**Manchester Computing Centre**
**Computer Graphics Unit**
Director: W. T. Hewitt

# The NURBS Procedure Library

W.T. Hewitt and D. Yip

$1^{st}$ | CGU 76 | 22nd May, 1992

# Contents

**1 Disclaimer** **4**

**2 Introduction** **5**

**3 The Mathematics of NURBS** **5**

3.1 Curves . . . . . 5

3.2 Surfaces . . . . . 6

**4 Creation Operators** **8**

4.1 2D . . . . . 8

4.2 3D . . . . . 8

**5 Using the Curve operations on Surfaces** **8**

**6 Other Operations** **10**

6.1 Degree Elevation . . . . . 10

6.2 Interpolation . . . . . 10

**7 Evaluation** **10**

7.1 Splitting NURBS . . . . . 11

**8 Surface Operators** **11**

8.1 Extrusion . . . . . 11

8.2 Surface of Revolution . . . . . 12

8.3 Ruled Surface . . . . . 12

8.4 Transposition . . . . . 13

8.5 Transformations . . . . . 13

8.6 Drawing NURBS . . . . . 14

**9 Constants and Records** **15**

9.1 Constants . . . . . 15

9.2 Data Structures . . . . . 15

**10 Use of the Library** **16**

**11 NURBS Application Level Procedures** **19**

11.1 Creation Operators . . . . . 36

11.2 Drawing Procedures . . . . . 38

**12 Installation** **41**

12.1 For UNIX . . . . . . 41

12.2 For VMS . . . . . . 41

12.3 Building the Library . . . . . . 42

12.3.1 UNIX . . . . . . 42

12.3.2 VMS . . . . . . 42

**13 Example Programs** **42**

13.1 Interactive NURBS Demonstration . . . . . . 44

**14 Compiling and Linking** **45**

14.1 Build for VMS . . . . . . 45

14.2 Build for UNIX . . . . . . 46

# 1 Disclaimer

This work was originally developed by Mr W. T. Hewitt. This version of the NURBS procedure library was funded by the Advisory Group On Computer Graphics (AGOCG).

This library is distributed in the hope that it will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

## 2 Introduction

This document describes a set of Non-Uniform Rational B-Spline (NURB) procedures. NURBS can be thought of as a superset of the more commonly known B-spline, they represent the state of the art in curve and surface design and representation.

The NURBS procedure library aims to make available a set of functions specifically for the creation, manipulation and display of NURBS for the non-expert.

NURBS are well known for their excellent geometric properties and characteristics [13].

- They offer a common mathematical form for representing and designing both standard analytic shapes and free-form curves and surfaces.
- NURBS are able to exactly represent conics.
- NURBS are invariant under scaling, rotation, translation and shear as well as parallel and perspective projection.
- By manipulating the control points as well as the weights, NURBS provide the flexibility to design a large variety of shapes.
- Evaluation is reasonably fast and computationally stable
- NURBS have a clear geometric interpretations, making them particularly useful for designers.
- NURBS have a powerful geometric tool kit (knot insertion/refinement/ removal, degree elevation, splitting, etc.) that can be used through out to design, analyze process and interrogate objects.
- NURBS are genuine generalizations of non-rational B-spline forms as well as rational and non-rational Bézier curves and surfaces.
- Provide the ability to model discontinuities.

The NURBS library relies on a set of functions TRAN from the PHIGS Toolkit project, which is also available from NETLIB and also `mcchpb.mcc.ac.uk`. This module provides transformation routines and a number of basic functions which the library is dependant on.

An example program showing how to use the NURBS procedures is included at the end of this document.

For any further information please contact Mr. W. T. Hewitt, hewitt@uk.ac.mcc.cgu

The authors would like to thank Steve Larkin and Manjula Patel for their contributions to the library.

I would also like to thank Colin Pinks for Beta testing the library for us.

## 3 The Mathematics of NURBS

### 3.1 Curves

To define rational B-splines, homogeneous coordinates are used. If $P = (x, y, z)$ is a point in 3D space, the corresponding point in 4D homogeneous coordinates is $P^h = (wx, wy, wz, w)$, where $w > 0$. $w$ is called the weight; only positive homogeneous coordinates are considered.

The NURBS curve is defined as the set of points obtained by evaluating equation 1 for all values of $u$ between some $u_{min}$ and $u_{max}$.

$$\mathbf{C}^h(u) = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \sum_{i=1}^{n} N_{i,k}(u)\mathbf{P}_i^h \tag{1}$$

where the $\mathbf{N}_{i,k}(u)$ are the $k$th-order polynomial B-spline basis functions, and the $P_i^h$ are the homogeneous control points $\mathbf{P}_i^h = (x_i w_i, y_i w_i, z_i w_i, w_i)$

The 3D point is recovered by dividing the first three coordinates by the fourth:

$$\mathbf{P}(u) = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \tag{2}$$

The $N_{i,k}(u)$'s are the normalized B-spline basis functions of degree $k$ defined recursively using the Cox-de Boor relationship:

$$N_i^0(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$N_i^k(u) = \frac{(u - u_i) * N_i^{k-1}(u)}{(u_{i+k} - u_i)} + \frac{(u_{i+k+1} - u) * N_{i+1}^{k-1}(u)}{(u_{i+k+1} - u_{i+1})} \tag{4}$$

(the indeterminate quantity $\frac{0}{0}$ is treated as 0)

where $u_i$ are the so-called knots forming a knot vector. Multiplicity of knot values enables discontinuities to be represented. The usual form of the knot vector is:

$$\{u_i\} = \{u_1 = \ldots = u_k, u_{k+1}, \ldots, u_n, u_{n+1} = \ldots = u_{n+k}\} \tag{5}$$

Note the first and last knot must be repeated k times (e.g., 4 for cubic splines). This is a requirement to ensure that the curve starts and ends at the first and last control points i.e., the multiplicity ensures that the curve interpolates its first and last control point.

Thus the curve is specified by its order, a knot vector, and a list of control points. Figure 1 shows a NURBS curve together with its defining control polygon. All the information is encompassed in the `PR_nurb` data structure described in section 9.

### 3.2 Surfaces

The NURBS Surface is the tensor product of two curves:

$$\mathbf{S}(u,v) = \sum_{i=1}^{n} \sum_{j=1}^{m} N_{i,k}(u) N_{j,l}(v) \mathbf{P}_{i,j} \tag{6}$$

The surface is specified by two orders $(k, l)$, two knot vectors, and an array of control points. Figure 2 shows a NURBS surface together with its defining control polygon. All the information is encompassed in the `PR_nurb` data structure described below.

In the scheme presented below, curves and surfaces are represented using one data structure. The data structure represents a curve if $m = 1$ and $l = 0$ in equation 6.

