CHAPTER
THIRTEEN
B-spline Programming Concepts

13.1 Introduction

In the previous twelve chapters we introduced different aspects of modeling with NURBS. These chapters covered basically two things: (1) the underlying mathematics, and (2) algorithms illustrating how the tools can be turned into computer code. Although the pseudocodes presented in each chapter are fairly detailed, containing all the necessary information to turn them into workable routines, it is a long way from the algorithmic sketch to a useful system. How to design a useful system is a very difficult question (contrary to many researchers who consider it as a minor "implementation issue"). A NURBS system has to satisfy many requirements in order to be useful in a complex surface or solid modeler. In this chapter we introduce a number of programming concepts that, we believe, contribute to good programming practices. We are fully aware of the diversity of programming styles and conventions. Consequently, we do not attempt to present the *best* or the *ultimate* solution simply because there is none.

During the course of writing this book we developed a comprehensive library, *Nlib V2.0*, implemented in ANSI C. This library contains numerous routines necessary to implement every tool discussed in the book. In this chapter we present the philosophy, architecture, and organization of that library. While developing *Nlib V2.0* we required our routines to adhere to the following general standards [see Stra92]:

- *toolability*: available tools should be used to build new routines;
- *portability*: it should be easy to port the software to different software and hardware platforms;
- *reusability*: the program should be written in such a way as to be able to reuse sections of the code;
- *testability*: the code should be made consistent and simple so that testing and debugging become easy;
- *reliability*: errors should be handled consistently and gracefully;

- *enhanceability*: the code must be easily understood so that new features can be added without significant effort;
- *fixability*: it must be reasonably easy to locate bugs;
- *consistency*: programming conventions should be consistent throughout the library;
- *communicability*: it must be easy to read and to understand the program.

We believe that these standards are necessary to minimize what is commonly referred to as "the nightmare of software maintenance". A common disease in the software industry is "software trashing", i.e., when one employee leaves the company his code gets trashed because nobody understands it, hence maintenance becomes impossible.

In addition to these general standards, we followed some specific standards:

- *style of programming*: we wanted the code to look very much like the mathematics in the book so that the reader has little trouble following the various steps; in addition, we followed a "user friendly" programming style as suggested in [Baly94];
- *usability*: since NURBS are very popular in many application areas, it should be easy for nonexperts to use the library to build higher level applications of various types;
- *numerical efficiency*: one of the reasons why NURBS were chosen in many standards is because of their excellent numerical properties; NURBS evaluations are mixed with other numerical code which we carefully wrote so as not to destroy the overall numerical efficiency;
- *object based programming*: programming curves and surfaces requires dealing with large amounts of data; to avoid carrying this data around and to enhance understanding the code, we grouped various entities into objects.

A word on Object Based Programming (OBP): When dealing with OBP, many people automatically think of languages such as C++. We did not implement *Nlib V2.0* in C++ simply because plain C does just fine, allowing both the C and the C++ programmer to use the library routines.

In the sections that follow we deal with issues of (1) data types and portability, (2) data structures, (3) memory allocation, (4) error handling, (5) utility routines, (6) arithmetic operations, (7) example programs, (8) some additional structures, and (9) system structure. These issues are often more complicated than the mathematics of NURBS. The mathematics is basically right or wrong; software is a large gray area with no boundaries.

13.2 Data Types and Portability

In this section we deal with the following portability problems:

- data type portability, e.g., one system represents integers as 32 bit numbers whereas the other uses only 16 bits;

- operations portability, e.g., arithmetic operations yield different results on different machines;
- domain portability, e.g., one system is a drafting system performing operations mainly in two-dimensional space, whereas the other is a design system requiring computations in three-dimensional space.

We deal with these issues in three ways: (1) using predefined data types and avoiding the explicit use of compiler defined data types, (2) introducing special arithmetic functions that take care of all arithmetic operations in all of the two-dimensional, three-dimensional, or four-dimensional domains (rational or nonrational, two-dimensional or three-dimensional), and (3) defining objects that are dimensionless. To fix portability problems one has to change only a few lines of code in one of the include files, or fix a few arithmetic routines, or redefine some of the objects. The rest of the system, which can contain millions of lines of code, remains intact.

In *Nlib V2.0* we use several predefined data types. Some examples are:

```
typedef int INDEX;
typedef short FLAG;
typedef int INTEGER;
typedef double REAL;
typedef char * STRING;
typedef short DEGREE;
typedef double PARAMETER;
```

The `INDEX` data type is used to declare variables that are used as the highest indexes of arrays of different dimensions. For example, if the highest index never exceeds 32,767 (16 bit), then the current definition is adequate. However, if arrays of larger dimensions are required, then the definition

```
typedef long INDEX;
```

must be used. Similar arguments hold for the `FLAG` data type (for example, used to "flag" or select different cases in a switch statement) and for the `INTEGER` data type. If a system uses only 8 bit integers, then the only change required is to make `INTEGER short` as opposed to `int`. The `REAL` data type is defined by default as `double`, which can be changed to `float` if single precision is satisfactory or the rest of the system uses single precision. The `STRING` data type is introduced for convenience and for conceptual reasons.

The explanation behind the `DEGREE` and the `PARAMETER` data types is similar to the `INDEX` data type, that is, the degree of a curve or surface is much lower than the integers used as variables. Consequently, a `short` integer is quite adequate. Defining `DEGREE` as a new data type also signals the fact that it is *different* from ordinary integers; it belongs to the parameters that collectively define a curve or a surface. The same holds for the `PARAMETER` type, which can be either `double` or `float`, depending on the application or on the current system's floating point computation.

One of the most difficult tasks in floating point calculations is to deal with numerical imprecision. It is almost impossible not to use tolerances for point coincidence or to measure the equality of two floating point numbers. In *Nlib V2.0* we use the following constants defined by the compiler:

```
#define BIGD DBL_MAX
#define SMAD DBL_MIN
#define BIGI INT_MAX
#define SMAI INT_MIN
#define DEPS DBL_EPSILON
```

BIGD and SMAD are the largest and the smallest double precision numbers, respectively. The ANSI C standard specifies that they must be defined in float.h. It also requires BIGD to be at least 10^{37} or greater and SMAD to be at least 10^{-37} or smaller. The requirement for the double precision epsilon DEPS is at least 10^{-9} or smaller.

Limits for integer variables are stored in limits.h. Some typical ranges are $-32,768$ to $32,767$ for int, and $-2,147,483,648$ to $2,147,483,647$ for long. The range for the unsigned long is up to 4,294,967,295. Working with these constants ensures the consistency of computations no matter which hardware platform one uses.

Operations and domain portabilities are ensured through special arithmetic functions operating on objects. Examples are given in subsequent sections.

