Version 6.0 User's Manual

A Solid modeling Program

EMail: gershon@cs.technion.ac.il

Join *IRIT* mailing list: gershon@cs.technion.ac.il
Mailing list: irit-mail@cs.technion.ac.il
Bug reports: irit-bugs@cs.technion.ac.il

This manual is for IRIT Version 6.0.

Contents

1 Introduction

IRIT is a solid modeler developed for educational purposes. Although small, it is now powerful enough to create quite complex scenes.

IRIT started as a polygonal solid modeler and was originally developed on an IBM PC under MSDOS. Version 2.0 was also ported to X11 and version 3.0 to SGI 4D systems. Version 3.0 also includes quite a few free form curves and surfaces tools. See the UPDATE.NEW file for more detailed update information. In Version 4.0, the display devices were enhanced, freeform curves and surfaces have further support, functions can be defined, and numerous improvement and optimizations are added.

2 Copyrights

IRIT is a freeware solid modeler. It is not public domain since I hold copyrights on it. However, unless you are to sell or attempt to make money from any part of this code and/or any model you made with this solid modeler, you are free to make anything you want with it.

IRIT can be compiled and executed on numerous Unix systems as well as OS2, Windows NT and AmigaDOS. However, beware the MSDOS support is fading away.

You are not obligated to me or to anyone else in any way by using *IRIT*. You are encouraged to share any model you made with it, but the models you made with it are *yours*, and you have no obligation to share them. You can use this program and/or any model created with it for non commercial and non profit purposes only. An acknowledgement on the way the models were created would be nice but is *not* required.

3 Command Line Options and Set Up

The *IRIT* program reads a file called **irit.cfg** each time it is executed. This file configures the system. It is a regular text file with comments, so you can edit it and properly modify it for your environment. This file is being searched for in the directory specified by the IRIT_PATH environment variable. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. Note IRIT_PATH must terminate with '/'. If the variable is not set only the current directory is being searched for **irit.cfg**.

In addition, if it exists, a file by the name of **iritinit.irt** will be automatically executed before any other '.irt' file. This file may contain any *IRIT* command. It is the proper place to put your predefined functions and procedures if you have some. This file will be searched much the same way **IRIT.CFG** is. The name of this initialization file may be changed by setting the StartFile entry in the configuration file. This file is far more important starting at version 4.0, because of the new function and procedure definition that has been added, and which is used to emulate BEEP, VIEW, and INTERACT for example.

The solid modeler can be executed in text mode (see the .cfg and the -t flag below) on virtually any system with a C compiler.

Under all systems the following environment variables must be set and updated:

path	Add to path the directory where *IRIT*'s binaries are.
IRIT_PATH	Directory with config., help and *IRIT*'s binary files.
IRIT_DISPLAY	The graphics driver program/options. Must be in path.
IRIT_BIN_IPC	If set, uses binary Inter Process Communication.

For example,

```
set path = ($path /u/gershon/irit/bin)
setenv IRIT_PATH  /u/gershon/irit/bin/
setenv IRIT_DISPLAY "xgldrvs -s-"
setenv IRIT_BIN_IPC 1
```

to set /u/gershon/irit/bin as the binary directory and to use the sgi's gl driver. If IRIT_DISPLAY is not set, the server (i.e., the *IRIT* program) will prompt and wait for you to run a client (i.e., a display driver). if IRIT_PATH is not set, none of the configuration files, nor the help file will be found.

If IRIT_BIN_IPC is not set, text based IPC is used, which is far slower. No real reason not to use IRIT_BIN_IPC, unless it does not work for you.

In addition, the following optional environment variables may be set.

IRIT_MALLOC	If set, apply dynamic memory consistency testing. Programs will execute much slower in this mode.
IRIT_MALLOC_PTR	Set to a pointer address and the program will scream once this pointer is allocated.
IRIT_NO_SIGNALS	If set, no signals are caught by IRIT.
IRIT_SERVER_HOST	Internet Name of IRIT server (used by graphics driver).
IRIT_SERVER_PORT	Used internally to the TCP socket number. Should not be set by users.

For example,

```
setenv IRIT_MALLOC 1
setenv IRIT_MALLOC_PTR 1234567890
setenv IRIT_NO_SIGNALS 1
setenv IRIT_SERVER_HOST irit.cs.technion.ac.il
```

IRIT_MALLOC is useful for programmers, or when reporting a memory fatal error occurrence. IRIT_NO_SIGNALS is also useful for debugging when contorl-C is used within a debugger. The IRIT_SERVER_HOST/PORT controls the server/client (*IRIT*/Display device) communication.

IRIT_SERVER_HOST and IRIT_SERVER_PORT are used in the unix and Window NT ports of *IRIT*.

See the section on the graphics drivers for more details.

A session can be logged into a file as set via LogFile in the configuration file. See also the LOGFILE command.

The following command line options are available:

```
IRIT [-t] [-z] [file.irt]
```

-t	Puts *IRIT* into text mode. No graphics will be displayed and the display commands will be ignored. Useful when one needs to execute an irt file to create data on a tty device...
-z	Prints usage message and current configuration/version information.
file.irt	A file to invoke directly instead of waiting to input from stdin.

3.1 IBM PC OS2 Specific Set Up

Under OS2 the IRIT_DISPLAY environment variable must be set (if set) to os2drvs.exe without any option (-s- will be passed automatically). os2drvs.exe must be in a directory that is in the PATH environment variable. IRIT_BIN_IPC can be used to signal binary IPC which is faster. Here is a complete example:

```
set IRIT_PATH=c:\irit\bin\
set IRIT_DISPLAY=os2drvs -s-
set IRIT_BIN_IPC=1
```

assuming the directory specified by IRIT_PATH holds the executables of IRIT and is in PATH.

If IRIT_BIN_IPC is not set, text based IPC is used which is far slower. No real reason not to use IRIT_BIN_IPC unless it does not work for you.

3.2 IBM PC Window NT Specific Set Up

The NT port uses sockets and is, in this respect, similar to the unix port. The envirnoment variables IRIT_DISPLAY, IRIT_SERVER_HOST, IRIT_BIN_IPC should all be set in a similar way to the Unix specific setup. As a direct result, the server (IRIT) and the display device may be running on different hosts. For example the server might be running on an NT system while the display device will be running on an SGI4D exploiting the graphic's hardware capabilities. Here is a complete example:

```
set IRIT_PATH=c:\irit\bin\
set IRIT_DISPLAY=wntgdrvs -s-
set IRIT_BIN_IPC=1
```

3.3 Unix Specific Set Up

Under UNIX using X11 (x11drvs driver) add the following options to your .Xdefaults. Most are self explanatory. The Trans attributes control the transformation window, while the View attributes control the view window. SubWin attributes control the subwindows within the Transformation window.

#if COLOR	
irit*Trans*BackGround:	NavyBlue
irit*Trans*BorderColor:	Red
irit*Trans*BorderWidth:	3
irit*Trans*TextColor:	Yellow
irit*Trans*SubWin*BackGround:	DarkGreen
irit*Trans*SubWin*BorderColor:	Magenta
irit*Trans*Geometry:	=150x500+500+0
irit*Trans*CursorColor:	Green
irit*View*BackGround:	NavyBlue
irit*View*BorderColor:	Red
irit*View*BorderWidth:	3
irit*View*Geometry:	=500x500+0+0
irit*View*CursorColor:	Red
irit*MaxColors:	15
#else	
irit*Trans*Geometry:	=150x500+500+0
irit*Trans*BackGround:	Black
irit*View*Geometry:	=500x500+0+0
irit*View*BackGround:	Black
irit*MaxColors:	1
#endif	

4 First Usage

Commands to *IRIT* are entered using a textual interface, usually from the same window the program was executed from.