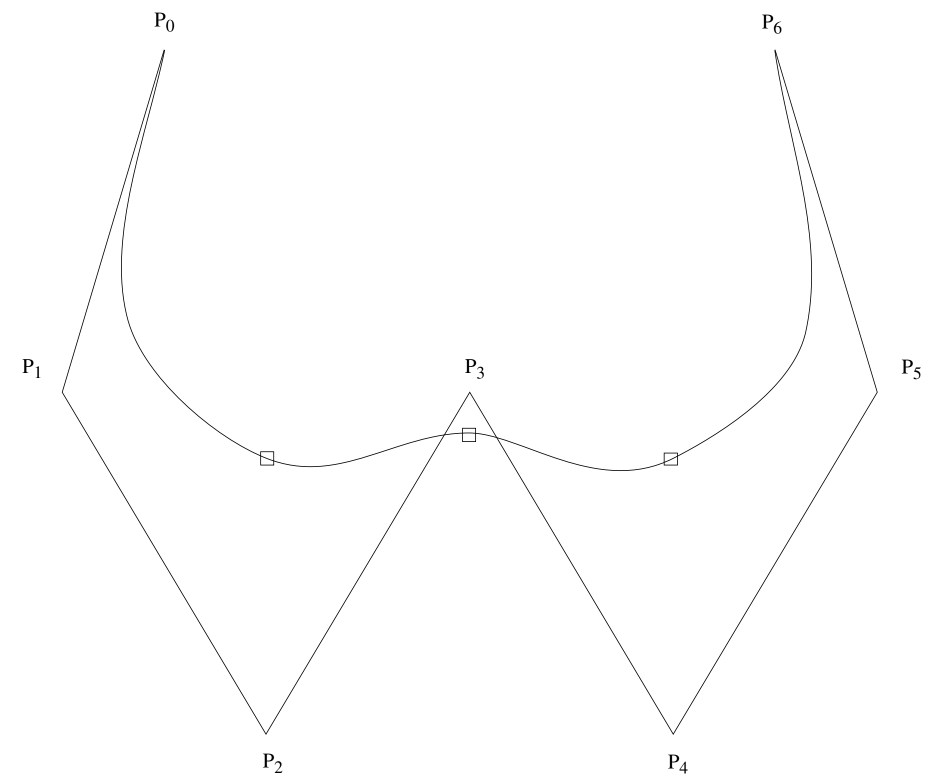


Figure 1: A Non-Rational B-spline curve with its associated control polygon

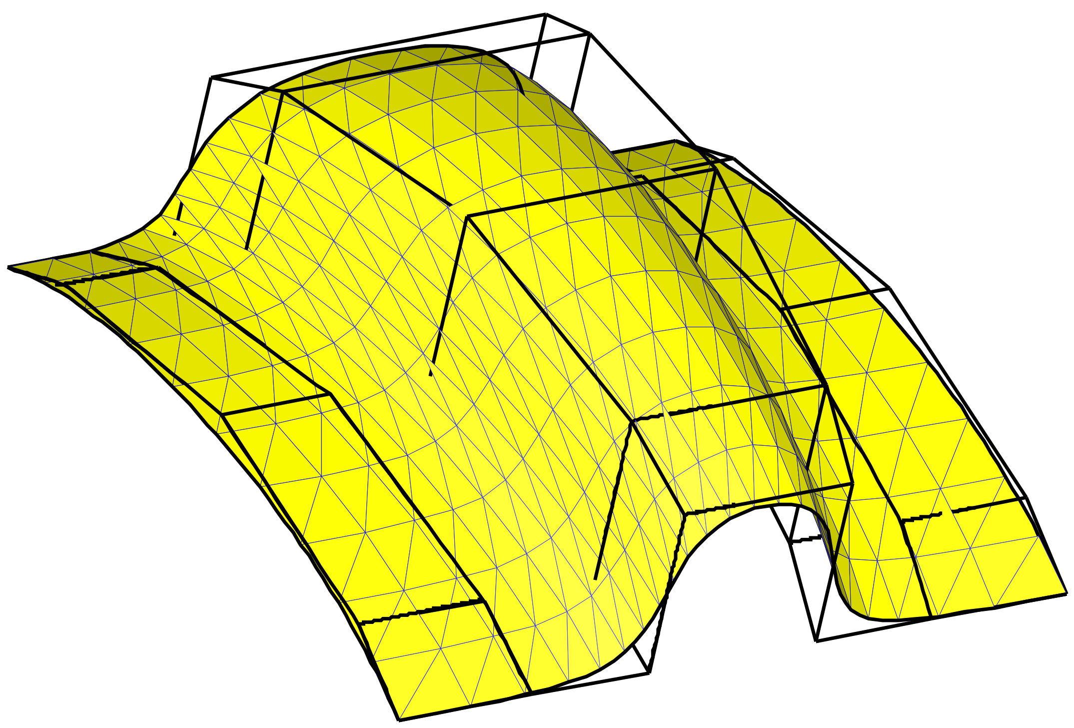

Figure 2: A Non-Rational B-spline surface with its associated control polygon

# 4 Creation Operators

The above NURBS representations for curves and surfaces are total encompassed in the `PR_nurb` data structure used through the procedure library (see section 9) to represent NURBS.

The library is equipped with a number of functions to create a series of common objects. This is achieved inserting values into the `PR_nurb` data structure.

## 4.1 2D

- Circle and Arc

  The circle is defined as a quadratic ($k = 3$) NURB, with nine control points, and twelve knots. The arc is created by splitting (see section 7.1) the circle at the appropriate points.

- Polyline

  A line can be represented as a linear NURBS ($k = 2$). If the endpoints are $\mathbf{P}_1$ and $\mathbf{P}_2$, then the knot vector will be $0, 0, 1, 1$ and the two control points will be $\mathbf{P}_1$ and $\mathbf{P}_2$. The NURBS representation of the polyline is obtained by making the list of control points the polyline points, and making the knot vector have a repeated knots at every point.

- Square

  The procedure creates a square with unit sides, centered at the origin (0, 0, 0), using the methodology of the previous section.

- Triangle

  This creates an equilateral triangle, centered at the origin, (0,0,0).

## 4.2 3D

- Sphere

  This creates a unit sphere, centered at the origin. A semi-circular arc is created, and then revolved using `nrb_revolve` (see section 8.2). The sphere is centered at (0,0,0).

- Torus

  The procedure creates a torus shape which is centered at (0,0,0). It is generated by revolving a circle around 360°.

- Vase

  This creates a test object, a vase which is centered at (0,0,0).

# 5 Using the Curve operations on Surfaces

Since curves and surfaces are very similar in nature, the operations on curves can easily be extended to surfaces.

Suppose we have some curve algorithm e.g., `nrb_refine` which takes a list of control points (denoted by):

```
1 1 1 1 1 1 1 1 1 1 1
```

and produces another NURBS with additional control points `a`:

```
1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1
```

It is relatively easy to extend the algorithm to operate on every row of an array of control points, so starting with:

```
1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4
```

an invocation of `nrb_refine` produces:

```
1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1
2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2
3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3
4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4
```

Now if we seek also to do the refinement in the 'other' direction, another version of the algorithm must be developed to take the above and do the refinement on the columns. (In general it just means duplicating the code, changing the name, and swapping around the array references). Thus this version of the algorithm operates on `1 2 3 4` not `1 1 1 1`, and introduces new points `A` for column 1, `B` for column 2, etc., resulting in:

```
1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1
A B C D E F G H I J K L M N O P Q R S T U
2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2
A B C D E F G H I J K L M N O P Q R S T U
3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3
A B C D E F G H I J K L M N O P Q R S T U
4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4
```

While this works, the code development is wasteful. de Boor [5] developed a methodology which enables the 'curve' code to be used twice. In general most of the algorithms described below must operate on $u$ and then on $v$. The simple curve algorithm is extended as above to operate once one each row, but rather than write the answers out in rows it writes the answers out in columns:

```
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
a b c d
1 2 3 4
```

Calling the algorithm again but with the 'new' surface, it operates on the rows now `1 2 3 4` (which were the original columns) producing `A B C` etc. and writes out the answers in columns, this second transposition putting the information back in the correct place:

```
1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1 a 1
A B C D E F G H I J K L M N O P Q R S T U
2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2 b 2
A B C D E F G H I J K L M N O P Q R S T U
3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3 c 3
A B C D E F G H I J K L M N O P Q R S T U
4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4 d 4
```

# 6 Other Operations

## 6.1 Degree Elevation

This elevates the degree of the spline in the $u$ direction, transposing the answer. So to degree elevate a surface, once in the $u$ and $v$ directions, use:

```
nrb_elevate(TRUE,&nrb1,&temp);
nrb_elevate(TRUE,&temp,&nrb1);
```

Most of the uses of degree elevation are for curves, and this could be achieved using:

```
nrb_elevate(TRUE,&nrb1,&nrb2);  /* Elevate the curve */
nrb_transpose(&nrb2);           /* Transpose U and V */
```

Recognizing this `nrb_elevate` takes a flag, indicating whether transposition should or should not take place. The previous sequence could then be written:

```
nrb_elevate(FALSE,&nrb1,&nrb2); /* Elevate the curve */
```

## 6.2 Interpolation

The library provides a function which can be used to create a NURBS given a set of user supplied data. `nrb_interpolate` interpolates a NURB by specifying a list of control points. A NURBS surface or curve is generated depending on whether a set of $v$ control points are specified.

`nrb_interpolate` Interpolate through a set of data points.

# 7 Evaluation

There are a number of different routines that may used to evaluate a NURB depending on the required evaluation.

The most commonly used is that of `nrb_evaluate`. This routine will evaluate a NURBS at a given $u$. If the NURBS is a surface, then the transpose is written out. For example to evaluate the curve at $u$ would be:

```
nrb_evaluate(&nrb,u,&temp)
printf("The point on the curve is: ");
nrb_putpoint4(stdout,temp.pf_ppp->pts[0])
```

To evaluate a surface at $(u, v)$ would be:

```
nrb_evaluate(&nrb,u,&temp);
nrb_evaluate(&temp,v,&temp2);
printf("The point on the surface is: ");
nrb_putpoint4(stdout,temp2.pf_ppp->pts[0]);
```

Thus `temp` contains the isoparametric NURBS curve on the surface specified by $u$.

The other NURBS evaluation routines are `nrb_bvalue` and `nrb_du`.

- `nrb_bvalue` is a general purpose routine which allows the evaluation of a derivative of a NURBS.
- `nrb_du` is used to evaluate a NURBS and its first derivative.

### 7.1 Splitting NURBS

This will split a NURBS curve in half, at some value t. Thus to split a surface into four along the isoparametric lines $u$ and $v$ use:

```
PR_nurb nrb, nrbl, nrbr, nrbl1, nrbl2, nrbr1, nrbr2;
...
nrb_split(&nrb ,u,&nrbl ,&nrbr);
nrb_split(&nrbl,v,&nrbl1,&nrbl2);
nrb_split(&nrbr,v,&nrbr1,&nrbr2);
```

The four sections of the subdivided surface are: `nrbl1, nrbl2, nrbr1`, and `nrbr2`.

## 8 Surface Operators

The library provides a number of different surface operators.

### 8.1 Extrusion

Extruded surfaces are created by defining a profile curve and then extruding the curve in a certain direction. If the profile curve has the form

$$\mathbf{C}_i(u) = \sum_{i=1}^{n} R_{i,p}(u)\mathbf{Q}_i \tag{7}$$

then the extruded surface takes the form

$$\mathbf{S}(u, v) = \sum_{i=1}^{n} \sum_{j=1}^{2} R_{i,p;j,1}(u, v)\mathbf{P}_{i,j} \tag{8}$$

Figure 3 shows a cylinder created by extruding a circle.