13.3 Data Structures

As mentioned in Section 13.1, we intend to use object based concepts and to follow the mathematical notation and concepts of previous chapters. As an example, in the curve evaluation routine we write

```
FLAG N_evncur( CURVE *cur, PARAMETER u, POINT *C )
```

or, given a CURVE and a PARAMETER, compute a POINT on the CURVE at the given PARAMETER value. The nice thing about this type of programming is that the user need not know anything about how the different entities (curve, parameter, and point) are represented internally. They are considered as objects built from smaller components. To define a curve (or shall we say, to define a curve *object*), we need to store these entities:

- n: the highest index of control points;
- $\mathbf{P}_0^w, \ldots, \mathbf{P}_n^w$: the control points;
- p: the degree;
- m: the highest index of knots;
- $u_0, \ldots, u_m$: the knots.

Although this data can be passed directly to every NURBS routine, it is fairly inconvenient and unnatural to the programmer, who thinks at different levels of

abstraction. For example, when dealing with curves for interactive design, he works with them without regard to the number of control points and knots. Or, for example, when dealing with curve/curve intersection, he might descend his thinking to the level of the control polygon if the overlapping of bounding boxes is what needs to be examined. For these and many other similar reasons, *Nlib V2.0* considers every geometric entity as an object built from simpler components.

In the design of a huge structure, simple building blocks are created first from basic data types. One of the basic elements of a curve object is the control point. It can be rational or nonrational, two-dimensional or three-dimensional, that is, it needs to hold two, three, or four coordinates. In addition, it plays a dual role as a control point for B-spline computations and as a Euclidean point used for simple geometric manipulations, such as computing distances, bounding boxes, and so on. A third requirement is to be able to refer to this entity as P[i], just as its mathematical abstraction is referred to in the book. To simplify matters and to satisfy object based concepts, we separated control points from "regular" Euclidean points just as we did in the book, by writing $\mathbf{P}$ for a Euclidean point and $\mathbf{P}^w$ for a control point that can be weighted. Hence the definitions of points are (see Figure 13.1):

```
typedef struct point
{
  REAL x,
       y,
       z;
} POINT;

typedef struct cpoint
{
  REAL x,
       y,
       z,
       w;
} CPOINT;
```

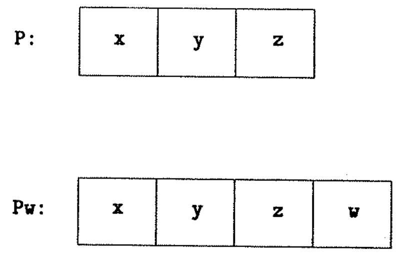

Figure 13.1. Point and control point data structures.

The attractive feature of these definitions is that they make programming fairly clean. The price one pays for this is waste of memory, e.g., if the curve is a two-dimensional nonrational curve, then half of the memory used to store the control points is wasted. While this certainly was an issue in the past, code understanding and maintenance have become more important then speed and memory. There is one thing, however, that one can do to restrict arithmetic operations to two-dimensional or three-dimensional in case the curve is nonrational or planar. The third and/or the fourth component can be set to a special value. *Nlib V2.0* uses the following:

```
#define _NOZ +BIGD
#define _NOW -BIGD
```

If the third component is set to `_NOZ` (no z), then no operation is performed on the z component, and similarly in case of `_NOW`, the w component is ignored.

Having defined points and control points, let us now build more complex objects. The next level of abstraction contains the control polygon and knot vector objects defined as (see Figures 13.2 and 13.3):

```
typedef struct cpolygon
{
  INDEX n;
  CPOINT *Pw;
} CPOLYGON;

typedef struct knotvector
{
  INDEX m;
  REAL *U;
} KNOTVECTOR;
```

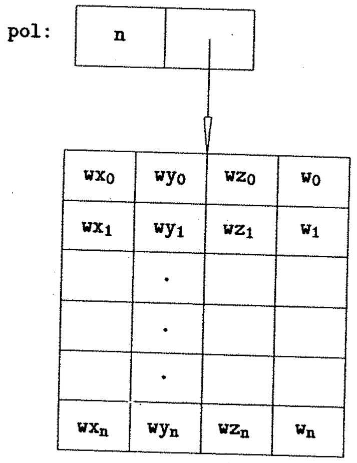

Figure 13.2. Control polygon data structure.

In other words, the control polygon object is simply a structure containing the highest index and a pointer to an array of control points. The knot vector also has a highest index field and a pointer to the array of knots. These structures contain the bare minimum of data. One can extend them according to special needs, e.g., the control polygon structure can store information on the bounding box, or the knot vector structure can store a flag as to whether it is uniform or nonuniform. Now, given a control polygon, a knot vector, and a specified degree, one can build a curve from these constituents as follows (see Figure 13.4):

```
typedef struct curve
{
  CPOLYGON *pol;
  DEGREE p;
  KNOTVECTOR *knt;
} CURVE;
```

that is, a curve object consists of a polygon pointer, the degree, and a knot vector pointer. The beauty of this definition is that the various constituents are easily extracted for further processing. For example, if one needs to compute the bounding box of a curve, in that routine the polygon subobject is extracted and passed on to another routine that deals with bounding box computation. Similarly, if a knot span is sought in which a parameter lies, the knot vector object is detached from the curve and passed on to a routine. Assuming that the current routine receives a curve pointer, the programming is fairly simple.

```
...

CPOLYGON *pol;
KNOTVECTOR *knt;
```

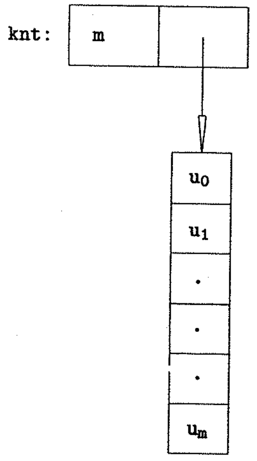

Figure 13.3. Knot vector data structure.

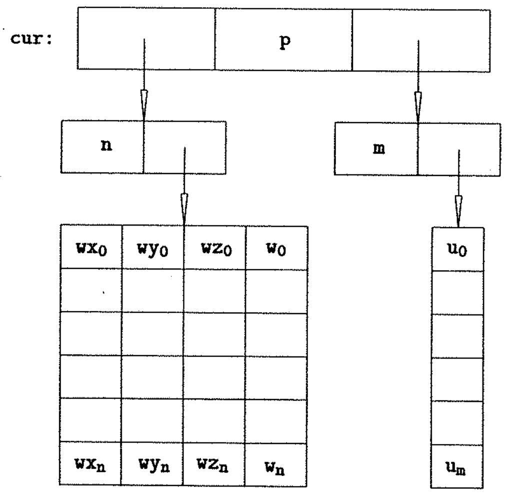

Figure 13.4. Curve data structure.

```
  ...
  pol = cur->pol;
  knt = cur->knt;
  ...
  N_polbox(pol,...);
  N_findsp(knt,...);
```

In other words, simple pointer assignments allow the program to descend to a lower level of abstraction, perform the necessary operations, and return the result to the calling routine's level.