Some important commands to begin with are,

1. include("file.irt"); - will execute the commands in file.irt. Note include can be recursive up to 10 levels. To execute the demo (demo.irt) simply type 'include("demo.irt");'. Another way to run the demo is by typing demo(); which is a predefined procedure defined in iritinit.irt.

2. help(""); - will print all available commands and how to get help on them. A file called irit.hlp will be searched as irit.cfg is being searched (see above), to provide the help.

3. exit(); - close everything and exit *IRIT*.

Most operators are overloaded. This means that you can multiply two scalars (numbers), or two vectors, or even two matrices, with the same multiplication operator (*). To get the on-line help on the operator '*' type 'help("*");'

The best way to learn this program (like any other program...) is by trying it. Print the manual and study each of the commands available. Study the demo programs (*.irt) provided as well.

The "best" mode to use irit is via the emacs editor. With this distribution an emacs mode for irit files (irt postfix) is provided (irit.el). Make your .emacs load this file automatically. Loading file.irt will switch emacs into Irit mode that supports the following three keystrokes:

Meta-E	Executes the current line
Meta-R	Executes the current Region (Between Cursor and Mark)
Meta-S	Executes a single line from input buffer

The first time one of the above keystrokes is hit, emacs will fork an Irit process so that Irit's stdin is controlled via the above commands. This emacs mode was tested under various unix environments and under OS2 2.x.

5 Data Types

These are the Data Types recognized by the solid modeler. They are also used to define the calling sequences of the different functions below:

ConstantType	Scalar real type that cannot be modified.
NumericType	Scalar real type.
VectorType	3D real type vector.
PointType	3D real type point.
CtlPtType	Control point of a freeform curve or surface.
MatrixType	4 by 4 matrix (homogeneous transformation matrix).
PolygonType	Object consists of polygons.
PolylineType	Object consists of polylines.
CurveType	Object consists of curves.
SurfaceType	Object consists of surfaces.
TrimSrfType	Object consists of trimmed surfaces.
TrivarType	Object consists of trivariate function.
GeometricType	One of Polygon/lineType, CurveType, SurfaceType, TrimSrfType, TrivarType.
GeometricTreeType	A list of GeometricTypes or GeometricTreeTypes.
StringType	Sequence of chars within double quotes - "A string". Current implementation is limited to 80 chars.
AnyType	Any of the above.
ListType	List of (any of the above type) objects. List size is dynamically increased, as needed.

Although points and vectors are not the same, *IRIT* does not destinguish between them, most of the time. This might change in the future.

6 Commands summary

These are all the commands and operators supported by the *IRIT* solid modeler:

+	CEDITPT	CSURFACE	INTERACT	ROTZ	STRIVAR
−	CEVAL	CTANGENT	IRITSTATE	ROTZ2V,	SURFPREV
*	CEXTREMES	CTLPT	LIST	ROTZ2V2,	SURFREV
/	CHDIR	CTRIMSRF	LN	RULEDSRF	SWEEPSRF
^	CINFLECT	CYLIN	LOAD	SAVE	SWPSCLSRF
=	CINTERP	CZEROS	LOFFSET	SBEZIER	SYMBCPROD
==	CIRCLE	DSTPTLN	LOG	SBSPLINE	SYMBDIFF
!=	CIRCPOLY	DSTPTPLN	LOGFILE	SCALE	SYMBDPROD
<	CLNTCLOSE	DSTLNLN	MERGEPOLY	SCRVTR	SYMBPROD
>	CLNTEXEC	ERROR	MESHSIZE	SDERIVE	SYMBSUM
<=	CLNTREAD	EVOLUTE	MOFFSET	SDIVIDE	SYSTEM
>=	CLNTWRITE	EXIT	MOMENT	SEDITPT	TAN
ABS	CMESH	EXP	MSLEEP	SEVAL	TBEZIER
ACOS	CMORPH	EXTRUDE	NIL	SFOCAL	TBSPLINE
ADAPISO	CMULTIRES	FFCOMPAT	NTH	SFROMCRVS	TDERIVE
ARC	CNORMAL	FFEXTREME	OFFSET	SGAUSS	TEVAL
AREA	COERCE	FFMATCH	PAUSE	SIN	TEXTGEOM
ASIN	COLOR	FFMERGE	PCIRCLE	SINTERP	TFROMSRFS
ATAN	COMMENT	FFPTTYPE	PDOMAIN	SIZEOF	TIME
ATAN2	COMPOSE	FFSPLIT	PLN3PTS	SMEANSQR	TINTERP
ATTRIB	CON2	FLOOR	POLY	SMERGE	THISOBJ
AOFFSET	CONE	FMOD	POWER	SMORPH	TORUS
AWIDTH	CONTOUR	FOR	PRINTF	SNOC	TRANS
BOOLONE	CONVEX	FREE	PRISA	SNORMAL	TREFINE
BOOLSUM	COORD	FUNCTION	PROCEDURE	SNRMLSRF	TREGION
BOX	COS	GBOX	PT3BARY	SPHERE	TRIMSRF
BSP2BZR	CPOLY	GETLINE	PTLNPLN	SQRT	TSUBDIV
BZR2BSP	CRAISE	GPOLYGON	PTPTLN	SRAISE	VARLIST
CBEZIER	CREFINE	GPOLYLINE	PTSLNLN	SREFINE	VECTOR
CBSPLINE	CREGION	HELP	RANDOM	SREGION	VIEW
CCINTER	CREPARAM	HERMITE	RMATTR	SREPARAM	VIEWOBJ
CCRVTR	CROSSEC	HOMOMAT	ROTVEC	SRINTER	VOLUME
CDERIVE	CRVLNDST	IF	ROTX	STANGENT	WHILE
CDIVIDE	CRVPTDST	INCLUDE	ROTY	STRIMSRF	

7 Functions and Variables

This section lists all the functions supported by the *IRIT* system according to their classes - mostly, the object type they return.

Functions that return a **NumericType**:

ABS	COS	EXP	POWER	THISOBJ
ACOS	CLNTEXEC	FLOOR	RANDOM	VOLUME
AREA	CPOLY	FMOD	SIN	
ASIN	DSTPTLN	LN	SIZEOF	
ATAN	DSTPTPLN	LOG	SQRT	
ATAN2	DSTLNLN	MESHSIZE	TAN	

Functions that return a **GeometricType**:

ADAPISO	CNORMAL	FFMATCH	RULEDSRF	STRIVAR
ARC	COERCE	FFMERGE	SBEZIER	SURFPREV
AOFFSET	COMPOSE	FFPTTYPE	SBSPLINE	SURFREV
BOOLONE	CON2	FFSPLIT	SCRVTR	SWEEPSRF
BOOLSUM	CONE	GBOX	SDERIVE	SWPSCLSRF
BOX	CONTOUR	GETLINE	SDIVIDE	SYMBCPROD
BSP2BZR	CONVEX	GPOLYGON	SEDITPT	SYMBDIFF
BZR2BSP	COORD	GPOLYLINE	SEVAL	SYMBDPROD
CBEZIER	CRAISE	HERMITE	SFOCAL	SYMBPROD
CBSPLINE	CREFINE	LOFFSET	SFROMCRVS	SYMBSUM
CCINTER	CREGION	MERGEPOLY	SGAUSS	TBEZIER
CCRVTR	CREPARAM	MOFFSET	SINTERP	TBSPLINE
CDERIVE	CROSSEC	MOMENT	SMEANSQR	TDERIVE
CDIVIDE	CRVLNDST	NIL	SMERGE	TEVAL
CEDITPT	CRVPTDST	OFFSET	SMORPH	TEXTGEOM
CEVAL	CSURFACE	PCIRCLE	SNORMAL	TFROMSRFS
CEXTREMES	CTANGENT	PDOMAIN	SNRMLSRF	TINTERP
CINFLECT	CTRIMSRF	PLN3PTS	SPHERE	TORUS
CINTERP	CTLPT	POLY	SRAISE	TREFINE
CIRCLE	CYLIN	PRISA	SREFINE	TREGION
CIRCPOLY	CZEROS	PROCEDURE	SREGION	TRIMSRF
CLNTREAD	EVOLUTE	PT3BARY	SREPARAM	TSUBDIV
CMESH	EXTRUDE	PTLNPLN	SRINTER	
CMORPH	FFCOMPAT	PTPTLN	STANGENT	
CMULTIRES	FFEXTREME	PTSLNLN	STRIMSRF	