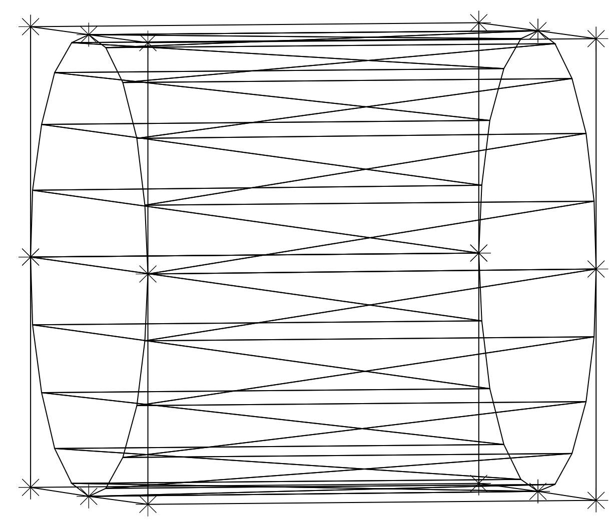

Figure 3: Cylinder created by extrusion

### 8.2 Surface of Revolution

A surface of revolution is generated by using a NURBS curve which lies in the $xz$-plane. A full surface of revolution is obtained by revolving the curve 360° about the $z$-axis. The required surface has the form

$$\mathbf{S}(u,v) = \sum_{i=1}^{9}\sum_{j=1}^{n} R_{i,2;j,k}(u,v)\mathbf{P}_{i,j} \tag{9}$$

Figure 4 shows a sphere created by revolving an arc of a circle.

### 8.3 Ruled Surface

This procedure generates a ruled surface between two nurb curves $C_1^h$ and $C_2^h$.

$$\mathbf{C}_i^h(u) = \sum_{j=1}^{n_i} R_{j,k_i}(u)\mathbf{P}_j^{i,h} \;, i = 1,2 \tag{10}$$

defined over two knots vectors. The required surface, is that of

$$\mathbf{S}(u,v) = \sum_{i=1}^{n}\sum_{j=1}^{n} R_{i,p;j,1}(u,v)\mathbf{P}_{i,j} \tag{11}$$

such that $\mathbf{S}(u,0) = \mathbf{C}_1(u)$ and $\mathbf{S}(u,1) = \mathbf{C}_2(u)$.

To create the surface between the two NURBS:

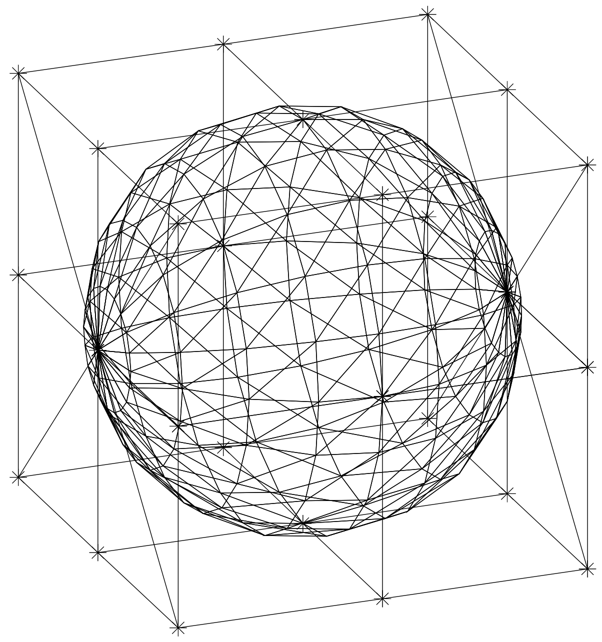

Figure 4: Sphere created by revolution

1. Degree elevate the curve to the same order.
2. Union the knot vectors of the curves.
3. Oslo each curve with the new knot vector. Each curve should now have the same knot vector and the same number of control points.
4. Create a surface using the control points and the knot vectors of the two curves.

Figure 5 shows a ruled surface between a circle and a square.

### 8.4 Transposition

As a result of some of the functions, the NURBS structure is transposed ready to be fed back into the same function. This process is usually applied to surfaces but for curves the function will only usually be called once. `nrb_tranpose` is provided to transpose the NURBS back.

Transposition also takes place at a lower level, usually within the routines themselves. `nrb_interchange` is provided to interchange the `u` and `v` components within the NURBS data structure.

| | |
|---|---|
| `nrb_transpose` | Transpose the whole NURBS structure. |
| `nrb_interchange` | Interchange only the `u` and `v` components within the NURBS data structure. |

### 8.5 Transformations

The library provides the function `nrb_xform` to transform the control points of a NURBS in euclidean space using a matrix which can be built up using a number of routines which are provided in the TRAN module.

e.g.,

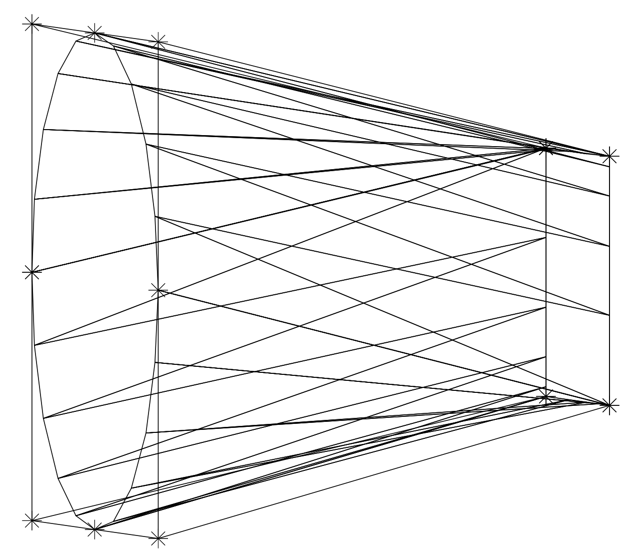

Figure 5: A ruled surface

```
PR_nurb circle, new_nurb;
Pvector3 point;
Pmatrix3 matrix;
...
/* Create a circle
*/
nrb_circle(&circle);
/* Displacement the circle
*  - shift in X
*  - rotate to XZ plane
*/
point = ptk_vector3(2.0, 0.0, 0.0);
ptk_shift3(&point, PREPLACE, matrix);
ptk_rotate3(90.0, 1, PPRECONCATENATE, matrix);
/* Apply the transformation
*/
nrb_xform(&circle, matrix);
```

### 8.6 Drawing NURBS

Provided within the library are a number of procedures to draw a NURB structure. These routines rely on the use of DEC PHIGS to produce the output and thus are provided solely for completeness and as a template for other systems.

| | |
|---|---|
| `nrb_drawsnet` | Draw a control polygon for a given surface. |
| `nrb_draws` | Draw a surface together with the control polygon. |
| `nrb_drawcnet` | Draw the control polygon for a given curve. |
| `nrb_drawc` | Draw a curve together with the control polygon. |
| `nrb_phigsplus` | Draw the NURBS using PHIGS PLUS primitives. |

# 9 Constants and Records

The following constants and record are relevant to NURBS application-level procedures, and are in the header file `nurbtype.h`.

## 9.1 Constants

```
#define PC_MaxNetpts    200  /*Maximum number of points in a net*/
#define PC_NrbMax       50   /*Pseudo maximum number of control points*/
#define PC_NrbKmax      20   /*Maximum order of the splines*/
```

## 9.2 Data Structures

The following are defined in `nurbtype.h`

```
typedef struct PR_nurb {            /* tensor product surface       */
  PR_dir                pf_u;       /* U and V knot vectors         */
  PR_dir                pf_v;
  PR_pts                *pf_ppp;    /* Pointer to control points    */
  struct PR_nurb        *pf_next;   /* Pointer to next in free list */
  struct PR_nurb        *pf_prev;   /* Pointer to prev in free list */
  Pint                  pf_id;      /* Integer ID for user use!     */
} PR_nurb;

typedef struct PR_dir {
  Pint                  pf_k;       /* order in u                   */
  Pint                  pf_n;       /* number of vertices           */
  Pint                  pf_nt;      /* number of knots  n+k         */
  PR_knots              *pf_kk;     /* Pointer to knots             */
} PR_dir;

typedef struct PR_knots {
  struct PR_knots       *pf_next;   /* Pointer to next one          */
  Pint                  pf_size;    /* Size of this one             */
  Pfloat                knots[1];   /* Array of the knots           */
} PR_knots;

typedef struct PR_pts {
  struct PR_pts         *pf_next;   /* Pointer to next one          */
  Pint                  pf_size;    /* Size of this one             */
  Ppoint4               pts[1];     /* Array of the Points          */
PR_pts;

typedef enum {PVudir, PVvdir} PE_dir;
```

# 10 Use of the Library

The `PR_nurb` data structure can be used either by static or dynamic allocation. The following examples serve to show this.