The definition of a surface object is quite similar. It needs a control net, two degrees, and two knot vectors. The control net is defined as (see Figure 13.5):

```
  typedef struct cnet
  {
    INDEX n,
          m;
    CPOINT **Pw;
  } CNET;
```

It contains the highest indexes in both directions and a pointer to a pointer, that is, a pointer to an array of pointers pointing to the first element in each column of control points. When one writes `P[i][j]`, then `P[i]` refers to the *i*th pointer in the pointer array pointing to the beginning of the *i*th column. The index `[j]` is the *j*th offset in this column. Putting the control net, the degrees, and two knot vectors together results in a surface defined as (see Figure 13.6):

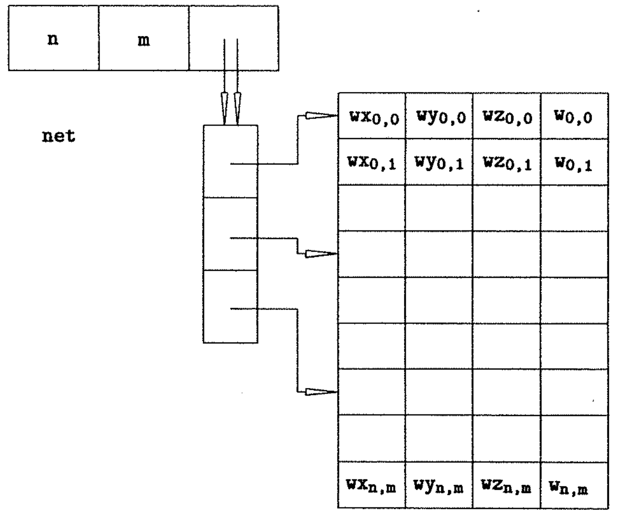

Figure 13.5. Control net data structure.

```
  typedef struct surface
  {
    CNET *net;
    DEGREE p,
           q;
    KNOTVECTOR *knu,
               *knv;
  } SURFACE;
```

Again, the control net and the knot vectors are extracted easily to perform various operations. If one has to perform many operations on the same curve or surface, these definitions become fairly natural to use and very economical; only pointers to different structures need to be passed around.

13.4 Memory Allocation

One of the most powerful tools of the C language is its ability to allocate and deallocate memory dynamically. This power, however, has its price: improperly allocating, using, or deallocating memory can cause fatal errors. Hence *Nlib V2.0* has very strict rules regarding memory allocation and deallocation. The most

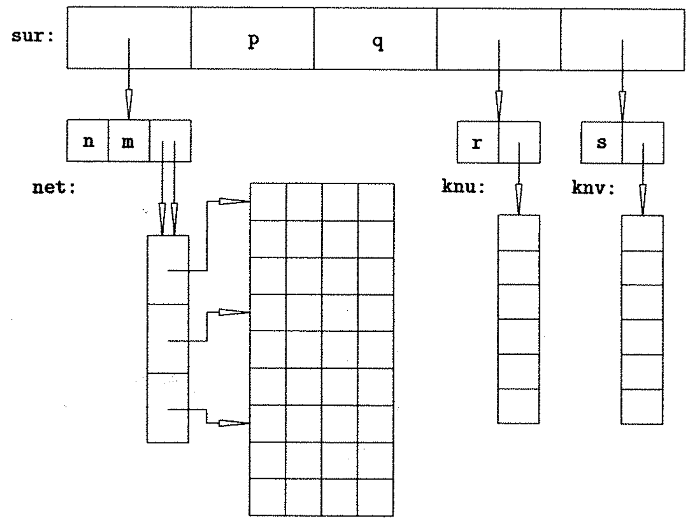

Figure 13.6. Surface data structure.

important one is that no routine is allowed to allocate and to deallocate memory other than the specially written memory handlers.

All memory allocated in a routine and not being passed back to the calling routine must be deallocated before returning. This fact, together with proper error handling, can lead to rather messy code, as this code segment illustrates.

```
x = (double *) malloc ( n1*sizeof(double) );
if ( x == NULL ) return( error );

y = (double *) malloc ( n2*sizeof(double) );
if ( y == NULL )
{
  free( x );
  return( error );
}

z = (double *) malloc ( n3*sizeof(double) );
if ( z == NULL )
{
  free( x );
  free( y );
  return( error );
}
```

This process can quickly become fairly complex if several allocations are required.

One can easily see that $n+1$ memory allocations require $n(n+1)/2$ lines of code for deallocation to handle the possible error, e.g., a code requiring 21 arrays contains 210 lines of `free` statements just to deal with the possible failure of dynamic memory allocation.

To make matters more manageable, *Nlib V2.0* has a three-step process to handle memory allocation and deallocation:

- on entering a new program, a memory stack is initialized to the empty stack;
- if memory is needed, special routines are called that allocate the required memory and save the memory pointers on appropriate stacks (see Figure 13.7);
- upon leaving the program the memory stacks are traversed, and memory, pointed to by the saved pointers, is deallocated.

As the programming examples in a later section show, this type of memory management is fairly clean, hides all unnecessary detail, and makes sure that all allocated memory is deallocated properly. Let us now examine the three steps in more detail.

The structure of every NURBS routine is sketched using this simple example:

```
...
FLAG error = 0;
INDEX n, m;
CPOINT **Pw;
```

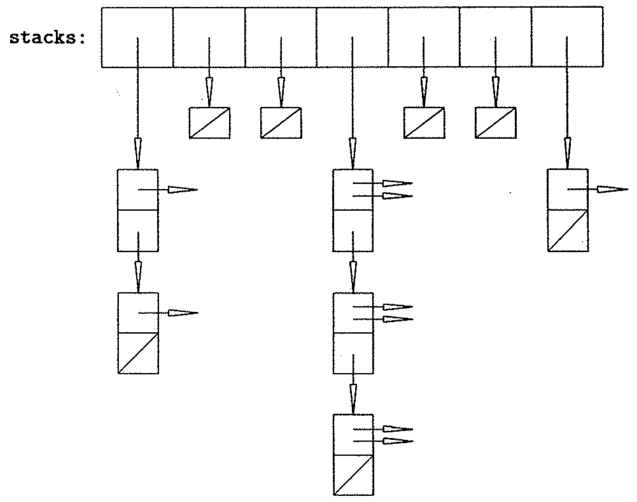

Figure 13.7. Memory stacks.

```
STACKS S;
...
N_inurbs(&S);
...
Pw = S_getc2d(n,m,&S);
if ( Pw == NULL ) { error = 1; goto EXIT; }
...
Pw[i][j] = ...;
...
EXIT:

N_enurbs(&S);
return( error );
```

Each routine has its own memory stack defined in nurbs.h as follows (see Figure 13.7):

```
typedef struct stacks
{
  I1DNODE *i1d;
  I2DNODE *i2d;
  R1DNODE *r1d;
  R2DNODE *r2d;
  ...

  KNTNODE *knt;
} STACKS;
```