Functions that create linear transformation matrices:

HOMOMAT	ROTX	ROTZ	ROTZ2V2	TRANS
ROTVEC	ROTY	ROTZ2V	SCALE	

Miscellaneous functions:

ATTRIB	ERROR	INCLUDE	NTH	SYSTEM
AWIDTH	EXIT	INTERACT	PAUSE	TIME
CHDIR	FOR	IRITSTATE	PRINTF	VARLIST
CLNTCLOSE	FREE	LIST	PROCEDURE	VECTOR
CLNTWRITE	FUNCTION	LOAD	RMATTR	VIEW
COLOR	HELP	LOGFILE	SAVE	VIEWOBJ
COMMENT	IF	MSLEEP	SNOC	WHILE

Variables that are predefined in the system:

AXES	MACHINE	POLY_APPROX_TOL	VIEW_MAT
DRAWCTLPT	POLY_APPROX_OPT	PRSP_MAT	
FLAT4PLY	POLY_APPROX_UV	RESOLUTION	

Constants that are predefined in the system:

AMIGA	E3	MAGENTA	PARAM_CENTRIP	SURFACE_TYPE
APOLLO	E4	MATRIX_TYPE	PARAM_CHORD	SUN
BLACK	E5	MSDOS	PARAM_UNIFORM	TRIMSRF_TYPE
BLUE	FALSE	NUMERIC_TYPE	PI	TRIVAR_TYPE
COL	GREEN	OFF	PLANE_TYPE	TRUE
CTLPT_TYPE	HP	ON	POINT_TYPE	UNDEF_TYPE
CURVE_TYPE	IBMOS2	P1	POLY_TYPE	UNIX
CYAN	IBMNT	P2	RED	VECTOR_TYPE
DEPTH	KV_FLOAT	P3	ROW	WHITE
E1	KV_OPEN	P4	SGI	YELLOW
E2	LIST_TYPE	P5	STRING_TYPE	

8 Language description

The front end of the *IRIT* solid modeler is an infix parser that mimics some of the C language behavior. The infix operators that are supported are plus (+), minus (-), multiply (*), divide (/), and power (^), for numeric operators, with the same precedence as in C.

However, unlike the C language, these operators are overloaded, [1] or different action is taken, based upon the different operands. This means that one can write '1 + 2', in which the plus sign denotes a numeric addition, or one can write 'PolyObj1 + PolyObj2', in which case the plus sign denotes the Boolean operation of a union between two geometric objects. The exact way each operator is overloaded is defined below.

In this environment, reals, integers, and even Booleans, are all represented as real types. Data are automatically promoted as necessary. For example, the constants TRUE and FALSE are defined as 1.0 and 0.0 respectively.

Each expression is terminated by a semicolon. An expression can be as simple as 'a;' which prints the value of variable a, or as complex as:

```
for ( t = 1.1, 0.1, 1.9,
        cb1 = csurface( sb, COL, t ):
        color( cb1, green ):
        snoc( cb1, cb_all )
    );
```

While an expression is terminated with a semicolon, a colon is used to terminate mini-expressions within an expression.

Once a complete expression is read in (i.e., a semicolon is detected) and parsed correctly (i.e. no syntax errors are found), it is executed. Before each operator or a function is executed, parameter type matching tests are made to make sure the operator can be applied to these operand(s), or that the function gets the correct set of arguments.

The parser is totally case insensitive, so Obj, obj, and OBJ will refer to the same object, while MergePoly, MERGEPOLY, and mergePoly will refer to the same function.

Objects (Variables if you prefer) need not be declared. Simply use them when you need them. Object names may be any alpha-numeric (and underscore) string of at most 30 characters. By assigning to an old object, the old object will be automatically deleted and if necessary its type will be modified on the fly.

Example:

[1] In fact the C language does support overloaded operators to some extent: '1 + 2' and '1.0 + 2.0' implies invocation of two different actions.

```
V = sin( 45 * pi / 180.0 );
V = V * vector( 1, 2, 3 );
V = V * rotx( 90 );
V = V * V;
```

will assign to V a NumericType equal to the sine of 45 degrees, the VectorType (1, 2, 3) scaled by the sine of 45, rotate that vector around the X axis by 90 degrees, and finally a NumericType which is the dot (inner) product of V with itself.

The parser will read from stdin, unless a file is specified on the command line or an INCLUDE command is executed. In both cases, when the end of file is encountered, the parser will again wait for input from stdin. In order to execute a file and quit in the end of the file, put an EXIT command as the last command in the file.

9 Operator overloading

The basic operators +, −, ∗, /, and ^ are overloaded. This section describes what action is taken by each of these operators depending on its arguments.

9.1 Overloading +

The + operator is overloaded above the following domains:

```
NumericType + NumericType -> NumericType
VectorType  + VectorType  -> VectorType   (Vector addition)
MatrixType  + MatrixType  -> MatrixType   (Matrix addition)
PolygonType + PolygonType -> PolygonType  (Boolean UNION operation)
CurveType   + CurveType   -> CurveType    (Curve curve profiling)
CurveType   + CtlPtType   -> CurveType    (Curve control point profiling)
CtlPtType   + CtlPtType   -> CurveType    (Control points profiling)
ListType    + ListType    -> ListType     (Append lists operator)
StringType  + StringType  -> StringType   (String concat)
StringType  + RealType    -> StringType   (String concat, real as int string)
```

Note: Boolean UNION of two disjoint objects (no common volume) will result with the two objects combined. It is the USER responsibility to make sure that the non intersecting objects are also disjoint - this system only tests for no intersection. Boolean UNION of two polyline objects will merge the list of polylines.

9.2 Overloading −

The − operator is overloaded above the following domains:

As a binary operator:

```
NumericType - NumericType -> NumericType
VectorType  - VectorType  -> VectorType   (Vectoric difference)
MatrixType  - MatrixType  -> MatrixType   (Matrix difference)
PolygonType - PolygonType -> PolygonType  (Boolean SUBTRACT operation)
```

As a unary operator:

```
- NumericType -> NumericType
- VectorType  -> VectorType    (Scale vector by -1)
- MatrixType  -> MatrixType    (Scale matrix by -1)
- PolygonType -> PolygonType   (Boolean NEGATION operation)
- CurveType   -> CurveType     (Curve parameterization is reversed)
- SurfaceType -> SurfaceType   (Surface parameterization is reversed)
```

Note: Boolean SUBTRACT of two disjoint objects (no common volume) will result with an empty object. For both a curve and a surface parameterization, reverse operation (binary minus) causes the object normal to be flipped as a side effect.

9.3 Overloading *

The * operator is overloaded above the following domains:

```
NumericType * NumericType   -> NumericType
VectorType  * NumericType   -> VectorType    (Vector scaling)
VectorType  * CurveType     -> CurveType     (Inner product projection)
VectorType  * SurfaceType   -> SurfaceType   (Inner product projection)
VectorType  * VectorType    -> NumericType   (Inner product)
MatrixType  * NumericType   -> MatrixType    (Matrix Scaling)
MatrixType  * PointType     -> PointType     (Point transformation)
MatrixType  * CtlPtType     -> CtlPtType     (Ctl Point transformation)
MatrixType  * VectorType    -> VectorType    (Vector transformation)
MatrixType  * MatrixType    -> MatrixType    (Matrix multiplication)
MatrixType  * GeometricType -> GeometricType (Object transformation)
MatrixType  * ListType      -> ListType      (Object hierarchy transform.)
PolygonType * PolygonType   -> PolygonType   (Boolean INTERSECTION operation)
```

Note: Boolean INTERSECTION of two disjoint objects (no common volume) will result with an empty object. Object hierarchy transform transforms any transformable object (GeometricType) found in the list recursively. Boolean INTERSECTION of two planar (XY plane) polyline objects will compute the intersection points of the two lists of polylines.