1. **Using static allocation**

   (a) Define each field in the NURBS data structure individually

```
PR_nurb any_nurb;
...
nrb_clear(&any_nurb);
/* Allocate values to each field in the NURBS data structure
*/
any_nurb.pf_u.pf_k  = order_of_any_nurb;
any_nurb.pf_u.pf_n  = number_of_control_points_of_any_nurb;
any_nurb.pf_u.pf_nt = number_of_knots_of_any_nurb;

/* The knots
*/
for(i=0;i<any_nurb.pf_x.pf_nt;i++)
   any_nurb.pf_x.pf_kk->knots[i] = knot_values[i];

/* Set the control points
*/
nrb_allocatepts(number_of_control_points_of_any_nurb,&any_nurb.pf_ppp);
for(i=0;i<number_of_control_points_of_any_nurb;i++)
   any_nurb.pf_ppp->pts[i]  = any_data_points[i];

/* Display the NURBS structure
*/
nrb_dump(stdout,&any_nurb);
...
```

   (b) Use routines to place values into the NURBS data structure

```
PR_nurb any_nurb;
...
nrb_clear(&any_nurb);
/* Allocate values to each field in the NURBS data structure
*/
nrb_init(order,number_of_control_points+order,0,1,&any_nurb);

/* The knots
*/
nrb_makeknots(0.0, 1.0, &any_nurb.pf_u);

/* Set the control points
*/
nrb_allocatepts(number_of_control_points,&any_nurb.pf_ppp);
for(i=0;i<number_of_control_points_of_any_nurb;i++)
   any_nurb.pf_ppp->pts[i]  = any_data_points[i];

/* Display the NURBS structure
*/
nrb_dump(stdout,&any_nurb);
...
```

(c) Use construction operators to build the NURBS data structure

```
PR_nurb any_nurb;
...
nrb_clear(&any_nurb);

/* Use a creation operator to fill the NURBS structure
*/
nrb_sphere(&any_nurb);
/* Display the NURBS structure
*/
nrb_dump(stdout,&any_nurb);
...
```

2. **Using dynamic allocation**

(a) Define each field in the NURBS data structure individually

```
PR_nurb *any_nurb;
...
any_nurb=NULL;
/* Allocate enough space for the NURB
*/
nrb_allocatenurb(&any_nurb)
/* Allocate values to each field in the NURBS data structure
*/
any_nurb->pf_u.pf_k  = order_of_any_nurb;
any_nurb->pf_u.pf_n  = number_of_control_points_of_any_nurb;
any_nurb->pf_u.pf_nt = number_of_knots_of_any_nurb;

/* The knots
*/
nrb_makeknots(0.0, 1.0, &any_nurb->pf_u);

/* Set the control points
*/
nrb_allocatepts(number_of_control_points_of_any_nurb,&any_nurb->pf_ppp);
for(i=0;i<number_of_control_points_of_any_nurb;i++)
   any_nurb->pf_ppp->pts[i]  = any_data_points[i];

/* Display the NURBS structure
*/
nrb_dump(stdout,any_nurb);
...
/* Deallocate the space used by the NURB
*/
nrb_deallocatenurb(&any_nurb)
```

(b) Use routines to place values into the NURBS data structure

```
PR_nurb *any_nurb;
...
any_nurb=NULL;
/* Allocate values to each field in the NURBS data structure
*/
nrb_init(order,number_of_control_points+order,0,1,any_nurb);

/* The knots
*/
```

```
nrb_makeknots(0.0, 1.0, any_nurb.pf_u);

/* Set the control points
*/
nrb_allocatepts(number_of_control_points_of_any_nurb,&any_nurb->pf_ppp);
for(i=0;i<number_of_control_points_of_any_nurb;i++)
   any_nurb->pf_ppp->pts[i]  = any_data_points[i];

/* Display the NURBS structure
*/
nrb_dump(stdout,any_nurb);
...
/* Deallocate the space used by the NURB
*/
nrb_deallocatenurb(&any_nurb)
```

(c) Use construction operators to build the NURBS data structure

```
PR_nurb *any_nurb;
...
any_nurb=NULL;

/* Use a creation operator to fill the NURBS structure
*/
nrb_sphere(any_nurb);
/* Display the NURBS structure
*/
nrb_dump(stdout,any_nurb);
...
/* Deallocate the space used by the NURB
*/
nrb_deallocatenurb(&any_nurb)
```

## 11 NURBS Application Level Procedures

The following procedures are available to the application programmer, and are defined in the header file `nurbfns.h` which is currently in the directory `temp`.

### nrb_allocateknots

void **nrb_allocateknots** ( PR_dir *kk )

Allocates space for the knots. It assumes the fields of `kk` have been assigned the appropriate values. If the pointer is not `NULL` that knot vector is added to the free list.

e.g.,

```
PR_nurb *nurb;
...
/* Set the order, number of control points and the number of knots
*/
nurb->pf_u.pf_k  = 4;
nurb->pf_u.pf_n  = 5;
nurb->pf_u.pf_nt = 9;
...
/* Allocate space for the knots in the u direction
*/
nrb_allocateknots(&nurb->pf_u);
```

### nrb_allocatenurb

void **nrb_allocatenurb** ( PR_nurb **nrb )

Creates a `PR_nurb` on the heap and makes `nrb` point to it.

e.g.,

```
PR_nurb *nurb;
...
nrb_clear(nurb);
...
/* Allocate space for the nurb.
*/
nrb_allocatenurb(nurb);
```

### nrb_allocatepts

void **nrb_allocatepts** ( Pint nx,
PR_pts **pp )

Creates space for `nx` homogeneous points and returns the pointer to them in `pp`. If the pointer is not `NULL` that point list is added to the free list.

e.g.,

```
PR_nurb *nurb;
...
/* Set the order, number of control points and the number of knots
*/
nurb->pf_u.pf_k  = 4;
nurb->pf_u.pf_n  = 5;
nurb->pf_u.pf_nt = 9;
...
/* Allocate space for the control points in the u direction
*/
nrb_allocatepts(nurb->pf_u.pf_n,&nurb->pf_ppp);
```

## nrb_boehmc

void **nrb_boehmc** ( PR_nurb *nrb1,
PR_dir *tnew,
PR_nurb *nrb2 )

Another knot insertion routine. Inserts the knots in `tnew` into each row (U direction) of `nrb1` returning the answer transposed in `nrb2`. It uses Boehm's [3] algorithm, and `nrb1` is unmodified.

e.g.,

```
PR_nurb nurb1,nurb2;
PR_dir new_knots;
...
/* Create some NURBS
*/
nrb_circle(&nurb1);
...
nrb_copy(&nurb1,&nurb2);
nurb2.pf_u.pf_nt = 20;
nurb2.pf_u.pf_n = nurb2.pf_u.pf_nt - nurb2.pf_u.pf_k;
...
/* Create a new set of knot vectors
*/
nrb_makeknots(0.0, 1.0, &nurb2.pf_u);
nrb_unionknots(&nurb1.pf_u, &nurb2.pf_u, &nurb2.pf_u);
nrb_differenceknots(&nurb1.pf_u,&nurb2.pf_u,&new_knots);
...
nrb_boehmc(&nurb1,&new_knots,&nurb2);
nrb_transpose(&nurb2);
...
nrb_dump(&nurb1);
```

## nrb_bvalue

void **nrb_bvalue** ( PR_nurb *nrb,
Pfloat x,
Pint jderiv,
PR_nurb *nrb2 )

Evaluate a NURBS at the value `u` of the $\texttt{jderiv}^{th}$ derivative of `nrb`.

e.g.,

```
PR_nurb triangle, evaluated_nurb;
float u=0.5, deriv=2;
...
/* Create a NURBS triangle
*/
nrb_triangle(&triangle);
/* Evaluate a NURBS at knot value u of deriv'th derivative of the NURB
*/
nrb_bvalue(&triangle, u, deriv, &evaluated_nurb);
```

## nrb_clear

void **nrb_clear** ( PR_nurb *nrb )

Assigns all the pointer fields to `NULL`. Used mainly by `nrb_allocatenurb`, to ensure the fields are properly initialized.

e.g.,

```
PR_nurb *nurb;
...
/* Make sure all the entries in the NURBS structure are pointing to NULL
*/
nrb_clear(nurb);
```

## nrb_copy

void **nrb_copy** ( PR_nurb *nrb1,
PR_nurb *nrb2 )

Makes `nrb2` a copy of `nrb1`. It allocates space for the knots and points and copies those as well.

e.g.,

```
PR_nurb circle1, circle2;
...
/* Create a NURBS representation of a circle
*/
nrb_circle(&circle1);
/* Make circle2 a copy of circle1
*/
nrb_copy(&circle1, &circle2);
```

## nrb_copyknots

void **nrb_copyknots** ( PR_dir *k1,
PR_dir *k2 )

Makes a `k2` a copy of `k1`. It allocates space for the knots, and copies them. Mainly used internally within the procedures.

e.g.,

```
PR_nurb *nurb1, *nurb2;
...
/* Set the order, number of control points and the number of knots
*/
nurb1->pf_u.pf_k  = 3;
nurb1->pf_u.pf_n  = 5;
nurb1->pf_u.pf_nt = 8;
...
/* Allocate space for the knots in the u direction
*/
nrb_allocateknots(&nurb1->pf_u);
nrb_makeknots(0.0, 1.0, &nurb1->pf_u);
/* Make nurb2's knots a copy nurb1's knots
*/
nrb_copyknots(&nurb1->pf_u, &nurb2->pf_u);
```

## nrb_deallocateknots

void **nrb_deallocateknots** ( PR_dir *kk )

Return a knot vector to the free list. Mainly used internally within the procedures.

e.g., used in functions such as `nrb_split`.

## nrb_deallocatenurb

void **nrb_deallocatenurb** ( PR_nurb **nrb )

Return a NURBS to the free list. Mainly used internally within the procedures.

e.g., used in `nrb_split` for instance.

## nrb_deallocatepts

void **nrb_deallocatepts** ( PR_pts **pp )

Return a point list to the free list. Mainly used internally within the procedures.

e.g., used in `nrb_elevate` for instance.