In other words, it is a structure of pointers pointing to linked lists of nodes that store the different types of memory pointers – one- or two-dimensional INTEGER and REAL arrays, POINT arrays, and so on. When a routine is entered, memory for this structure is allocated and the pointers are set to NULL by the N_inurbs() routine. In order to allocate memory, the program (1) declares the appropriate memory pointer (**Pw in the previous example), and (2) calls a memory routine (S_getc2d()). The memory routine then allocates the required memory and saves the pointer(s) on the stack. As an example, let us see how S_getc2d() allocates memory for a two-dimensional array of control points (error handling, used in this routine, is discussed in Section 3.5).

```
#include "nurbs.h"
static STRING rname = "S_GETC2D";

CPOINT **S_getc2d( INDEX n, INDEX m, STACKS *S )
{
  INDEX k, l;
  CPOINT **Pw, *Qw;
```

```
  C1DNODE *c1d;
  C2DNODE *c2d;

  /* Allocate memory */

  Pw = (CPOINT **) malloc ( (n+1)*sizeof(CPOINT *) );
  if ( Pw == NULL )
  {
    E_seterr(MEM_ERR,rname);
    return NULL;
  }

  Qw = (CPOINT *) malloc ( (n+1)*(m+1)*sizeof(CPOINT) );
  if ( Qw == NULL )
  {
    E_seterr(MEM_ERR,rname);
    free( Pw );
    return NULL;
  }

  /* Make pointer assignments */

  l = 0;
  for ( k=0; k<=n; k++ )
  {
    Pw[k] = &Qw[l];
    l = l+m+1;
  }

  /* Put pointers on memory stacks */

  c1d = (C1DNODE *) malloc ( sizeof(C1DNODE) );
  if ( c1d == NULL )
  {
    E_seterr(MEM_ERR,rname);
    free( Pw ); free( Qw );
    return NULL;
  }
  c1d->ptr = Qw;
  c1d->next = S->c1d;
  S->c1d = c1d;

  c2d = (C2DNODE *) malloc ( sizeof(C2DNODE) );
  if ( c2d == NULL )
  {
    E_seterr(MEM_ERR,rname);
    free( Pw );
```

```
      return NULL;
    }
    c2d->ptr = Pw;
    c2d->next = S->c2d;
    S->c2d = c2d;

    /* Exit */

    return Pw;
  }
```

The nodes C1DNODE and C2DNODE are defined in geometry.h as

```
typedef struct c1dnode
{
  CPOINT *ptr;
  struct c1dnode *next;
} C1DNODE;

typedef struct c2dnode
{
  CPOINT **ptr;
  struct c2dnode *next;
} C2DNODE;
```

Similar nodes are used for other types of memory pointers.

Once the memory is allocated it can be used as needed throughout the program. At the end, all allocated memory is freed by the N_enurbs() routine that traverses the different memory stacks, frees the allocated memory, and kills the linked lists. It is sketched as follows:

```
#include "nurbs.h"

VOID N_enurbs( STACKS *S )
{
  I1DNODE *i1p;
  ...
  KNTNODE *knp;

  /* Free integer arrays */

  while ( S->i1d != NULL )
  {
    i1p = S->i1d;
    S->i1d = i1p->next;
    free( i1p->ptr );
    free( i1p );
  }
```

```
  ...
  /* Free knot vectors */

  while ( S->knt != NULL )
  {
    knp = S->knt;
    S->knt = knp->next;
    free( knp->ptr );
    free( knp );
  }
}
```

This style of memory management provides a fairly clean way of dealing with the difficult task of memory allocation and deallocation. It frees the programmer from the burden of handling distracting details and remembering to deallocate all dynamically allocated memory. In its present form the method does not allow deallocation in the middle of a running program. If this creates a problem in a particular application, simple free routines can be written to deallocate memory and to eliminate nodes from the memory stack.

A note on the N_inurbs() and N_enurbs() routines: In the previous discussions these routines initialized and killed memory stacks. However, these routines can also be used to initialize and de-initialize other global parameters the programmer wants to use throughout a NURBS routine. This is one of the many reasons why *every* NURBS routine starts with N_inurbs() and ends with N_enurbs() even if no dynamic memory allocation is required. The concept is similar to building cars prewired for a car phone. The driver may never want a phone, but if he does the installation becomes a five-minute (clean) job. NURBS routines in *Nlib V2.0* are "prewired" for dynamic memory allocation and for initialization so that adding new features is easy and requires no change in the overall structure of the program.

13.5 Error Control

Error handling is one of the most crucial issues in designing a system. Each routine can have errors, and the calling routine has to deal with these errors in a sensible manner. There are basically two types of error processing:

- *direct control*: in case of an error a message is printed; if the error is fatal the process is killed;
- *indirect control*: each routine returns an error flag to the calling routine, and that determines how to handle the error and how to pass this information to a higher level routine.

Direct control is appropriate in a system like a compiler but is not acceptable in a NURBS system. For example, a geometry routine can return an error flag indicating that the given lines cannot be intersected because the input data, representing the lines, has corrupt information (e.g., a line of zero length). Since this error can possibly be corrected from the calling routine and hence the process completed with success, it is totally unacceptable to kill the entire session and print an error message.

NURBS routines need to handle errors in the background by passing error flags back to each calling routine. This can be done in a number of different ways. One sophisticated method is to use an *error stack.* If a routine locates an error it is put on the stack and control is returned to the calling routine. This routine either deals with the error or puts its own error onto the stack and returns control to a higher level routine. At each higher level, the calling routine receives a stack of errors (see Figure 13.8) that show the *error history*, and at each level the routine can correct the error, if possible, or pass the stack (with its own error on top) one level higher. Ultimately, a routine at a level higher than the NURBS library deals with the error. Let us consider a simple example. The data reduction algorithm discussed in Chapter 9 uses the following simple calling hierarchy:

```
Data reduction;
    Interpolate with given knot vector;
    Remove knots;
    Compute degree elevated knot vector;
    Least-squares approximation with given knot vector;
        LU decomposition;
```

If LU decomposition fails at the deepest level, least-squares approximation cannot be performed. Instead of killing the entire process, the data reduction routine can elect to restart the entire process with a higher degree interpolatory curve (which is exactly what the actual implementation does).

