez_points[3] = coeff*bez_points[2] + (1-coeff)*bez_points[3];
}


void BSpline3d::draw_deboor(View3d& view) const
{
    // Draws the B-Spline surface as a wireframe,
    // by displaying the according BRep.

    Camera cam;   // A default camera, just for the back surface cull.
```

```
    BRep brep;
    convert_to_BRep_deboor(brep);
    brep.set_no_HSE(TRUE);   // Do not perform hidden surface elimination.
    brep.back_surface_cull(cam);
    brep.sort_visible();
    brep.draw(view);
}


void BSpline3d::draw_deboor(View3d& view, Camera& cam,
                            Lighting_model& model) const
{
    // Renders the B-Spline surface by displaying the according BRep.

    BRep brep;
    convert_to_BRep_deboor(brep);
    brep.back_surface_cull(cam);
    brep.sort_visible();
    brep.draw(view, cam, model);
}


void BSpline3d::draw_transformed(View3d& view, const Trans3d& trans) const
{
    // Draws the B-Spline surface transformed, without affecting it.

    BSpline3d bspline(*this);
    bspline.transform(trans);
    bspline.draw(view);
}
```

APPENDIX D: THE TEAPOT PROGRAM

The program presented in the following pages draws the teapot shown in the examples of Chapter 3. The teapot consists of 32 individual bi-cubic Bézier surfaces, given essentially in their boundary representation: One file (`teapot.pts`) contains the coordinates of all the control points of these Bézier surfaces, and another (`teapot.vrt`) contains references to the 16 control points of each surface. The program reads all the points in memory, and then reads from the second file the references to the control points and constructs each Bézier surface in turn. Before displaying them, it sorts them from back to front with respect to the camera so that they are drawn in the correct order.

The program has a number of inputs for greater flexibility. These inputs can conveniently be provided in a text file which is piped to the program when it is executed. In turn, the program accepts:

- The drawing method ('1' for de Casteljau or '2' for differences).
- For the method chosen, the relevant drawing parameters (tolerance for the de Casteljau algorithm and evaluation steps for t and u separately for the differences method).
- The drawing action for the polygons displayed ('1' for wireframe, '2' for filled polygons with the sides displayed, '3' for filled polygons with no sides displayed).
- The angles 'phi' and 'theta' determining the position of the camera in polar coordinates.
- The number of lights, and for each of them, its position in space and its intensity.

```
#include "project3d.h"   // A header file including "graphics3d.h" and the new
                         // classes necessary.
#include "BRep.h"
#include <math.h>
#include <iostream.h>
#include <fstream.h>


int main()
{
    // The selection sort procedure used to sort the Bezier surfaces,
    // to ensure correct order of drawing.
    void sel_sort(const float* dist, int* look_up, int start, int end);

    // Functions to set up the colour table.
    float red(const float x);
    float green(const float x);
    float blue(const float x);

    // Input drawing parameters.
    Bezier_algorithm method = DIFFERENCES;
    float tolerance = 0.05;
    float h_t = 0.1;
    float h_u = 0.1;
    AreaAction action = EdgeArea;

    cerr << "Choose the drawing method for the teapot: " << endl
         << "    1 - De Casteljau" << endl
         << "    2 - Differences" << endl;
    int choice;
    cin >> choice;

    if (choice == 1)  {
        method = DE_CASTELJAU;
        cerr << "Enter tolerance: " << endl;
        cin >> tolerance;
    }
    else if (choice == 2)  {
        method = DIFFERENCES;
        cerr << "Enter evaluation step for t: " << endl;
        cin >> h_t;
        cerr << "Enter evaluation step for u: " << endl;
        cin >> h_u;
    }

    cerr << "Choose drawing mode:" << endl
         << "    1 - EdgeArea" << endl
         << "    2 - EdgeAndFillArea" << endl
         << "    3 - FillArea" << endl;
    cin >> choice;

    if (choice == 1)
        action = EdgeArea;
    else if (choice == 2)
        action = EdgeAndFillArea;
    else if (choice == 3)
        action = FillArea;

    Attributes a;   // Default attributes used for all Bezier surfaces.
    int i;   // Used in for loops below.

    // Read the teapot.

    ifstream pts_file("teapot.pts");
    ifstream vert_file("teapot.vrt");

    // Read all the points and store them in an array.
    cerr << "Reading points...   ";
    int total_pts;
    pts_file >> total_pts;
    GPoint3d* all_pts = new GPoint3d[total_pts];
    for (i = 0; i < total_pts; i++)
        pts_file >> all_pts[i];
    cerr << total_pts << " points read." << endl;

    // Read each GPoint3dArray and construct the corresponding Bezier surface.
    cerr << "Constructing Bezier surfaces...   ";
    int no_bezier;
    vert_file >> no_bezier;
    Bezier3d* bezier = new Bezier3d[no_bezier];
    for (i = 0; i < no_bezier; i++)  {
        int nr, nc;   // Number of rows and columns of the array to be read.
        vert_file >> nr >> nc;