## nrb_differenceknots

void **nrb_differenceknots** ( PR_dir *k1,*k2,*k3 )

Create a knot sequence `k3` which contains those knots in `k2` which are NOT in `k1`. It assumes `k2.pf_nt` > `k1.pf_nt`.

e.g.,

```
PR_nurb *nurb1, *nurb2;
PR_dir *new_knots;
...
nurb1->pf_u.pf_nt = 8;
nurb2->pf_u.pf_nt = 10;
...
/* Create Knot vectors
*/
nrb_makeknots(0.0, 0.5, &nurb1->pf_u);
nrb_makeknots(0.5, 1.0, &nurb2->pf_u);
...
/* Difference the knot sequences from nurb1 and nurb2
*/
nrb_differenceknots(&nurb1->pf_u, &nurb2->pf_u, new_knots);
```

## nrb_du

void **nrb_du** ( PR_nurb *nrb,
Pfloat x,
PR_pts **pp )

Evaluate a NURBS at the value `x`. This function simply calls `nrb_bvalue` to evaluate the NURBS at the $1^{st}$ derivative.

e.g.,

```
PR_nurb triangle;
PR_pts *set_of_points;
float u=0.5;
...
/* Create a NURBS triangle
*/
nrb_triangle(&triangle);
/* Evaluate a NURBS at knot value u at the first derivative of the NURB
* returning a set of pts
*/
nrb_du(&triangle, u, &set_of_points);
```

## nrb_dump

void **nrb_dump** ( FILE *fp,
PR_nurb *nrb )

Dump a PR_nurb representation to file `fp`.

e.g.,

```
PR_nurb circle;
FILE *data_file;
...
/* Create a NURBS representation of a circle
*/
```

```
nrb_circle(&circle);
/* Dump the values within the NURBS structure
*/
if(dump_to_file)
{
   if((data_file=fopen("file.name","w"))==NULL)
      fprintf(stderr,"ERROR: Unable to open file\n");
   else
      nrb_dump(data_file, &circle);
}
else
   /* To the screen
   */
   nrb_dump(stdout, &circle);
```

## nrb_dumpknots

void **nrb_dumpknots** ( FILE *fp,
Pchar *str,
PR_dir kk )

Dump the knots to file `fp`.

e.g., use is similar to that of `nrb_dump`.

## nrb_dumplists

void **nrb_dumplists** ( FILE *fp )

Dump the freelists of points, knots and NURBS to file `fp`.

e.g., Useful for debugging purposes.

## nrb_dumpsummary

void **nrb_dumpsummary** ( FILE *fp,
PR_nurb *nrb )

Dump a summary of the PR_nurb representation to file `fp`. It is the equivalent to `nrb_dump` without the list of points.

e.g., Useful for debugging purposes.

## nrb_elevate

void **nrb_elevate** ( boolean transpose,
PR_nurb *nrb1, *nrb2 )

Elevates the degree of `nrb1` returning the answer in `nrb2`. Recognizing that this is more likely to be called for curves rather than surfaces, `transpose` controls whether or not `nrb2` is transposed.

e.g.,

```
PR_nurb nurb1, nurb2;
...
/*  Check to see if the NURBS is a curve
*/
if(nurb1.pf_v.pf_nt > 1)
{
   /* The NURBS is a surface, so elevate both directions of the NURB.
   */
   nrb_elevate(TRUE, &nurb1, &nurb2);
   nrb_elevate(TRUE, &nurb2, &nurb1);
}
else
{
   /* The NURBS is a curve, so only need to elevate the NURBS in
   * single direction
   */
   nrb_elevate(FALSE, &nurb1, &nurb2);
   nrb_copy(&nurb2, &nurb1);
}
```

## nrb_elevateknots

void **nrb_elevateknots** ( PR_dir *k1, *k2 )

Used by `nrb_elevate` it raises the degree of each of the knots in `k1` by one, but ensures that no knot is repeated more than `k1.pf_k+1`. Mainly used internally within the library.

e.g., used within `nrb_elevate` for instance.

## nrb_evaluate

void **nrb_evaluate** ( PR_nurb *nrb1,
Pfloat x,
PR_nurb *nrb2 )

Evaluate a NURBS at the value `x`. This function simply calls `nrb_bvalue` to evaluate the NURBS at the $0^{th}$ derivative.

e.g.,

```
PR_nurb triangle, evaluated_nurb;
float u=0.5;
...
/* Create a NURBS triangle
*/
nrb_triangle(&triangle);
/* Evaluate a NURBS at knot value u of the NURB
*/
nrb_evaluate(&triangle, u, &evaluated_nurb);
```

## nrb_extrude

void **nrb_extrude** ( PR_nurb *nrb1 ,
Ppoint3 w ,
PR_nurb *nrb2 ,
Pint *error )

Create `nrb2` a linear extrusion of `nrb1` along the vector `w`. Taken from Piegl [14].

e.g.,

```
PR_nurb circle, nurb;
int err;
...
nrb_circle(&circle);
/* Create a cylinder by extruding a circle
*/
nrb_extrude(&circle, ptk_vector3(0.0,0.0,2.0), &nurb, &err);
```

## nrb_getpoint4

void **nrb_getpoint4** ( FILE *fp ,
Ppoint4 *pt )

Input a homogenous point from file pointer. If successful returns `true`, `false` otherwise.

e.g., used in `nrb_read`,

```
PR_nurb nurb;
FILE *data_file;
...
if((data_file=fopen("file.name","r"))==NULL)
   fprintf(stderr,"ERROR: Unable to open file\n");
...
/* Input a list of control points
*/
for(i=0;i<number_of_control_points;i++)
   nrb_getpoint4(data_file,nurb.pf_ppp->pts[i]);
```

## nrb_index

Pint **nrb_index** ( Pint x, y, nx )

For efficiency reasons, the knots and control points are stored in 1D arrays. Given `nx` which is the number of knots or control points in x, this routine calculates the correct index into a conceptually 2D array. Mainly used internally within the procedures.

e.g., used in `nrb_interpolate` for instance.

## nrb_init

void **nrb_init** ( Pint kx,ntx,ky,nty ,
PR_nurb *nrb )

Initialize a NURBS using the given order and number of knots in x and y. Space for the knots and control points is allocated, though the contents of these arrays is undefined.

e.g.,

```
PR_nurb nurb;
...
nrb_clear(&nurb)
/* Initialize a nurb with a given set of values
*/
nrb_init(3, 13, 4, 24, &nurb);
```

## nrb_initialise

void **nrb_initialise**

Initialize the library. Should be called once at the beginning of your program.

e.g.,

```
main(int argc, char *argv[])
{
   /* Initialize the NURBS procedure library
   */
   nrb_intialise();
   ...
```

## nrb_inqerrorcount

Pint **nrb_inqerrorcount** ( Pint *i )

Returns the maximum number of errors that are reported. The default is 10.

Useful for debugging purposes.

## nrb_interchange

void **nrb_interchange** ( PR_nurb *nrb )

Interchange the `u` and `v` components of NURBS i.e., the control points, the knots and the order.

e.g., mainly used within internal routines such as `nrb_osloc` .

## nrb_interpolate

void **nrb_interpolate** ( PR_nurb *nrb1,
Pfloat tol,
PR_nurb *nrb2 )

Interpolate a NURBS to a given tolerance through a set of data points.

e.g.,

```
PR_nurb interpolating_nurb, nurb;
int number_of_points, order;
float tolerance=0.05;
...
nrb_clear(&interpolating_nurb);
nrb_clear(&nurb);
...
/* Define the NURBS to be interpolated i.e. order, etc. together
* with the data points to be interpolated through.
*/
nrb_init(order,number_of_points+order,0,1,&interpolating_nurb);
nrb_makeknots(0.0,1.0,&interpolating_nurb.pf_x);
nrb_allocatepts(number_of_points,&interpolating_nurb.pf_ppp);

/* Set the data to be interpolating through
*/
for(i=0;i<number_of_points;i++)
   dummy_path_nurb.pf_ppp->pts[i] = data_points[i];

/* Interpolate through the data points
*/
nrb_interpolate(&interpolating_nurb,tolerance,&nurb);
nrb_transpose(&nurb);
```

## nrb_interv

void **nrb_interv** ( PR_dir *kk,
Pfloat x,
Pint *left, *mflag )

Mainly used internally within the procedures.

e.g., used in `nrb_split` for instance.

## nrb_makeknots

void **nrb_makeknots** ( Pfloat tmin, tmax,
PR_dir *kk )

This routine produces `kk.pf_nt` knots between `tmin` and `tmax`, with `tmin` and `tmax` repeated `kk.pf_k` times, with the rest of the knots distributed uniformly.

e.g.,

```
PR_nurb *nurb;
...
/* Set the order, number of control points and the number of knots
*/
nurb->pf_u.pf_k  = 3;
nurb->pf_u.pf_n  = 5;
```

```
nurb->pf_u.pf_nt = 8;
...
/* Allocate space for the knots in the u direction
*/
nrb_allocateknots(&nurb->pf_u);
nrb_makeknots(0.0, 1.0, &nurb->pf_u);
```

## nrb_numsubdivs

Pint **nrb_numsubdivs** ( PR_nurb *nrb,
PE_dir xory,
Pfloat tol )

This routine determines the level of refinement to be applied to the NURB.