9.4 Overloading /

The / operator is overloaded above the following domains:

```
NumericType / NumericType -> NumericType
PolygonType / PolygonType -> PolygonType   (Boolean CUT operation)
```

Note: Boolean CUT of two disjoint objects (no common volume) will result with an empty object.

9.5 Overloading ^

The ^ operator is overloaded above the following domains:

```
NumericType ^ NumericType -> NumericType
VectorType  ^ VectorType  -> VectorType  (Cross product)
MatrixType  ^ NumericType -> MatrixType  (Matrix to the (int) power)
PolygonType ^ PolygonType -> PolygonType (Boolean MERGE operation)
StringType  ^ StringType  -> StringType  (String concat)
StringType  ^ RealType    -> StringType  (String concat, real as real string)
```

Note: Boolean MERGE simply merges the two sets of polygons without any intersection tests. Matrix powers must be positive integers or -1, in which case the matrix inverse (if it exists) is computed.

9.6 Overloading Equal (Assignments)

Assignments are allowed as side effects, in any place in an expression. If "Expr" is an expression, then "var = Expr" is the exact same expression with the side effect of setting Var to that value. There is no guarantee on the order of evaluation, so using Vars that are set within the same expression is a bad practice. Use parentheses to force the order of evaluation, i.e., "(var = Expr)".

9.7 Comparison operators ==, ! =, <, >, <=, >=

The conditional comparison operators can be applied to the following domains (o for a comparison operator):

```
NumericType o NumericType -> NumericType
StringType  o StringType  -> NumericType
PointType   o PointType   -> NumericType
VectorType  o VectorType  -> NumericType
PlaneType   o PlaneType   -> NumericType
```

The returned NumericType is non-zero if the condition holds, or zero if not. For PointTypes, VectorTypes, and PlaneTypes, only == and ! = comparisons are valid. This is either the same or different. For NumericTypes and StringTypes (uses strcmp) all comparisons are valid.

9.8 Logical operators &&, ||||, !

Complex logical expressions can be defined using the logical *and* (&&), logical *or* (||||) and logical *not* (!). These operators can be applied to NumericTypes that are considered Boolean results. That is, true for a non-zero value, and false otherwise. The returned NumericType is true if both operands are true for the *and* operator, at least one is true for the *or* operator, and the operand is false for the *not* operator. In all other cases, a false is returned. To make sure Logical expressions are readable, the *and* and *or* operators are defined to have the *same* priority. Use parentheses to disambiguate a logical expression and to make it more readable.

9.9 Geometric Boolean Operations

The *IRIT* solid modeling system supports Boolean operations between polyhedra objects. Freeform objects will be automaticaly converted to a polygonal representation when used in Boolean operations. The $+$, $*$, and $-$ are overloaded to denote Boolean union, intersection and subtraction when operating on geometric entities. $-$ can also be used as an unary operator to reverse the object orientation inside out.

Example:

```
resolution = 20;
B = box(vector(-1, -1, -0.25), 2, 1.2, 0.5);
C = con2(vector(0, 0, -1.5), vector(0, 0, 3), 0.7, 0.3);

D = convex(B - C);
E = convex(C - B);
F = convex(B + C);
G = convex(B * C);

tr = rotx( -90 ) * roty( 40 ) * rotx( -30 );
```

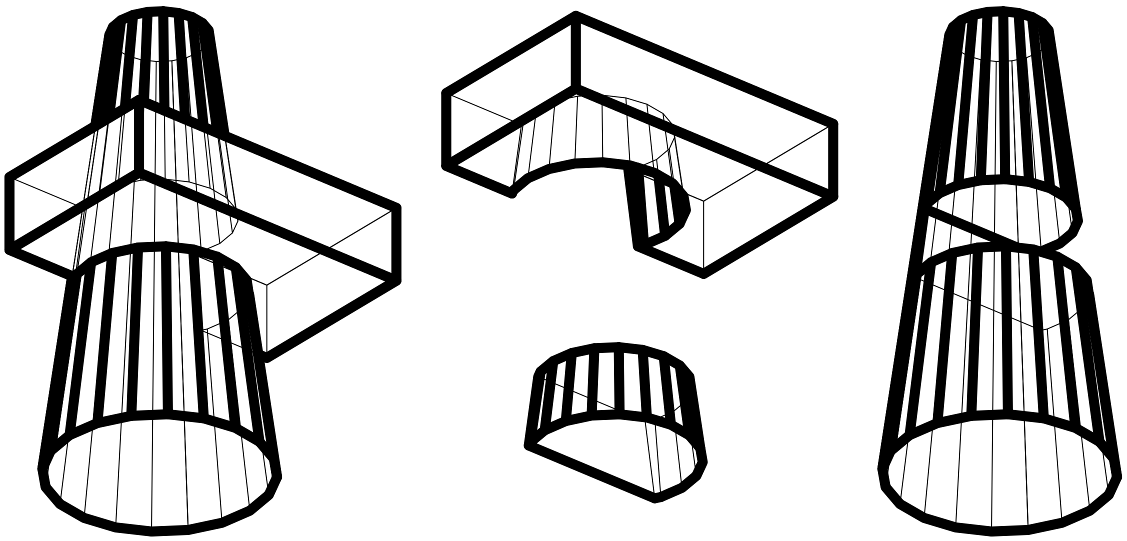


Figure 1: Geometric Boolean operatioms between a box and a truncated cone. Shown are union (left), intersection (bottom center), box minus the cone (top center), and cone minus the box (right).

```
All = list( D * tr * trans( vector(  0.6,  0.5, 0.0 ) ),
            E * tr * trans( vector(  3.0,  0.0, 0.0 ) ),
            F * tr * trans( vector( -2.0,  0.0, 0.0 ) ),
            G * tr * trans( vector(  0.7, -1.0, 0.0 ) ) )
        * scale( vector( 0.25, 0.25, 0.25 ) )
        * trans( vector( -0.1, -0.3, 0.0 ) );
view_mat = rotx( 0 );
view( list( view_mat, All ), on );
save( "booleans", list( view_mat, All ) );
```

A complete example to compute the union, intersection and both differences of a box and a truncated cone. The result of this example can be seen in Figure 1 with its hidden lines removed.

There are several flags to control the Boolean operations. See IRITSTATE command for the "InterCrv", "Coplanar", and "PolySort" states.

9.10 Priority of operators

The following table lists the priority of the different operators.

<table>
<tr><th>Lowest</th><th>Operator</th><th>Name of operator</th></tr>
<tr><td>priority</td><td>,</td><td>comma</td></tr>
<tr><td></td><td>:</td><td>colon</td></tr>
<tr><td></td><td>&&, ||||</td><td>logical and, logical or</td></tr>
<tr><td></td><td>=, ==, ! =, <=, >=, <, ></td><td>assignment, equal, not equal, less equal, greater equal, less, greater</td></tr>
<tr><td></td><td>+, -</td><td>plus, minus</td></tr>
<tr><td></td><td>*, /</td><td>multiply, divide</td></tr>
<tr><td>Highest</td><td>^</td><td>power</td></tr>
<tr><td>priority</td><td>-, !</td><td>unary minus, logical not</td></tr>
</table>

9.11 Grammar

The grammar of the *IRIT* parser follows similar guidelines as the C language for simple expressions. However, complex statements differ. See the IF, FOR, FUNCTION, and PROCEDURE below for the usage of these clauses.