        GPoint3dArray bez_pts(nr, nc);
        for (int r = 0; r < nr; r++)
            for (int c = 0; c < nc; c++)  {
                int index;
                vert_file >> index;
                index -= 1;   // Indices in the file start from 1.
                bez_pts.set_cell(r, c, all_pts[index]);
            }

        bezier[i] = Bezier3d(bez_pts, a);
        bezier[i].set_drawing_method(method);
        bezier[i].set_area_action(action);
        bezier[i].set_tolerance(tolerance);
        bezier[i].set_eval_step_t(h_t);
        bezier[i].set_eval_step_u(h_u);
    }
    cerr << no_bezier << " surfaces constructed." << endl;
```

```
// Find the bounding box of the teapot.
float xmin = all_pts[0].x(), xmax = all_pts[0].x(),
      ymin = all_pts[0].y(), ymax = all_pts[0].y(),
      zmin = all_pts[0].z(), zmax = all_pts[0].z();

for (i = 1; i < total_pts; i++)  {
    const float current_x = all_pts[i].x();
    if (current_x < xmin)
        xmin = current_x;
    else if (current_x > xmax)
        xmax = current_x;

    const float current_y = all_pts[i].x();
    if (current_y < ymin)
        ymin = current_y;
    else if (current_y > ymax)
        ymax = current_y;

    const float current_z = all_pts[i].z();
    if (current_z < zmin)
        zmin = current_z;
    else if (current_z > zmax)
        zmax = current_z;
}

Box3d teapot_box(xmin, ymin, zmin, xmax, ymax, zmax);

// Input and set the camera and view.
Vector3d diag = Vector3d(teapot_box.max()) - Vector3d(teapot_box.min());

const float radius = diag.length() / 2.0;
float scale = 30;
float phi = 0.0;
float theta = -1.5;
cerr << "Enter camera angles phi and theta: ";
cin >> scale >> phi >> theta;

GPoint3d cen = GPoint3d( Vector3d(teapot_box.max()) -
                         Vector3d(teapot_box.min()) );
cerr << "Centre:  " << cen << endl;
float cx = cen.x() + radius*scale*cos(phi)*cos(theta);
float cy = cen.y() + radius*scale*cos(phi)*sin(theta);
float cz = cen.z() + radius* 5;
GPoint3d ref(cx, cy, cz);
cerr << "Camera reference point: " << ref << endl;

Camera camera;
camera.set_ref_pt(ref);
camera.set_up(0, 0.2, 1.2);
camera.set_normal(Vector3d(0, 0, 0) - Vector3d(ref));
camera.set_proj_type(Perspective);
camera.set_CofP(0, 0, -10);

View3d view(camera);
view.set_window(-3, -3, -3, 3, 3, 3);

// Set colour Table.
set_colour_table(red, green, blue);

// Read the lights and set lighting model.
int no_lights = 1;
cerr << "Enter number of lights: " << endl;
cin >> no_lights;
Light* light = new Light[no_lights];

light[0].set_position(Vector3d(-5, -5, 5));
light[0].set_direction(Vector3d(-5, -5, 5));
light[0].set_I_P(1);

for (i = 0; i < no_lights; i++)  {
    GPoint3d light_pos;
    cerr << "Enter position of light " << i << ": " << endl;
    cin >> light_pos;
    float I_P;
    cerr << "Enter intensity of light " << i << ": " << endl;
    cin >> I_P;

    light[i].set_position(light_pos);
    light[i].set_direction(light_pos);
    light[i].set_I_P(I_P);
}

Lighting_model light_model;
light_model.set_light_array(light, no_lights);
light_model.set_I_A(0.5);

// Sort objects by diminishing distance from the camera.

// Calculate distances from camera.
float* dist = new float[no_bezier];
for (i = 0; i < no_bezier; i++)
    dist[i] = Vector3d(Vector3d(camera.get_ref_pt()) -
              Vector3d(bezier[i].centre())).length();

// Initialise look-up table.
int* look_up = new int[no_bezier];
for (i = 0; i < no_bezier; i++)
    look_up[i] = i;

// Sort all objects.
sel_sort(dist, look_up, 0, no_bezier-1);
```

```
    // Draw all objects.
    if (action == EdgeArea)
        for (i = 0; i < no_bezier; i++)
            bezier[look_up[i]].draw(view);
    else
        for (i = 0; i < no_bezier; i++)
            bezier[look_up[i]].draw(view, camera, light_model);


    delete[] all_pts;
    delete[] bezier;
    delete[] light;
    delete[] dist;
    delete[] look_up;
}


void sel_sort(const float* dist, int* look_up, int start, int end)
{
    // Selection sort methodorithm.
    // A look-up table is used to hold the order of the objects.

    void swap(int& x, int& y);

    for (int i = start; i < end; i++)  {
        int max = i;
        for (int j = i+1; j < end+1; j++)
            if (dist[look_up[j]] > dist[look_up[max]])
                max = j;
        swap(look_up[max], look_up[i]);
    }
}


void swap(int& x, int& y)
{
    // Swaps the two integers.

    const int temp = x;
    x = y;
    y = temp;
}


float red(const float x)
{
    return x;
}


float green(const float x)
{
    return x;
}


float blue(const float x)
{
    return x;
}
```