Although error stacking is a sophisticated technique, it requires some programming to respond to the different errors at each level. In *Nlib V2.0* this technique is not employed, mainly for two reasons: (1) each NURBS routine calls only a few other routines, i.e., the call stack is fairly small, and (2) the calling routine

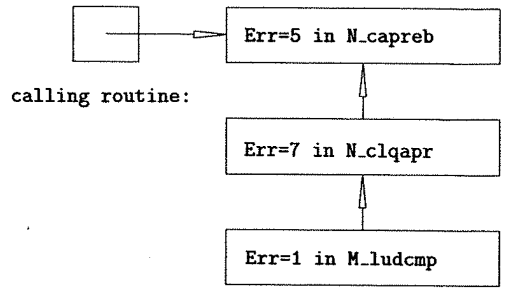

Figure 13.8. Error stack.

checks for most errors, hence only a few errors remain to be detected in lower level routines. Thus it seems reasonable for *Nlib V2.0* to pass around only one error flag, and to make each routine responsible for dealing with this error. The error flag is a simple structure defined in `datastr.h` as:

```
typedef struct enode
{
  INTEGER eno;
  STRING fna;
} ENODE;
```

that is, the structure contains an error number and the name of the routine where the error was detected. The most frequently occurring errors are numbered and defined in `datastr.h`. Some examples are:

```
#define CUR_ERR 1
#define SUR_ERR 2
#define DEG_ERR 3
#define KNT_ERR 4
#define WEI_ERR 5

...

#define MEM_ERR 15
...
```

where `CUR_ERR` is curve definition error; `KNT_ERR` is knot vector error, for example decreasing knots; `MEM_ERR` is memory allocation error, and so on; as a convenience, an error file can be created where all these errors are explained in more detail. Using these error numbers, a global error variable is required. *Nlib V2.0* defines and initializes the error variable `NERR` in `globals.h` as

```
extern volatile ENODE NERR = { 0, " " };
```

As soon as an error is detected, `NERR` receives a new value by means of the error routine `E_seterr()`:

```
#include "nurbs.h"
extern volatile ENODE NERR;

VOID E_seterr( INTEGER err, STRING name )
{
  NERR.eno = err;
  NERR.fna = name;
}
```

Nlib V2.0 has a set of error routines that check every conceivable error and set `NERR` to the appropriate value. We present two examples here; the first checks whether knots are nondecreasing, whereas the second performs a complete curve check, i.e., it checks curve definition consistency, weights, and the knot vector.

```
#include "nurbs.h"

FLAG E_kntdef( KNOTVECTOR *knt, STRING rname )
{
  FLAG error = 0;
  INDEX i, m;
  REAL *U;

  /* Convert to local notation */

  m = knt->m;
  U = knt->U;

  /* Check knots */

  for ( i=0; i<m; i++ )
  {
    if ( U[i] > U[i+1] )
    {
      E_seterr(KNT_ERR,rname);
      error = 1;
      break;
    }
  }

  /* Exit */

  return( error );
}
```

In other words, if no error is found the routine returns 0, otherwise it returns 1 and sets NERR to the appropriate value.

In many applications curve data is read from a file, for example from an IGES file. Before calling any NURBS routine it is prudent to make sure that the file contains correct information, i.e., it is useful (and necessary) to perform a complete curve (surface) check. An additional positive feature of this check is that further curve (surface) error checking is not required when curves and surfaces are passed to a NURBS routine. Before the complete curve check routine is considered, two more error routines are discussed: (1) check curve definition consistency, and (2) check weights. First consider the curve definition check routine.

```
#include "nurbs.h"

FLAG E_curdef( CURVE *cur, STRING rname )
{
  FLAG olddim, oldrat, newdim, newrat, error = 0;
  INDEX i, n, m;
```

```
  DEGREE p;
  CPOINT *Pw;

  /* Convert to local notation */

  n = cur->pol->n;
  Pw = cur->pol->Pw;
  p = cur->p;
  m = cur->knt->m;

  /* Check definition */

  if ( (n+p+1) != m )
  {
    E_seterr(CUR_ERR,rname);
    error = 1;
    goto EXIT;
  }

  /* Check for consistency */

  if ( Pw[0].z == _NOZ ) olddim = 2; else olddim = 3;
  if ( Pw[0].w == _NOW ) oldrat = 0; else oldrat = 1;
  for ( i=1; i<=n; i++ )
  {
    if ( Pw[i].z == _NOZ ) newdim = 2; else newdim = 3;
    if ( newdim != olddim )
    {
      E_seterr(CUR_ERR,rname);
      error = 1;
      break;
    }
    if ( Pw[i].w == _NOW ) newrat = 0; else newrat = 1;
    if ( newrat != oldrat )
    {
      E_seterr(CUR_ERR,rname);
      error = 1;
      break;
    }
  }

  /* Exit */

  EXIT:

  return( error );
}
```

The weight check routine is:

```
#include "nurbs.h"

FLAG E_curwei( CURVE *cur, STRING rname )
{
  FLAG error = 0;
  INDEX i, n;
  CPOINT *Pw;

  /* Convert to local notation */

  n = cur->pol->n;
  Pw = cur->pol->Pw;

  /* Check if curve is rational */

  if ( Pw[0].w == _NOW ) return( 0 );

  /* Now check the weights */

  for ( i=0; i<=n; i++ )
  {
    if ( Pw[i].w <= 0.0 )
    {
      E_seterr(WEI_ERR,rname);
      error = 1;
      break;
    }
  }

  /* Exit */

  return( error );
}
```

As a practical matter, application programmers might want to set a limit to the largest permissible weight and check for it in this routine. Large weights create very bad parametrizations, which in turn can result in unacceptable surfaces constructed from curves.

The input checking routine is:

```
#include "nurbs.h"

FLAG E_curinp( CURVE *cur, STRING rname )
{
```

```
  FLAG error;
  KNOTVECTOR *knt;

  /* Convert to local notation */

  knt = cur->knt;

  /* Check curve definition */

  error = E_curdef(cur,rname);
  if ( error > 0 ) return(1);

  /* Check curve weights */

  error = E_curwei(cur,rname);
  if ( error > 0 ) return(1);

  /* Check knot vector */

  error = E_kntdef(knt,rname);
  if ( error > 0 ) return(1);

  /* Exit */

  return(0);
}
```

Each NURBS routine can, in principle, handle error checking individually. However, with error routines programming becomes a lot cleaner. The programmer can focus on the task at hand, e.g., curve refinement, and miscellaneous tasks such as error checking and memory allocation are handled at a conceptual level.

13.6 Utility Routines

Utility routines aid the programmer in creating special structures or in converting one type of input to another. There is an infinite variety of such utility programs, but one must draw the line somewhere. In *Nlib V2.0* a few dozen such routines exist that support special structure constructions.