## nrb_osloc

void **nrb_osloc** ( PR_nurb *nrb1, *nrb2 )

Performs refinement on nrb1 and places the new NURBS in nrb2 [4].

e.g.,

```
PR_nurb nurb1,nurb2;
...
nurb1.pf_u.pf_k  = 3;
nurb1.pf_u.pf_n  = 5;
nurb1.pf_u.pf_nt = 8;
nurb2.pf_u.pf_k  = 4;
nurb2.pf_u.pf_n  = 6;
nurb2.pf_u.pf_nt = 10;
...
nrb_makeknots(0.0, 1.0, &nurb1.pf_u);
nrb_makeknots(0.0, 1.0, &nurb2.pf_u);
...
/* Create a new knot vector
*/
nrb_unionknots(&nurb2.pf_u, &nurb1.pf_u, &nurb1.pf_u);
/* Now insert the new control points corresponding to the
* new knot vector into the NURB
*/
nrb_osloc(&nurb2, &nurb1);
nrb_transpose(&nurb1);
...
nrb_dump(&nurb1);
```

## nrb_partials

void **nrb_partials** ( PR_nurb *nrb,
PR_pts **du, **dv, **duv )

Calculate the partial derivatives of the NURBS in `u`, `v` and `uv`.

e.g.,

```
PR_nurb *torus, *evaluated_nurb;
PR_pts *du_values, *dv_values, *duv_values;
...
/* Create a NURBS torus
*/
nrb_torus(torus);
/* Evaluate the partial derivatives of a NURBS and retrieve a set of points
* for each partial derivative.
*/
nrb_partials(torus, &du_values, &dv_values, &duv_values);
```

## nrb_putpoint4

void **nrb_putpoint4** ( FILE *fp,
Ppoint4 *pt )

Output a homogenous point to a file pointer.

e.g., used in `nrb_write`,

```
PR_nurb nurb;
FILE *data_file;
...
/* Create a NURB
*/
...
/* Output the list of control points
*/
for(i=0;i<number_of_control_points;i++)
   nrb_putpoint4(data_file,nurb.pf_ppp->pts[i]);
```

## nrb_read

boolean **nrb_read** ( FILE *fp,
PR_nurb *nrb )

Read a `PR_nurb` representation from a file, returning the answer in `nrb`. The function returns the value `true` if this was successful.

e.g.,

```
PR_nurb nurb;
FILE *data_file;
...
if((data_file=fopen("file.name","r"))==NULL)
   fprintf(stderr,"ERROR: Unable to open file\n");
...
/* Read in NURB
*/
if(!nrb_read(data_file,&nurb))
   fprintf(stderr,"ERROR: Unable read NURBS from file\n");
```

## nrb_revolve

void **nrb_revolve** ( PR_nurb *nrb1,
PR_nurb *nrb2 )

Do a surface of revolution about the z axis. Assumes curve is in x-z plane. Implemented as per Piegl [14]. This will use the (3D) projection of CURVE onto y= 0 plane.

e.g.,

```
PR_nurb circle, new_nurb;
Pvector3 point;
Pmatrix3 matrix;
...
/* Create a circle
*/
nrb_circle(&circle);
/* Displacement the circle
*  - shift in X
*  - rotate to XZ plane
*/
point = ptk_vector3(2.0, 0.0, 0.0);
ptk_shift3(&point, PREPLACE, matrix);
ptk_rotate3(90.0, 1, PPRECONCATENATE, matrix);
nrb_xform(&circle, matrix);
/* Revolve the circle to create a torus
*/
nrb_revolve(&circle, &new_nurb);
```

## nrb_ruled

void **nrb_ruled** ( PR_nurb *nrb1, *nrb2 ,
PR_nurb *nrb3 ,
Pint error )

Do a ruled surface between nrb1 and nrb2. Taken from Piegl [14].

e.g.,

```
PR_nurb square, circle, new_nurb;
Pvector3 point;
Pmatrix3 matrix;
int err;
...
/* Create some different objects
*/
nrb_square(&square);
nrb_circle(&circle);
/* Displacement the circle
*/
point = ptk_vector3(0.0,0.0,2.0);
ptk_shift3(&point, PREPLACE, matrix);
nrb_xform(&circle,matrix);
/* Create a ruled surface between the the two objects
```

```
*/
nrb_ruled(&circle, &square, &new_nurb, &err);
nrb_draws(&new_nurb)
```

## nrb_scaleknots

void **nrb_scaleknots** ( Pfloat new_tmin, new_tmax,
PR_dir *kk )

This routine rescales a knot vector to within new limits.

e.g.,

```
PR_nurb *nurb;
...
/* Set the order, number of control points and the number of knots
*/
nurb->pf_u.pf_k  = 3;
nurb->pf_u.pf_n  = 5;
nurb->pf_u.pf_nt = 8;
...
/* Allocate space for the knots in the u direction
*/
nrb_allocateknots(&nurb->pf_u);
nrb_makeknots(0.5, 2.0, &nurb->pf_u);
...
/* Re-scale the knots vector to 0.0 - 1.0
*/
nrb_scaleknots(0.0, 1.0, &nurb->pf_u);
```

## nrb_seterrorcount

void **nrb_seterrorcount** ( Pint i )

Sets the maximum number of errors that are reported. The default is 10.

Useful for debugging purposes.

## nrb_split

void **nrb_split** ( PR_nurb *nrb ,
Pfloat t ,
PR_nurb *nrb1, *nrb2 )

Split a NURBS in two.

e.g.,

```
PR_nurb circle, part1_nurb, part2_nurb;
float split_point;
Pvector3 point;
Pmatrix3 matrix;
int err;
...
nrb_circle(&circle) ;
/* Calculate a split point in the knot sequence
*/
split_point=xx;
/* Split NURBS into two NURBS objects
*/
nrb_split(&circle, split_point, &part1_nurb, &part2_nurb);
```

## nrb_stats

void **nrb_stats** ( FILE *fp )

Prints out statistics concerning the free lists to file `fp`.

Useful for debugging purposes.

## nrb_subdivc

void **nrb_subdivc** ( PR_nurb *nrb1, *nrb2,
Pint mu, j )

Calculates the control points corresponding to new knots—called by `nrb_osloc`.

## nrb_tessalate

void **nrb_tessalate** ( PR_nurb *nrb1,
Pint nx,
PR_nurb *nrb2 )

Increase the refinement of the nrb1 by nx, by inserting new knots. The routine transposes the NURB.

This routine could also be used to evaluate a NURBS to a given tolerance returning a list of points on the NURBS in the form of the control points.

e.g.,

```
PR_nurb nurb1, nurb2, new_nurb;
...
/* Create a NURB
*/
...
/*  Check to see if the NURBS is a curve
*/
if(nurb1.pf_v.pf_nt > 1)
{
   /* The NURBS is a surface, so tessalate a NURBS in both the
```

```
      * U and V directions
      */
      nrb_tessalate(&nurb1, 20, &nurb2);
      nrb_tessalate(&nurb2, 20, &new_nurb);
   }
   else
   {
      /* The NURBS is a curve, so only need to tessalate the NURBS in
      * single direction
      */
      nrb_tessalate(&nurb1, 20, &new_nurb);
      nrb_transpose(&new_nurb);
   }
```

## nrb_transpose

void **nrb_transpose** ( PR_nurb *nrb )

Transpose a NURBS in the x and y directions.

e.g.,

```
   PR_nurb nurb1, temp_nurb;
   ...
   /* As a result of this degree elevation the NURBS is transposed
   */
   nrb_elevate(TRUE,&nurb1,&temp_nurb);
   /* This transpose corrects the previous transposition, (transpose U and V)
   */
   nrb_transpose(&temp_nurb);
   /* Copy the nurb back to the original one
   */
   nrb_copy(&temp_nurb,&nurb1);
```

## nrb_unionknots

void **nrb_unionknots** ( PR_dir *k1,*k2,*k3 )

Create a union of knot sequences.

e.g.,

```
   PR_nurb nurb1, nurb2;
   ...
   nurb1.pf_u.pf_k  = 3;
   nurb1.pf_u.pf_n  = 5;
   nurb1.pf_u.pf_nt = 8;
   nurb2.pf_u.pf_k  = 4;
   nurb2.pf_u.pf_n  = 6;
   nurb2.pf_u.pf_nt = 10;
   ...
   nrb_makeknots(0.0, 1.0, &nurb1.pf_u);
```

```
nrb_makeknots(0.0, 1.0, &nurb2.pf_u);
...
/* Create a new knot vector
*/
nrb_unionknots(&nurb2.pf_u, &nurb1.pf_u, &nurb1.pf_u);
...
nrb_dump(&nurb1);
```

## nrb_write

void **nrb_write** ( FILE *fp,
PR_nurb *nrb )

Write a PR_nurb representation to a file.

e.g.,

```
PR_nurb nurb;
FILE *data_file;
...
nrb_torus(&nurb);
...
if((data_file=fopen("file.name","w"))==NULL)
   fprintf(stderr,"ERROR: Unable to open file\n");
...
/* Write NURB
*/
nrb_write(data_file,&nurb);
```

## nrb_xform

void **nrb_xform** ( PR_nurb *nrb,
Pmatrix3 xx )

Transforms the control points of `nrb` using matrix `xx`, and returns them in `nrb`.

e.g.,

```
PR_nurb circle, new_nurb;
Pvector3 point;
Pmatrix3 matrix;
...
/* Create a circle
*/
nrb_circle(&circle);
/* Displacement the circle
*  - shift in X
*  - rotate to XZ plane
*/
point = ptk_vector3(2.0, 0.0, 0.0);
ptk_shift3(&point, PREPLACE, matrix);
ptk_rotate3(90.0, 1, PPRECONCATENATE, matrix);

/* Perform the actual transformation on the NURBS according to the matrix
*/
nrb_xform(&circle, matrix);
```

### 11.1 Creation Operators

nrb_arc

void **nrb_arc** ( Pfloat theta1, theta2 ,
PR_nurb *nrb )