10 Function Description

The description below defines the parameters and returned values of the predefined functions in the system, using the notation of functions in ANSI C. Listed are all the functions in the system, in alphabetic order, according to their classes.

10.1 NumericType returning functions

10.1.1 ABS

```
NumericType ABS( NumericType Operand )
```

Returns the absolute value of the given **Operand**.

10.1.2 ACOS

```
NumericType ACOS( NumericType Operand )
```

Returns the arc cosine value (in radians) of the given **Operand**.

10.1.3 AREA

```
NumericType AREA( PolygonType Object )
```

Returns the area of the given **Object** (in object units). Returned is the area of the polygonal object, not the area of the primitive it might approximate.

This means that the area of a polygonal approximation of a sphere will be returned, not the exact area of the sphere.

10.1.4 ASIN

```
NumericType ASIN( NumericType Operand )
```

Returns the arc sine value (in radians) of the given **Operand**.

10.1.5 ATAN

```
NumericType ATAN( NumericType Operand )
```

Returns the arc tangent value (in radians) of the given **Operand**.

10.1.6 ATAN2

```
NumericType ATAN2( NumericType Operand1, NumericType Operand2 )
```

Returns the arc tangent value (in radians) of the given ratio: **Operand1 / Operand2**, over the whole circle.

10.1.7 COS

```
NumericType COS( NumericType Operand )
```

Returns the cosine value of the given **Operand** (in radians).

10.1.8 CLNTEXEC

```
NumericType CLNTEXEC( StringType ClientName )
```

Initiate communication channels to a client named **ClientName**. **ClientName** is executed by this function as a sub process and two communication channels are opened between the IRIT server and the new client, for read and write. See also CLNTREAD, CLNTWRITE, and CLNTCLOSE. if **ClientName** is an empty string, the user is provided with the new communication port to be used and the server blocks for the user to manualy executed the client after setting the proper IRIT_SERVER_HOST/PORT environment variables.

Example:

```
h1 = CLNTEXEC( "" );
h2 = CLNTEXEC( "nuldrvs -s-" );
```

executes two clients, one is named **nuldrvs** and the other one is prompted for by the user. As a result of the second invokation of CLNTEXEC, the user will be prompted with a message similar to,

```
Irit: Startup your program - I am waiting...

setenv IRIT_SERVER_PORT 2182
```

and he/she will need to set the proper environment variable and execute their client manually.

10.1.9 CPOLY

```
NumericType CPOLY( PolygonType Object )
```

Returns the number of polygons in the given polygonal **Object**.

10.1.10 DSTPTLN

```
NumericType DSTPTLN( PointType Pt, PointType LineOrig, VectorType LineRay )
```

Returns the distance between a given point **Pt** and line **LineOrig**, **LineRay**. See also PTPTLN.

10.1.11 DSTPTPLN

```
NumericType DSTPTPLN( PointType Pt, PlaneType Plane )
```

Returns the distance between a given point **Pt** and plane **Plane**.

10.1.12 DSTLNLN

```
NumericType DSTLNLN( PointType Line1Orig, VectorType Line1Ray,
                     PointType Line2Orig, VectorType Line2Ray )
```

Returns the distance between two lines defined by point **LineiOrig** and ray **LineiRay**. See also PTSLNLN.

10.1.13 EXP

```
NumericType EXP( NumericType Operand )
```

Returns the natural exponent value of the given **Operand**.

10.1.14 FLOOR

```
NumericType FLOOR( NumericType Operand )
```

Returns the largest integer not greater than **Operand**.

10.1.15 FMOD

```
NumericType FMOD( NumericType Operand, NumericType Mod )
```

Returns the floating point remainder of the division of **Operand** by **Mod**.

10.1.16 LN

```
NumericType LN( NumericType Operand )
```

Returns the natural logarithm value of the given **Operand**.

10.1.17 LOG

```
NumericType LOG( NumericType Operand )
```

Returns the base 10 logarithm value of the given **Operand**.

10.1.18 MESHSIZE

```
NumericType MESHSIZE( SurfaceType Srf, ConstantType Direction )
```

Returns the size of **Srf**'s mesh in **Direction**, which is one of COL or ROW.
Example:

```
RSize = MESHSIZE( Sphere, ROW );
CSize = MESHSIZE( Sphere, COL );
```

10.1.19 POWER

```
NumericType POWER( NumericType Operand, NumericType Exp )
```

Returns **Operand** to the power of **Exp**.

10.1.20 RANDOM

```
NumericType RANDOM( NumericType Min, NumericType Max )
```

Returns a randomized value between **Min** and **Max**.

10.1.21 SIN

```
NumericType SIN( NumericType Operand )
```

Returns the sine value of the given **Operand** (in radians).

10.1.22 SIZEOF

```
NumericType SIZEOF( ListType List | PolyType Poly |
                    CurveType Crv | StringType Str )
```

Returns the length of a list if **List**, the number of polygons if **Poly**, the length of the control polygon if **Crv**, or the number of characters in string if **Str**. If, however, only one polygon is in **Poly**, it returns the number of vertices in that polygon.

Example:

```
len = SIZEOF( list( 1, 2, 3 ) );
numPolys = SIZEOF( axes );
numCtlpt = SIZEOF( circle( vector( 0, 0, 0 ), 1 ) );
```

will assign the value of 3 to the variable **len**, set **numPolys** to the number of polylines in the axes object, and set numCtlPt to 9, the number of control points in a circle.

10.1.23 SQRT

```
NumericType SQRT( NumericType Operand )
```

Returns the square root value of the given **Operand**.

10.1.24 TAN

```
NumericType TAN( NumericType Operand )
```

Returns the tangent value of the given **Operand** (in radians).

10.1.25 THISOBJ

```
NumericType THISOBJ( StringType Object )
```

Returns the object type of the given name of an **Object**. This can be one of the constants,

UNDEF_TYPE	VECTOR_TYPE	MATRIX_TYPE	SURFACE_TYPE
NUMERIC_TYPE	POINT_TYPE	POLY_TYPE	TRIMSRF_TYPE
STRING_TYPE	CTLPT_TYPE	CURVE_TYPE	TRIVAR_TYPE

This is also a way to ask if an object by a given name do exist (if the returned type is UNDEF_TYPE or not).

10.1.26 VOLUME

```
NumericType VOLUME( PolygonType Object )
```

Returns the volume of the given **Object** (in object units). It returns the volume of the polygonal object, not the volume of the object it might approximate.

This routine decomposes all non-convex polygons to convex ones as a side effect (see CONVEX).

10.2 GeometricType returning functions

10.2.1 ADAPISO

```
CurveType ADAPISO( SurfaceType Srf, NumericType Dir, NumericType Eps,
                   NumericType FullIso, NumericType SinglePath )
```

Constructs a *coverage* to **Srf** using isocurve in the **Dir** direction, so that for any point p on surface **Srf**, there exists a point on one of the isocurves that is close to p within **Eps**. If **FullIso**, the extracted isocurves span the entire surface domain, otherwise they may span only a subset of the domain. If **SinglePath**, an approximation to a single path (Hamiltonian path) that visits all isocurves is constructed.

```
srf = sbezier( list( list( ctlpt( E3, -0.5, -1.0,  0.0 ),
                           ctlpt( E3,  0.4,  0.0,  0.1 ),
                           ctlpt( E3, -0.5,  1.0,  0.0 ) ),
                     list( ctlpt( E3,  0.0, -0.7,  0.1 ),
                           ctlpt( E3,  0.0,  0.0,  0.0 ),
                           ctlpt( E3,  0.0,  0.7, -0.2 ) ),
                     list( ctlpt( E3,  0.5, -1.0,  0.1 ),
                           ctlpt( E3, -0.4,  0.0,  0.0 ),
                           ctlpt( E3,  0.5,  1.0, -0.2 ) ) ) );
aiso = ADAPISO( srf, COL, 0.1, FALSE, FALSE );
```

Constructs an adaptive isocurve approximation with tolerance of **0.1** to surface **srf** in direction **COL**. Isocurves are allowed to span a subset of the surface domain. No single path is needed.