Creating a curve structure can be done in a number of different ways, depending on what type of information is available at which level. A simple example is:

```
#include "nurbs.h"

VOID U_makcu1( CURVE *cur, CPOLYGON *pol, DEGREE p,
               KNOTVECTOR *knt )
{
    cur->pol = pol;
    cur->p = p;
    cur->knt = knt;
  }
```

This is probably the simplest type of utility responsible for assigning pointers of existing structures to the elements of the curve structure. Polygons and knot vectors can also be created by utility routines. A simple example is:

```
#include "nurbs.h"

FLAG U_makpl2( CPOLYGON *pol, REAL *wx, REAL *wy, REAL *wz,
               REAL *w, INDEX n, STACKS *S )
{
  INDEX i;
  CPOINT *Pw;

  /* Allocate memory for point array */

  Pw = S_getc1d(n,S);
  if ( Pw == NULL ) return (1);

  /* Fill in point array */

  for ( i=0; i<=n; i++ )
  {
    Pw[i].x = wx[i];
    Pw[i].y = wy[i];
    if ( wz != NULL )
      Pw[i].z = wz[i];
    else
      Pw[i].z = _NOZ;
    if ( w != NULL )
      Pw[i].w = w[i];
    else
      Pw[i].w = _NOW;
  }

  /* Make pointer assignments */

  pol->n = n;
  pol->Pw = Pw;

  /* Exit */

  return (0);
}
```

This routine not only creates a polygon structure (memory to store the structure is allocated in the calling routine), it also converts the traditional (wx, wy, wz, w) type input to the more object based point input. And it also takes care of the proper setting of the z and w components for nonrational or planar curves. We present a similar routine that creates a curve from the smallest bits and pieces:

```
#include "nurbs.h"

FLAG U_makcu3( CURVE *cur, REAL *wx, REAL *wy, REAL *wz,
               REAL *w, INDEX n, DEGREE p, REAL *U, INDEX m,
               STACKS *S )
{
  CPOINT *Pw;
  CPOLYGON *pol;
  KNOTVECTOR *knt;

  /* Allocate memory */

  Pw = S_getcpa(wx,wy,wz,w,n,S);
  if ( Pw == NULL ) return (1);
  pol = S_getpol(S);
  if ( pol == NULL ) return (1);
  knt = S_getknt(S);
  if ( knt == NULL ) return (1);

  /* Make pointer assignments */

  pol->n = n;
  pol->Pw = Pw;
  knt->m = m;
  knt->U = U;
  cur->pol = pol;
  cur->p = p;
  cur->knt = knt;

  /* Exit */

  return (0);
}
```

Notice how well the memory routines `S_getcpa()`, `S_getpol()`, and `S_getknt()` are integrated with this utility. No explicit allocation and deallocation is taking place, yet there is proper error handling through the global error flag **NERR**. This routine is also a good example illustrating how errors are handled in a higher level routine. If an error is detected in any of the three previous memory routines, **NERR** is set and control is returned. **U_makcu3** handles the error by returning the control one level higher. Since memory allocation is the problem in both the memory routines and in the utility routine, no new setting of the global **NERR** is required.

As a final example, let us consider a surface utility routine which creates a surface from control point and knot vector information.

```
#include "nurbs.h"

FLAG U_maksu2( SURFACE *sur, CPOINT **Pw, INDEX n, INDEX m,
               DEGREE p, DEGREE q, REAL *U, REAL *V, INDEX r,
               INDEX s, STACKS *S )
{
  CNET *net;
  KNOTVECTOR *knu, *knv;

  /* Allocate memory for control net */
  /* and knot vector structures */

  net = S_getnet(S);
  if ( net == NULL ) return(1);
  knu = S_getknt(S);
  if ( knu == NULL ) return(1);
  knv = S_getknt(S);
  if ( knv == NULL ) return(1);

  /* Make pointer assignments */

  net->n = n;
  net->m = m;
  net->Pw = Pw;
  knu->m = r;
  knu->U = U;
  knv->m = s;
  knv->U = V;
  sur->net = net;
  sur->p = p;
  sur->q = q;
  sur->knu = knu;
  sur->knv = knv;

  /* Exit */

  return (0);
}
```

13.7 Arithmetic Routines

Performing operations on NURBS curves and surfaces, e.g., evaluation or knot insertion, requires arithmetic operations on the control points. Since control points are considered as objects, special arithmetic routines are needed to perform such

tasks as convex combination (corner cutting), summation, or projection to Euclidean space. Besides performing arithmetic on objects, these routines help to understand the programs and make it possible to port NURBS routines to different environments. For example, assume that a three-dimensional design system, based on rational curves and surfaces, is to be used as a two-dimensional drafting system employing only nonrational curves. Since the control point object is dimensionless, only its definition needs to be changed and a few arithmetic routines need to be modified. The rest of the system, which could contain millions of lines of code, remains unchanged.

We show a few examples of arithmetic routines used in *Nlib V2.0*. The first example is an initialization routine. Computing a point on a curve or surface calls for summation, which requires that the sum be initialized to zero.

```
#include "nurbs.h"

VOID A_initcp( CPOINT Aw, CPOINT *Bw )
{
  Bw->x = Aw.x;
  Bw->y = Aw.y;
  Bw->z = Aw.z;
  Bw->w = Aw.w;
}
```

This routine takes proper care of the different types of control points, rational or nonrational, two-dimensional or three-dimensional, through `Aw`'s third and fourth coordinates.

The next example is a routine that updates a given control point used again to compute a sum of blended control points.

```
#include "nurbs.h"

VOID A_updcpt( REAL alpha, CPOINT Aw, CPOINT *Bw )
{
  Bw->x = Bw->x + alpha*Aw.x;
  Bw->y = Bw->y + alpha*Aw.y;
  if ( Aw.z != _NOZ )
    Bw->z = Bw->z + alpha*Aw.z;
  else
    Bw->z = _NOZ;
  if ( Aw.w != _NOW )
    Bw->w = Bw->w + alpha*Aw.w;
  else
    Bw->w = _NOW;
}
```

As a final example we consider the projection routine, which maps a rational point to the Euclidean space.

```
#include "nurbs.h"

VOID A_euclid( CPOINT Pw, POINT *P )
{
  if ( Pw.w != _NOW )
  {
    P->x = Pw.x/Pw.w;
    P->y = Pw.y/Pw.w;
    if ( Pw.z != _NOZ )
      P->z = Pw.z/Pw.w;
    else
      P->z = 0.0;
  }
  else
  {
    P->x = Pw.x;
    P->y = Pw.y;
    if ( Pw.z != _NOZ )
      P->z = Pw.z;
  else
      P->z = 0.0;
  }
}
```

These routines are used in the next section to illustrate two NURBS routines, curve and surface evaluations.