Create an arc of a circle centered at (0,0,0), with radius 1 in XY plane. The arc starts and finishes at `theta1`, and `theta2` degrees respectively.

e.g.,

```
PR_nurb nurb;
float arc_begin=0.0, arc_end=90.0;
...
nrb_clear(&nurb);
/* Allocate space for the knots in the u direction
*/
nrb_arc(arc_begin, arc_end, &nurb);
```

nrb_circle

void **nrb_circle** ( PR_nurb *nrb )

Make a unit circle in the XY plane, with it's centre at (0,0,0).

e.g.,

```
PR_nurb circle;
...
/* Create a NURBS representation of a circle
*/
nrb_circle(&circle);
```

nrb_polyline

void **nrb_polyline** ( Pint npts ,
Ppoint3 *pp ,
PR_nurb *nrb )

Generate the NURBS representation of a line.

e.g.,

```
PR_nurb nurb;
int order, number_of_points;
float *pp;
...
/* Initialize a nurb with a give set of values
*/
nrb_init(order,number_of_points+order,0,1,&nurb);
```

```
/* Set the data for the polyline
*/
for(i=0;i<number_of_points;i++)
   pp[i]=data_points[i];
/* Create the polyline
*/
nrb_polyline(number_of_points, pp, &nurb);
```

## nrb_torus

void **nrb_torus** ( PR_nurb *nrb )

Generate the NURBS representation of a torus about the origin.

e.g.,

```
PR_nurb torus;
...
/* Create a NURBS representation of a torus
*/
nrb_torus(&torus);
```

## nrb_triangle

void **nrb_triangle** ( PR_nurb *nrb )

Generate the NURBS representation of a triangle about the origin.

e.g.,

```
PR_nurb triangle;
...
/* Create a NURBS representation of a triangle
*/
nrb_triangle(&triangle);
```

## nrb_square

void **nrb_square** ( PR_nurb *nrb )

Create a NURBS representation of a square about the origin.

e.g.,

```
PR_nurb square;
...
/* Create a NURBS representation of a square
*/
nrb_square(&square);
```

### nrb_sphere

void **nrb_sphere** ( PR_nurb *nrb )

Generate the NURBS representation of a sphere about the origin.

e.g.,

```
PR_nurb sphere;
...
/* Create a NURBS representation of a sphere
*/
nrb_sphere(&sphere);
```

### nrb_vase

void **nrb_vase** ( PR_nurb *nrb )

Generate the NURBS representation of a vase about the origin.

e.g.,

```
PR_nurb vase;
...
/* Create a NURBS representation of a vase
*/
nrb_vase(&vase);
```

## 11.2 Drawing Procedures

The following drawing procedures are implemented using DEC PHIGS and are provided for completeness.

### nrb_drawc

void **nrb_drawc** ( PR_nurb *nrb,
Pfloat t,
PE_dir xory )

Draw a NURBS curve together with it control polygon.

e.g.,

```
PR_nurb circle;
...
/* Create a NURBS representation of a circle
*/
nrb_circle(&circle);
/* Open PHIGS structure
*/
popenstruct(NURB_STRUC);
   /* Draw the circle
   */
   nrb_drawc(&circle,0.05,PVudir);
```

## nrb_drawcnet

void **nrb_drawcnet** ( PR_nurb *nrb,
Pint start,
PE_dir xory )

Draw a NURBS curve's defining control polygon.

e.g.,

```
PR_nurb circle;
...
/* Create a NURBS representation of a circle
*/
nrb_circle(&circle);
/* Open PHIGS structure
*/
popenstruct(NURB_STRUC);
   /* Draw the circle's control polygon
   */
   nrb_drawcnet(&circle, 1, PVudir);
```

## nrb_draws

void **nrb_draws** ( PR_nurb *nrb )

Draw a NURBS surface together with it control polygon.

e.g.,

```
PR_nurb torus;
...
/* Create a NURBS representation of a torus
*/
nrb_torus(&torus);
/* Open PHIGS structure
*/
popenstruct(NURB_STRUC);
   /* Draw the torus
   */
   nrb_draws(&torus);
```

## nrb_drawsnet

void **nrb_drawsnet** ( PR_nurb *nrb )

Draw a NURBS surface's defining control polygon.

e.g.,

```
PR_nurb torus;
...
/* Create a NURBS representation of a torus
*/
nrb_torus(&torus);
/* Open PHIGS structure
*/
popenstruct(NURB_STRUC);
   /* Draw the torus
   */
   nrb_drawsnet(&torus);
```

## nrb_phigsplus

void **nrb_phigsplus** ( PR_nurb *nrb )

This routine actually draws the NURBS using PHIGS PLUS primitives. It is provided for completeness. If another application programming interface is used this routine must be rewritten.

e.g.,

```
PR_nurb nurb;
...
/* Construct a NURB
*/
nrb_init(order_u,number_knots_u,order_v,number_knots_v,&nurb);
nrb_makeknots(0.0,1.0,&nurb.pf_u);
nrb_makeknots(0.0,1.0,&nurb.pf_v);
nrb_allocatepts(npt,&nurb.pf_ppp);
/* Set the control points
*/
...
/* Open PHIGS structure
*/
popenstruct(NURB_STRUC);
   /* Draw the NURB
   */
   nrb_phigsplus(&nurb);
```

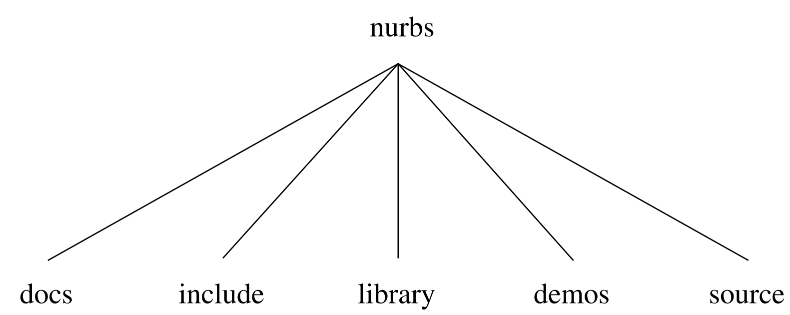


Figure 6: NURBS library directory structure.

# 12 Installation

There are two different methods of installation of the procedure library depending on the type of machine to be installed on. Both methods create the directory structure as shown in figure 6.

The header files may be copied into the system header file directory or left as is. If the files are copied into the system area then the various `makefiles` must be updated.

Currently, the NURBS library is built in the form of a shared executable library for VMS and an object library for UNIX. This to can be built and installed into the appropriate system area, but at present is created in `nrb_library`. If the library is to be moved into the system area then again the `makefiles` must be updated.

## 12.1 For UNIX

The library is provided in the form of a compressed tar archive. Once the archive has been obtained.

1. Uncompress the archive by typing

   ```
   > uncompress nurb.tar.Z
   ```

2. Untar the archive

   ```
   > tar xvf nurb.tar target-directory
   ```

   This will extract the library from the `target-directory` creating the correct directory structure for the library and placing the files in the correct place below the `target-directory`.

## 12.2 For VMS

The library is provided in the form of a backup save set. Once the archive has been obtained.

1. Restore the files from the save set in to an appropriate place in the directory structure.

   ```
   > BACKUP NURB.BCK/SAVE_SET [target-directory...]
   ```

   This will restore the library correctly into the VMS directory structure, correctly placing the files in the correct places.

### 12.3 Building the Library

For both VMS and UNIX systems it is assumed that LaTeX, `bibtex` and `idxtex` are available to produce the documentation.

#### 12.3.1 UNIX

A number of `makefiles` exist to build the various parts of the library. Each `makefile` is called by the main `makefile` which is located at the top of the NURBS library structure, in `nurb`.

- To Build the Entire Distribution

  ```
  > make all
  ```

- To Build the Documentation

  ```
  > make docs
  ```

- To Build the Library

  ```
  > make library
  ```

#### 12.3.2 VMS

A single build file exists in the form of a VMS command procedure. This is located at the top of the NURBS library directory structure.

Alias a symbol to execute the command procedure.

```
> build :== @makefile
```

- To Build the Entire Distribution

  ```
  > build all
  ```

- To Build the Documentation

  ```
  > build docs
  ```

- To Build the Library

  ```
  > build library
  ```

## 13 Example Programs

Provided with this distribution are a number of example programs to demonstrate the use and also test the procedure library. These programs are located in the `nrb_demos` directory.

- `nrbtst00.c` Simple program to create a NURB
- `nrbtst01.c` Test transformations

- `nrbtst02.c` Create a ruled surface
- `nrbtst03.c` Test routine `nrb_bvalue`
- `nrbtst04.c` Test routine `nrb_oslo`
- `nrbtst05.c` Test routine `nrb_boehmc`
- `nrbtst06.c` Create a simple surface
- `nrbtst07.c` Test routine `nrb_interpolate`
- `nrbtst08.c` Test routine `nrb_interpolate`
- `nrbdemo1.c` Test routine `nrb_extrude`
- `nrbdemo2.c` Interactive demonstration program written in DEC PHIGS
- `trantest.c` Test the TRAN module

Note that since the NURBS routines make use of a separate object which is derived from the PHIGS Toolkit project, also available from Kent, this to included in this version of the distribution.