The **SinglePath** option is currently not supported.

10.2.2 ARC

```
CurveType ARC( VectorType StartPos, VectorType Center, VectorType EndPos )
```

Constructs an arc between the two end points **StartPos** and **EndPos**, centered at **Center**. Arc will always be less than 180 degrees, so the shortest circular path from **StartPos** to **EndPos** is selected. The case where **StartPos**, **Center**, and **EndPos** are collinear is illegal, since it attempts to define a 180 degrees arc. Arc is constructed as a single rational quadratic Bezier curve.

Example:

```
Arc1 = ARC( vector( 1.0, 0.0, 0.0 ),
            vector( 1.0, 1.0, 0.0 ),
            vector( 0.0, 1.0, 0.0 ) );
```

constructs a 90 degrees arc, tangent to both the X and Y axes at coordinate 1. See Figure 2.

10.2.3 AOFFSET

```
CurveType AOFFSET( CurveType Crv, NumericType OffsetDistance,
                   NumericType Epsilon, NumericType TrimLoops,
                   NumericType BezInterp )
```

Computes an offset of **OffsetDistance** with globally bounded error (controlled by **Epsilon**). The smaller **Epsilon** is, the better the approximation to the offset. The bounded error is achieved by adaptive refinement of the **Crv**. If **TrimLoops** is TRUE or on, the regions of the object that self-intersect as a result of the offset operation are trimmed away. If **BezInterp** is TRUE, each curve's segment is interpolated instead of approximated.

Example:

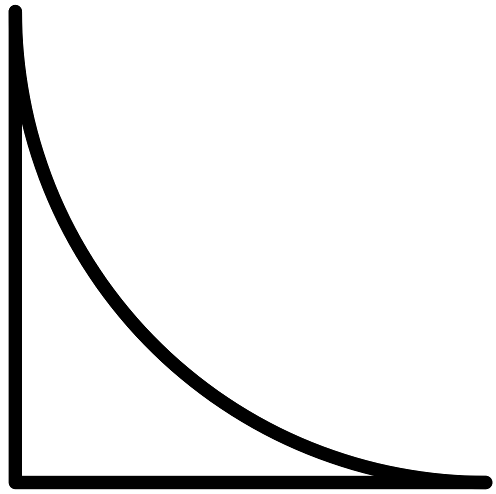


Figure 2: A 90 degree arc constructed using the ARC constructor.

Figure 3: Adaptive offset approximation (thick) of a Bspline curve (thin). On the left, the self intersections in the offset computed in the right are eliminated. Both offsets were computedusing AOFFSET. (See also Figure 31.)

```
OffCrv1 = AOFFSET( Crv, 0.5, 0.01, FALSE, FALSE );
OffCrv2 = AOFFSET( Crv, 0.5, 0.01, TRUE, FALSE );
```

computes an adaptive offset to **Crv** with **OffsetDistance** of 0.5 and **Epsilon** of 0.01 and trims the self-intersection loops in the second instrance. See also OFFSET, LOFFSET, and MOFFSET. See Figure 3.

10.2.4 BOOLONE

```
SurfaceType BOOLONE( CurveType Crv )
```

Given a closed curve, the curve is subdivided into four segments equally spaced in the parametric space that are fed into BOOLSUM. Useful if a surface should "fill" the area enclosed by a closed curve.

Example:

```
Srf = BOOLONE( circle( vector( 0.0, 0.0, 0.0 ), 1.0 ) );
```

Creates a disk surface containing the area enclosed by the unit circle. See Figure 4.

10.2.5 BOOLSUM

```
SurfaceType BOOLSUM( CurveType Crv1, CurveType Crv2,
                     CurveType Crv3, CurveType Crv4 )
```

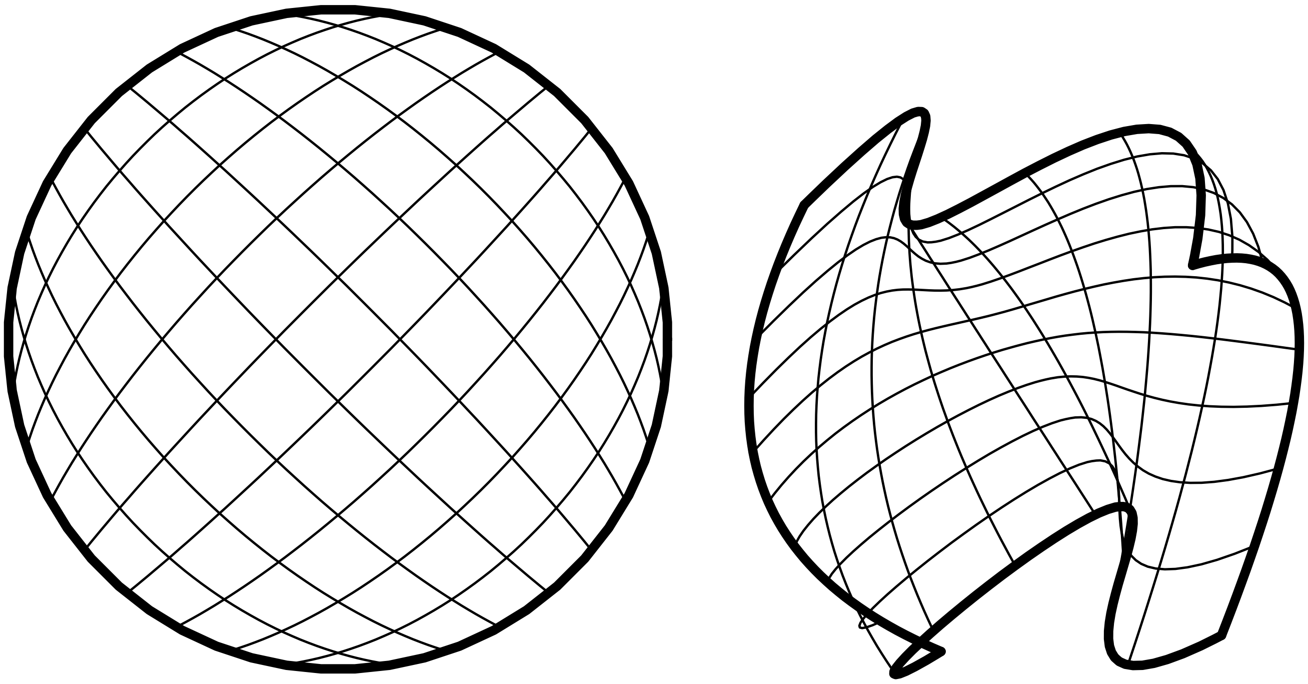


Figure 4: A boolean sum of a circle creates a disk (left) using BOOLONE and a general boolean sum of four curves (right).

Construct a surface using the provided four curves as its four boundary curves. Curves do not have to have the same order or type, and will be promoted to their least common denominator. The end points of the four curves should match as follows:

Crv1 start point,	to **Crv3** start point.
Crv1 end point,	to **Crv4** start point.
Crv2 start point,	to **Crv3** end point.
Crv2 end point,	to **Crv4** end point.

where **Crv1** and **Crv2** are the two boundaries in one parametric direction, and **Crv3** and **Crv4** are the two boundaries in the other parametric direction.