13.8 Example Programs

In this section we present two complete programs, one for curve evaluation and the other for surface evaluation. These routines show the structure of NURBS programs and how the different utility routines are integrated with higher level NURBS routines. First consider the curve evaluator.

```
#include "nurbs.h"
static STRING rname = "N_EVNCUR";
extern CPOINT CZERO;

FLAG N_evncur( CURVE *cur, PARAMETER u, POINT *C )
{
  FLAG error = 0;
  INDEX i, j, k;
  DEGREE p;
  REAL *N;
  KNOTVECTOR *knt;
  CPOINT *Pw, Cw;
```

```
  STACKS S;

  /* Start NURBS environment */

  N_inurbs(&S);

  /* Get local notation */

  Pw = cur->pol->Pw;
  p = cur->p;
  knt = cur->knt;

  /* Check parameter */

  error = E_parval(knt,u,rname);
  if ( error == 1 ) goto EXIT;

  /* Compute non-vanishing B-splines */

  N = S_getr1d(p,&S);
  if ( N == NULL ) { error = 1; goto EXIT; }
  error = N_allbas(knt,p,u,LEFT,N,&j);
  if( error == 1 ) goto EXIT;

  /* Compute the point on the curve */

  A_initcp(CZERO,&Cw);
  for ( i=0; i<=p; i++ )
  {
    k = j-p+i;
    A_updcpt(N[i],Pw[k],&Cw);
  }
  A_euclid(Cw,C);

  /* End NURBS and exit */

  EXIT:

  N_enurbs(&S);
  return(error);
}
```

As mentioned earlier, each NURBS routine starts with `N_inurbs()`, which initializes the memory stacks to `NULL`. After initialization, local notations are normally introduced to help follow the computational details. The notation `Pw[k]` is visually more pleasing than `cur->pol->Pw[k]` and much closer to the customary notation, $\mathbf{P}_k^w$. The next step in almost every NURBS routine is to check the incoming parameters for possible error. `E_parval()` checks if the given parameter u is outside of the range $[u_0, u_m]$. The rest of the code deals with the actual computation of a curve point. To do so, the nonvanishing basis functions are computed first after the proper amount of memory is allocated to store them (since there is no limit on the degree, the memory must be dynamically allocated). Arithmetic routines, discussed in the previous section, are then used to compute the sum

$$\mathbf{C}^w(u) = \sum_{k=j-p}^{j} N_{k,p}(u)\mathbf{P}_k^w$$

and to locate the point in the Euclidean space; note the index transformation to use the proper basis functions. Finally, N_enurbs() deallocates all allocated memory, and control is returned to the calling routine by returning either zero (no error) or one (error saved in NERR).

This next test program shows how easy it is to compute a point on the curve.

```
#include "nurbs.h"
#include "globals.h"

REAL wx[6] = { 0.2, 0.8, 0.4, 1.2, 1.2, 0.8 };
REAL wy[6] = { 0.2, 0.4, 0.4, 0.8, 0.4, 0.2 };
REAL wz[6] = { 0.2, 0.4, 0.2, 0.4, 0.4, 0.2 };
REAL w[6] = { 1.0, 2.0, 1.0, 2.0, 2.0, 1.0 };
REAL U[10] = { 0.0, 0.0, 0.0, 0.0, 0.3, 0.7, 1.0, 1.0, 1.0,
               1.0 };

main( )
{
  FLAG error;
  CURVE cur;
  PARAMETER u;
  POINT C;
  STACKS S;

  /* Start processing */

  N_inurbs(&S);

  /* Create the curve */

  U_makcu3(&cur,wx,wy,wz,w,5,3,U,9,&S);

  /* Compute a point on the curve */

  printf( "Enter parameter u = " ); scanf( "%lf", &u );
  error = N_evncur(&cur,u,&C);
  if ( error > 0 )
  {
    printf( "error = %d %s", NERR.eno, NERR.fna );
    goto EXIT;
```

```
  }
  printf( "C = %lf %lf %lf", C.x, C.y, C.z );

  /* End NURBS session */

  EXIT:

  N_enurbs(&S);
}
```

The many utility routines allow the programmer to define a curve in a number of different ways, e.g., reading in a curve file, receiving the (wx, wy, wz, w) coordinates from a curve sketching interface, and so on.

The structure of surface programs is very similar to the curve programs. This next surface evaluator shows an example.

```
#include "nurbs.h"
static STRING rname = "N_EVNSUR";
extern CPOINT CZERO;

FLAG N_evnsur( SURFACE *sur, PARAMETER u, PARAMETER v,
               POINT *S )
{
  FLAG error = 0;
  INDEX i, j, k, l, ju, jv;
  DEGREE p, q;
  REAL *NU, *NV;
  KNOTVECTOR *knu, *knv;
  CPOINT **Pw, *Tw, Sw;
  STACKS ST;

  /* Start NURBS environment */

  N_inurbs(&ST);

  /* Get local notation */

  Pw = sur->net->Pw;
  p = sur->p;
  q = sur->q;
  knu = sur->knu;
  knv = sur->knv;

  /* Check parameters */

  error = E_parval(knu,u,rname);
  if ( error == 1 ) goto EXIT;
  error = E_parval(knv,v,rname);
```

```
    if ( error == 1 ) goto EXIT;

    /* Compute nonvanishing B-splines */

    NU = S_getr1d(p,&ST);
    if ( NU == NULL ) { error = 1; goto EXIT; }
    NV = S_getr1d(q,&ST);
    if ( NV == NULL ) { error = 1; goto EXIT; }
    error = N_allbas(knu,p,u,LEFT,NU,&ju);
    if ( error == 1 ) goto EXIT;
    error = N_allbas(knv,q,v,LEFT,NV,&jv);
    if ( error == 1 ) goto EXIT;

    /* Compute the point on the surface */

    Tw = S_getc1d(p,&ST);
    if ( Tw == NULL ) { error = 1; goto EXIT; }
    for ( i=0; i<=p; i++ )
    {
      A_initcp(CZERO,&Tw[i]);
      k = ju-p+i;
      for ( j=0; j<=q; j++ )
      {
        l = jv-q+j;
        A_updcpt(NV[j],Pw[k][l],&Tw[i]);
      }
    }

    A_initcp(CZERO,&Sw);
    for ( i=0; i<=p; i++ )
    {
      A_updcpt(NU[i],Tw[i],&Sw);
    }
    A_euclid(Sw,S);

    /* End NURBS and exit */

    EXIT:

    N_enurbs(&ST);
    return(error);
  }
```

The next test program shows another way to create a surface and to evaluate a point on it. The routine `U_inpsur()` reads in a data file, named `SURFACE.DAT`, containing the control polygon and knot vector information. `U_inisur()` is a routine that initializes the surface object to the `NULL` object to let `U_inpsur()` know that memory allocation is required.

```
#include "nurbs.h"
#include "globals.h"

main( )
{
  FLAG error;
  SURFACE sur;
  PARAMETER u, v;
  POINT S;
  STACKS ST;

  /* Start processing */

  N_inurbs(&ST);

  /* Read in surface data */

  U_inisur(&sur);
  error = U_inpsur(&sur,"SURFACE.DAT",&ST);
  if ( error > 0 )
  {
    printf( "error = %d %s", NERR.eno, NERR.fna );
    goto EXIT;
  }

  /* Compute surface point */

  printf( "Enter parameters <u,v> = " ); scanf( "%lf%lf", &u, &v );
  error = N_evnsur(&sur,u,v,&S);
  if ( error > 0 )
  {
    printf( "error = %d %s", NERR.eno, NERR.fna );
    goto EXIT;
  }
  printf( "S = %lf %lf %lf", S.x, S.y, S.z );

  /* End NURBS session */

  EXIT:

  N_enurbs(&ST);
}
```

13.9 Additional Structures

In order for the NURBS library to be useful for a large variety of purposes, some additional structures (objects) are required. These structures are mostly used to convert from the NURBS form to other forms, to reparametrize a NURBS entity, and so on.