The following is an example program `nrbdemo1.c`, showing how to make use of the NURBS procedure described above.

```
/* nrbdemo1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <ptk.h>

/* Declarations */
PR_nurb c1;                             /* NURB curve                   */
PR_nurb s1;                             /* NURB surface                 */
Pvector3 v1;                            /* Scaling Vector               */

Pint ee;                                /* Extrusion Error              */

Pmatrix3 xx;                            /* Tranformation Matrix         */

main()
{
  printf("Testing NURBS\n");

  nrb_initialise;                       /* Start up NURBS library       */

  nrb_clear(&c1);                       /* Make sure the data structures */
  nrb_clear(&s1);                       /* are empty                    */

  nrb_circle(&c1);                      /* Make c1 a circle             */

  v1 =ptk_vector3(100.0,100.0,100.0);
  ptk_scale3(&v1,PREPLACE,xx);
  nrb_xform(&c1,xx);                    /* Transform the unit circle, so */
                                        /* it is 100 units in radius    */
```

```
  nrb_dump(stdout,&c1);                  /* Print it out the contents of  */
                                         /* the NURB structure            */
  printf("Unit Cylinder\n");
  v1=ptk_vector3(0.0,0.0,100.0);
  nrb_extrude(&c1,v1,&s1,&ee);          /* Convert it to a cylinder by   */
                                         /* extrusion                     */
  nrb_dump(stdout,&s1);                  /* Print it out the contents of  */
                                         /* the NURB structure            */

  nrb_stats(stdout);                     /* Print out some statistics     */
  nrb_dumplists(stdout);

  printf("End of NURBs test.\n");
}
```

## 13.1 Interactive NURBS Demonstration

The program `nrbdemo2.c` is an interactive demonstration of the NURB procedure library. The program is written using DEC PHIGS version 2.2 which supports PHIGS PLUS primitives, the program could be ported to other versions of PHIGS PLUS quite easily. It also gives a good example of how to use the procedure library.

The program uses the procedure library to generate a number of different NURBS objects which can be manipulated by routines in the library.

The objects are created by using the creation and constructions operators.

- Torus.
- Cylinder.
- Cone.
- Sphere.
- Cuboid.
- Tetrahedron.
- Triangular Prism.
- Vase.
- Circle.
- Square.
- Triangle.
- Ruled Surface.

The operations on the NURBS objects give an indication of how easy the procedure library is to use.

- Elevate a NURB,
- Transpose a NURB,
- Refine a NURBS knot vector,

as well as a number of interactive functions which include rotation and scaling.

# 14 Compiling and Linking

There are two different ways in which to build the program depending on the operating system. In both cases a number of compiler directives must be issued in order to build the programs correctly.

The NURB library is designed to coexist with the PHIGS Toolkit (PTK) and as such share a number of common header files. There are three cases inwhich both the NURB library and the Toolkit can exist.

1. Just PTK
   PTK and no NURB, no action need be taken
2. Just NURBS
   NURB and no PTK, define NURB
3. NURB and PTK
   Define PTKNURB

## 14.1 Build for VMS

If the NURBS shared executable library has not been built and placed into the system area then the `linker` must be told explicitly where the library is located. This is done be defining a `logical symbol` for the library, indicating the full path to the library.

e.g.,

```
DEFINE NURBS DISK$USERS_C:[YIP.NURBS.LIBRARY]NURB.EXE
```

Compile with:

```
CC/INCLUDE_DIRECTORY=[-.INCLUDE] NRBDEMO1
```

Link with:

```
LINK NRBDEMO1,NURBS/OPT,VAXCRTL/OPT
```

and run with

```
RUN NRBDEMO1
```

The option files, `NURBS.OPT` and `VAXCRTL.OPT` contain details of the locations of the shared libraries that must be linked with the programs.

```
NURBS.OPT:

DISK$USERS_C:[YIP.NURBS.LIBRARY]NURB.EXE/SHARE
```

```
VAXCRTL.OPT:

SYS$LIBRARY:VAXCRTL.EXE/SHARE
```

### 14.2 Build for UNIX

Compile with:

```
cc -c nrbdemo.o nrbdemo1.c -I../library
```

Link with:

```
cc -o nrbdemo1 nrbdemo1.o ../library/libnurb.a -lm
```

and run with

```
nrbdemo1
```

All the demonstration and test programs in the `nrb_demos` directory can be compiled and linked in a similar manner.

## References

[1] R E Barnhill and R F Riesenfeld. *Computer-Aided Geometric Design*. Academic Press, 1974.

[2] P Bezier. *The Mathematical Basis of the UNISURF CAD System*. Butterworths, 1986.

[3] Wolfgang Boehm. Inserting New Knots into B-spline Curves. *Computer-Aided Design*, 12(4):199–201, July 1988.

[4] E Coehn, T Lyche, and L L Schumaker. Algorithms for degree-raising of splines. *ACM Transactions on Graphics*, pages 171–181, July 1985.

[5] C de Boor. *A Practical Guide to Splines*. Springer-Verlag, 1978.

[6] G Farin. *Geometric Modeling: Algorithms and New Trends*. SIAM, 1400 Architects Building, 117 South 17th Street, Philadelphia, Pensylvania 19103-5052 USA, 1987.

[7] G Farin. *Curves and Surfaces for Computer-Aided Geometric Design*. Academic Press, 1988.

[8] P J Hartley and C J Judd. Parameterization of bezier type b-spline curves and surfaces. *Computer Aided Design*, pages 130–134, March 1978.

[9] P J Hartley and C J Judd. Parametrization and shape of b-spline curves for cad. *Computer Aided Design*, pages 235–238, September 1980.

[10] P Lancaster and K Salkauskas. *Curve and Surface Fitting*. Academic Press, 1986.

[11] L A Piegl. Modifying the shape of rational b-splines. part 1: Curves. *Computer Aided Design*, pages 509–519, October 1989.

[12] L A Piegl. Modifying the shape of rational b-splines. part 2: Surfaces. *Computer Aided Design*, pages 538–547, November 1989.

[13] L A Piegl. On NURBS: A Survey. *IEEE Computer Graphics & Applications*, 11(1):55–71, January 1991.

[14] L A Piegl and W Tiller. Curve and Surface construction using rational B-splines. *Computer-Aided Design*, 19(9):485–498, November 1987.

[15] H Prautzsch. Degree elevation of B-spline curves. *Computer Aided Geometric Design*, pages 193–198, 1984.

[16] L Schumaker. *Spline Functions Basic Theory*. Wiley, 1981.

[17] M Shantz and S L Chang. Rendering trimmed nurbs with adaptive forward differencing. *Computer Graphics*, 1989.

[18] W Strasser and H-P Seidel, editors. *Theory and Practice of Geometric Modelling*. Springer-Varlag, 1989.

[19] K J Versprille. *Computer-Aided Design Applications of the Rational B-Spline Approximation Form*. PhD thesis, Syracuse University, 1975.

[20] F Yamaguchi. *Curves and Surfaces for Computer-Aided Geometric Design*. Springer-Verlag, 1988.

# Index


— **C** —

Compiling and linking, 45–46
Constants and Records, 15
Curve definition, 6
Curve evaluation, 10
Curves, 5

— **D** —

Data structures, 15
Degree elevation, 10
Drawing NURBS, 14

— **E** —

Evaluation, 10

— **H** —

Homogeneous coordinates, 5

— **I** —

Installation for UNIX, 41
Installation for VMS, 41–42
Installation procedure, 41
Interpolation, 10

— **K** —

Knot vector, 6
Knot vectors, 6

— **L** —

Library installation, 42

— **N** —

Normalized B-spline basis, 6
nrb_allocateknots, 19
nrb_allocatenurb, 19, 21
nrb_allocatepts, 19
nrb_arc, 8, 36
nrb_boehmc, 20
nrb_bvalue, 11, 20, 23, 25
nrb_circle, 8, 36
nrb_clear, 21
nrb_copy, 21
nrb_copyknots, 21
nrb_deallocateknots, 22
nrb_deallocatenurb, 22
nrb_deallocatepts, 22
nrb_differenceknots, 22
nrb_drawc, 14, 38
nrb_drawcnet, 14, 39
nrb_draws, 14, 39
nrb_drawsnet, 14, 39
nrb_du, 11, 23
nrb_dump, 23–24
nrb_dumpknots, 24
nrb_dumplists, 24
nrb_dumpsummary, 24
nrb_elevate, 10, 22, 24–25
nrb_elevateknots, 25
nrb_evaluate, 10, 25
nrb_extrude, 11, 26
nrb_getpoint4, 26
nrb_index, 26
nrb_init, 26
nrb_initialise, 27
nrb_interchange, 13, 27
nrb_interpolate, 10, 26–27
nrb_interv, 28
nrb_makeknots, 28
nrb_numsubdivs, 29
nrb_osloc, 27, 29, 33
nrb_partials, 29
nrb_phigsplus, 14, 40
nrb_polyline, 36
nrb_putpoint4, 10–11, 30
nrb_read, 26, 30
nrb_revolve, 8, 12, 31
nrb_ruled, 12, 31
nrb_scaleknots, 32
nrb_sphere, 8, 38
nrb_split, 11, 22, 28, 32
nrb_square, 8, 37
nrb_stats, 33
nrb_subdivc, 33
nrb_tessalate, 33
nrb_torus, 8, 37
nrb_transpose, 13, 34
nrb_triangle, 8, 37
nrb_unionknots, 34
nrb_vase, 8, 38
nrb_write, 30, 35
nrb_xform, 13, 35

### — P —

Programming example, 42

### — S —

Splitting NURBS, 11
Surface evaluation, 11
Surface operators, 11
Surfaces, 6

### — T —

Transformations, 13
Transposition, 13