Example:

```
Cbzr1 = cbezier( list( ctlpt( E3, 0.1, 0.1, 0.1 ),
                       ctlpt( E3, 0.0, 0.5, 1.0 ),
                       ctlpt( E3, 0.4, 1.0, 0.4 ) ) );
Cbzr2 = cbezier( list( ctlpt( E3, 1.0, 0.2, 0.2 ),
                       ctlpt( E3, 1.0, 0.5, -1.0 ),
                       ctlpt( E3, 1.0, 1.0, 0.3 ) ) );
Cbsp3 = cbspline( 4,
                  list( ctlpt( E3, 0.1,  0.1, 0.1 ),
                        ctlpt( E3, 0.25, 0.0, -1.0 ),
                        ctlpt( E3, 0.5,  0.0, 2.0 ),
                        ctlpt( E3, 0.75, 0.0, -1.0 ),
                        ctlpt( E3, 1.0,  0.2, 0.2 ) ),
                  list( KV_OPEN ) );
Cbsp4 = cbspline( 4,
                  list( ctlpt( E3, 0.4,  1.0, 0.4 ),
```

```
                           ctlpt( E3, 0.25, 1.0, 1.0 ),
                           ctlpt( E3, 0.5,  1.0, -2.0 ),
                           ctlpt( E3, 0.75, 1.0, 1.0 ),
                           ctlpt( E3, 1.0,  1.0, 0.3 ) ),
                     list( KV_OPEN ) );
   Srf = BOOLSUM( Cbzr1, Cbzr2, Cbsp3, Cbsp4 );
```

10.2.6 BOX

```
PolygonType BOX( VectorType Point,
                 NumericType Dx, NumericType Dy, NumericType Dz )
```

Creates a BOX polygonal object, whose boundary is coplanar with the XY, XZ, and YZ planes. The BOX is defined by **Point** as base position, and **Dx, Dy, Dz** as BOX dimensions. Negative dimensions are allowed.

Example:

```
   B = BOX( vector( 0, 0, 0 ), 1, 1, 1);
```

creates a unit cube from 0 to 1 in all axes.

10.2.7 BZR2BSP

```
CurveType BZR2BSP( CurveType Crv )
```

or

```
SurfaceType BZR2BSP( SurfaceType Srf )
```

Creates a Bspline curve or a Bspline surface from the given Bezier curve or Bezier surface. The Bspline curve or surface is assigned open end knot vector(s) with no interior knots, in the parametric domain of zero to one.

Example:

```
   BspSrf = BZR2BSP( BzrSrf );
```

10.2.8 BSP2BZR

```
CurveType | ListType BSP2BZR( CurveType Crv )
```

or

```
SurfaceType | ListType BSP2BZR( SurfaceType Srf )
```

Creates Bezier curve(s) or surface(s) from a given Bspline curve or a Bspline surface. The Bspline input is subdivided at all internal knots to create Bezier curves or surfaces. Therefore, if the input Bspline does have internal knots, a list of Bezier curves or surfaces is returned. Otherwise, a single Bezier curve or surface is returned.

Example:

```
   BzrCirc = BSP2BZR( circle( vector( 0.0, 0.0, 0.0 ), 1.0 ) );
```

would subdivide the unit circle into four 90 degrees Bezier arcs returned in a list.

10.2.9 CBEZIER

```
CurveType CBEZIER( ListType CtlPtList )
```

Creates a Bezier curve out of the provided control point list. **CtlPtList** is a list of control points, all of which must be of type (E1-E5, P1-P5), or regular PointType defining the curve's control polygon. Curve's point type will be of a space which is the union of the spaces of all points. The created curve is the polynomial (or rational),

$$C(t) = \sum_{i=0}^{k} P_i B_i(t), \tag{1}$$

where P_i are the control points **CtlPtList**, and k is the degree of the curve, which is one less than the number of points.

Example:

```
s45 = sin(pi / 4);
Arc90 = CBEZIER( list( ctlpt( P2, 1.0, 0.0, 1.0 ),
                       ctlpt( P2, s45, s45, s45 ),
                       ctlpt( P1, 1.0, 1.0 ) ) );
```

constructs an arc of 90 degrees as a rational quadratic Bezier curve.

10.2.10 CBSPLINE

```
CurveType CBSPLINE( NumericType Order, ListType CtlPtList,
                                                   ListType KnotVector )
```

Creates a Bspline curve out of the provided control point list, the knot vector, and the specified order. **CtlPtList** is a list of control points, all of which must be of type (E1-E5, P1-P5), or regular PointType defining the curve's control polygon. Curve's point type will be of a space which is the union of the spaces of all points. The length of the **KnotVector** must be equal to the number of control points in **CtlPtList** plus the **Order**. If, however, the length of the knot vector is equal to **#CtlPtList + Order + Order - 1** the curve is assumed *periodic.* The knot vector list may be specified as either **list(KV_OPEN)** or **list(KV_FLOAT)** or **list(KV_PERIODIC)** in which a uniform open, uniform floating or uniform periodic knot vector with the appropriate length is automatically constructed.

The created curve is the piecewise polynomial (or rational),

$$C(t) = \sum_{i=0}^{k} P_i B_{i,\tau}(t), \tag{2}$$

where P_i are the control points **CtlPtList** and k is the degree of the curve, which is one less than the **Order** or number of points. τ is the knot vector of the curve.

Example:

```
s45 = sin(pi / 4);
HalfCirc = CBSPLINE( 3,
                     list( ctlpt( P3,  1.0,  1.0,  0.0, 0.0 ),
                           ctlpt( P3,  s45,  s45,  s45, 0.0 ),
                           ctlpt( P3,  1.0,  0.0,  1.0, 0.0 ),
                           ctlpt( P3,  s45, -s45,  s45, 0.0 ),
                           ctlpt( P3,  1.0, -1.0,  0.0, 0.0 ) ),
                     list( 0, 0, 0, 1, 1, 2, 2, 2 ) );
```

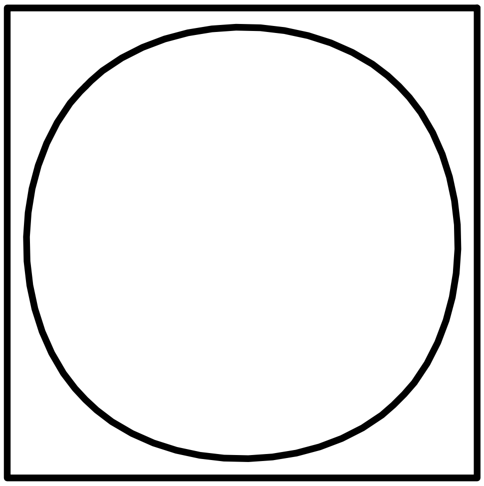


Figure 5: A cubic periodic curve created using KV_PERIODIC end conditions.

constructs an arc of 180 degrees in the XZ plane as a rational quadratic Bspline curve.
Example:

```
c = CBSPLINE( 4,
              list( ctlpt( E2,  0.5,  0.5 ),
                    ctlpt( E2, -0.5,  0.5 ),
                    ctlpt( E2, -0.5, -0.5 ),
                    ctlpt( E2,  0.5, -0.5 ) ),
              list( KV_PERIODIC ) );
color( c, red );
viewobj( c );

c1 = cregion( c, 3, 4 );
color( c1, green );
c2 = cregion( c, 4, 5 );
color( c2, yellow );
c3 = cregion( c, 5, 6 );
color( c3, cyan );
c4 = cregion( c, 6, 7 );
color( c3, magenta );
viewobj( list( c1, c2, c3, c4 ) );
```

creates a periodic curve and extracts its four polynomial domains as four *open* end Bspline curves. See Figure 5.