A simple modification of the NURBS curve and surface objects results in scalar-valued univariate and bivariate B-spline functions. The only change one has to make is to replace the control polygon and control net objects with curve values and surface values defined as:

```
typedef struct cvalue
{
  INDEX n;
  REAL *fu;
} CVALUE;

typedef struct svalue
{
  INDEX n,
        m;
  REAL **fuv;
} SVALUE;
```

Special routines, for example function evaluation and knot insertion, can be written that deal with functions that are used mostly for functional compositions (see Section 6.4).

Vector valued polynomials play an important role in data exchange and in form conversion. *Nlib V2.0* has curve and surface polynomial entities to be able to convert from the NURBS form to piecewise polynomial form, and back. These are defined as:

```
typedef struct cpoly
{
  CVECTOR *cve;
  DEGREE p;
  INTERVAL I;
} CPOLY;

typedef struct spoly
{
  SVECTOR *sve;
  DEGREE p,
         q;
  RECTANGLE R;
} SPOLY;
```

where the curve vector CVECTOR and surface vector SVECTOR entities are defined just as the control polygon and control net objects. The interval and rectangle objects are simply

```
typedef struct interval
{
  PARAMETER ul,
            ur;
} INTERVAL;
```

```
typedef struct rectangle
{
  PARAMETER ul,
            ur,
            vb,
            vt,
} RECTANGLE;
```

That is, polynomials are defined by their vector coefficients and the one- and two-dimensional interval/rectangle to which the polynomial is restricted.

Bézier curves and surfaces play an important role in any NURBS-based system, not because they are special cases of NURBS but rather because many operations can be performed very efficiently on the individual Bézier components. For this reason, two special structures are introduced, one for curves and one for surfaces.

```
typedef struct bcurve
{
  CPOLYGON *pol;
  INTERVAL I;
} BCURVE;

typedef struct bsurface
{
  CNET *net;
  RECTANGLE R;
} BSURFACE;
```

Thus the control polygon/net and the parameter interval/rectangle uniquely define a Bézier curve/surface.

In closing, we present a few additional objects used mainly in simple geometric computations and in some mathematical routines. *Nlib V2.0* has extensive geometry and mathematics libraries used to support various NURBS operations. First we consider some geometric objects.

```
typedef struct vector
{
  REAL x,
       y,
       z;
} VECTOR;

typedef struct line
{
  POINT P;
  VECTOR V;
  FLAG bounded;
} LINE;
```

```
typedef struct plane
{
  POINT P;
  VECTOR N;
} PLANE;

typedef struct circle
{
  POINT C;
  REAL r;
  VECTOR N;
} CIRCLE;
```

Although the POINT and the VECTOR structures contain the same structure elements they are conceptually different, hence their introductions. A last example is the matrix object that represents real matrices.

```
typedef struct rmatrix
{
  INDEX n,
        m;
  REAL **RM;
  MATRIXTYPE mtp;
  INDEX bandwidth;
} RMATRIX;
```

where MATRIXTYPE is defined as follows:

```
typedef enum
{
  full,
  lowerleft,
  upperright,
  banded
} MATRIXTYPE;
```

Similar structures are available for integer (IMATRIX), point (PMATRIX), and control point (CMATRIX) matrices. Row and column *vectors* are considered as $(1 \times m)$ and $(n \times 1)$ matrices, respectively. No special vector structure is introduced.

13.10 System Structure

Nlib V2.0 has the following routines as parts of a complex structure:

- NURBS routines (N_*.c);
- Bézier routines (B_*.c);
- geometry routines (G_*.c);

- mathematics routines (M_*.c);
- utility routines (U_*.c);
- memory allocation routines (S_*.c);
- error handling routines (E_*.c);
- arithmetic routines (A_*.c).

These routines must be integrated into a complex system that satisfies all the requirements mentioned in the previous sections. The dependence relationship of each module is shown in Figure 13.9. The different layers represent individual modules, and the arrows illustrate their dependence on lower level routines, i.e., they make calls to these routines. In the center there are three modules, memory allocation, error handling, and arithmetic routines. Although these are somewhat dependent on one another, e.g., error routines might need some memory, and memory routines have to report errors, their dependence is not as substantial as those in the upper layers. As we go to the second and higher layers, routines become dependent on all the lower layer programs. For example, mathematics routines need utility programs to create matrices; they need memory, error checking, and special arithmetic to perform matrix operations with matrices having control points as matrix elements. At the top there are the NURBS programs that make calls to every other program in the system. For instance, it is very

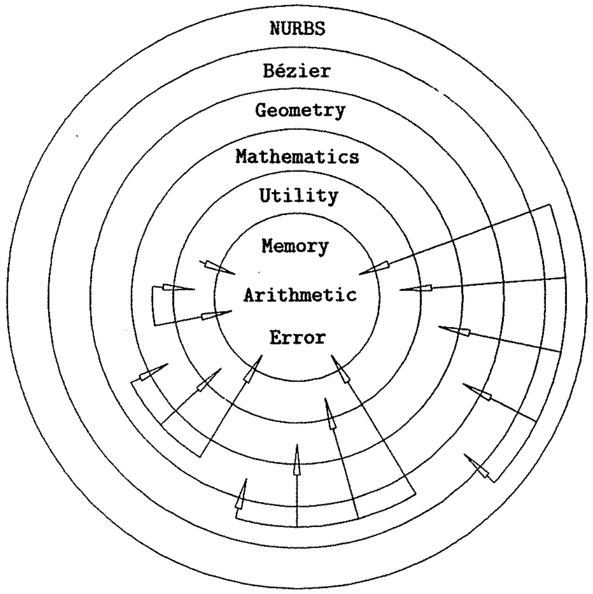

Figure 13.9. *Nlib V2.0*'s system structure.

rare that a geometry routine has to make a call to an existing NURBS program. If this happens to be the case, a special lower level routine can be written so that the correspondence among the layers remains unidirectional. This isolation of the different layers becomes important if parts of the system are to be used for purposes other than NURBS geometry. For example, the mathematics and geometry libraries should be stand-alone libraries so that they can be used for simple geometric and mathematical computations without worrying about the much larger Bézier and NURBS libraries.

We close this section by restating our opinion on system design: it is very difficult, it is not unique, and it is changing as computer technology advances. However, designing and implementing a system is enlightening and essential in obtaining a broad and deep knowledge of NURBS technology. As Herbert B. Voelcker puts it [Pieg93]

> "It is important to do both theoretical research and experimental system building. They are synergistic, and the exclusive pursuit of either can lead to sterile theory or quirky, opaque systems."

This book is intended to provide both the theory and the implementation of NURBS. Let the reader judge whether this goal has been successfully met.