10.2.11 CCINTER

```
ListType CCINTER( CurveType Crv1, CurveType Crv2, NumericType Epsilon,
                                                  NumericType SelfInter )
```

or

```
SurfaceType CCINTER( CurveType Crv1, CurveType Crv2, NumericType Epsilon,
                                                  NumericType SelfInter )
```

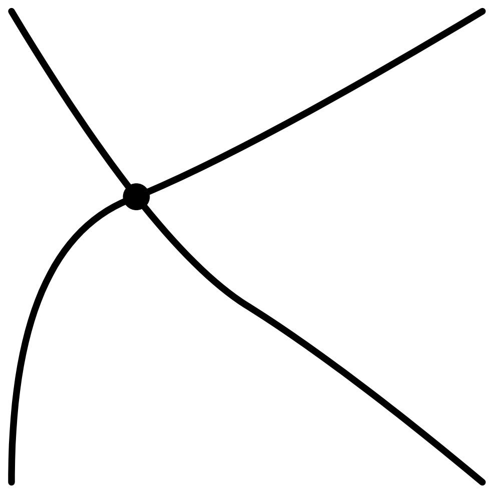


Figure 6: A intersection point of two freeform curve computed using CCINTER.

Computes the intersection point(s) of **Crv1** and **Crv2** in the XY plane. Since this computation involves numeric operations, **Epsilon** controls the accuracy of the parametric values of the result. It returns a list of PointTypes, each containing the parameter of **Crv1** in the X coordinate, and the parameter of **Crv2** in the Y coordinate. If, however, **Epsilon** is negative, a scalar field surface representing the square of the distance function is returned instead. If **SelfInter** is TRUE, **Crv1** and **Crv2** can be the same curve, and self-intersection points are searched instead.

Example:

```
crv1 = cbspline( 3,
                 list( ctlpt( E2, 0, 0 ),
                       ctlpt( E2, 0, 0.5 ),
                       ctlpt( E2, 0.5, 0.7 ),
                       ctlpt( E2, 1, 1 ) ),
                 list( KV_OPEN ) );
crv2 = cbspline( 3,
                 list( ctlpt( E2, 1, 0 ),
                       ctlpt( E2, 0.7, 0.25 ),
                       ctlpt( E2, 0.3, 0.5 ),
                       ctlpt( E2, 0, 1 ) ),
                 list( KV_OPEN ) );
inter_pts = CCINTER( crv1, crv2, 0.0001, FALSE );
```

Computes the parameter values of the intersection point of **crv1** and **crv2** to a tolerance of 0.0001. See Figure 6.

10.2.12 CCRVTR

```
NumericType CCRVTR( CurveType Crv, NumericType Epsilon )
```

or

```
CurveType CCRVTR( CurveType Crv, NumericType Epsilon )
```

Computes the extreme curvature points on **Crv** in the XY plane. This set includes not only points of maximum (convexity) and mimumum (concavity) curvature, but also points of zero curvature such as

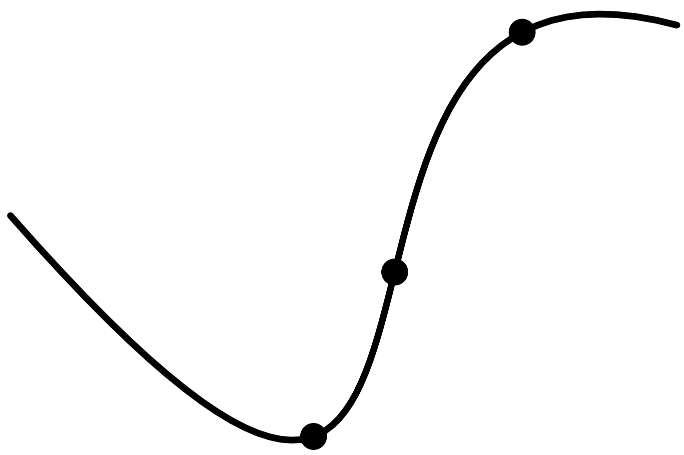


Figure 7: Extreme curvature locations on a freeform curve computed using CCRVTR.

inflection points. Since this operation is partially numeric, **Epsilon** is used to set the needed accuracy. It returns the parameter value(s) of the location(s) with extreme curvature along the **Crv**. If, however, **Epsilon** is negative, the curvature scalar field curve is returned as a two dimensional rational vector field curve, for which the first dimension is equal to the parameter, and the second is the curvature value at that parameter.

This function computes the curvature scalar field for planar curves as,

$$\kappa(t) = \frac{x'(t)y''(t) - x''(t)y'(t)}{((x'(t))^2 + (y'(t))^2\ )^{\frac{3}{2}}}, \tag{3}$$

and computes kN for three dimensional curves as the following vector field,

$$\kappa(t)N(t) = \kappa(t)B(t) \times T(t) = \frac{C' \times C''}{\|C'\|^3} \times \frac{C'}{\|C'\|} = \frac{(C' \times C'') \times C'}{\|C'\|^4}. \tag{4}$$

The extremum values are extracted from the computed curvature field. This curvature field is a high order curve, even if the input geometry is of low order. This is especially true for rational curves, for which the quotient rule for differentiation is used and almost doubles the degree in every differentiation.

See also CZEROS, CEXTREMES, and SCRVTR.

Example:

```
crv = cbezier( list( ctlpt( E2, -1.0,  0.5 ),
                     ctlpt( E2, -0.5, -2.0 ),
                     ctlpt( E2,  0.0,  1.0 ),
                     ctlpt( E2,  1.0,  0.0 ) ) ) * rotz( 30 );
crvtr = CCRVTR( crv, 0.001 );
pt_crvtr = nil();
pt = nil();
for ( i = 1, 1, sizeof( crvtr ),
    ( pt = ceval( crv, nth( crvtr, i ) ) ):
    snoc( pt, pt_crvtr )
);
interact( list( crv, pt_crvtr ) );
```

finds the extreme curvature points in **Crv** and displays them all with the curve. See Figure 7.

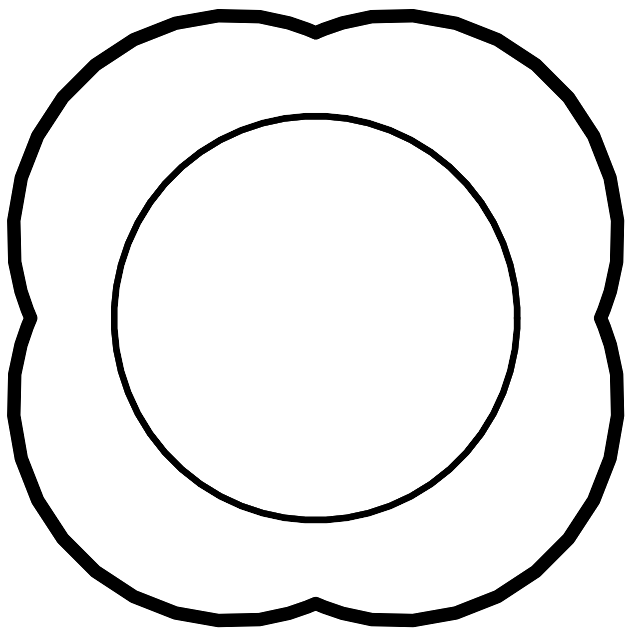


Figure 8: The hodograph (thick) of a bspline circle (thin) constructed as four 90 degrees rational bezier arcs, computed using CDERIVE.

10.2.13 CDERIVE

```
CurveType CDERIVE( CurveType Curve )
```

Returns a vector field curve representing the differentiated curve, also known as the Hodograph curve.

Example:

```
Circ = circle( vector( 0.0, 0.0, 0.0 ), 1.0 );
Hodograph = CDERIVE( Circ );
```

See Figure 8.

10.2.14 CDIVIDE

```
ListType CDIVIDE( CurveType Curve, NumericType Param )
```

Subdivides a curve into two sub-curves at the specified parameter value. **Curve** can be either a Bspline curve in which **Param** must be within the Curve's parametric domain, or a Bezier curve in which **Param** must be in the range of zero to one.

It returns a list of the two sub-curves. The individual curves may be extracted from the list using the NTH command.

Example:

```
CrvLst = CDIVIDE( Crv, 1.3 );
Crv1 = nth( CrvLst, 1 );
Crv2 = nth( CrvLst, 2 );
```

subdivides the curve **Crv** at the parameter value of 0.5. See Figure 9.

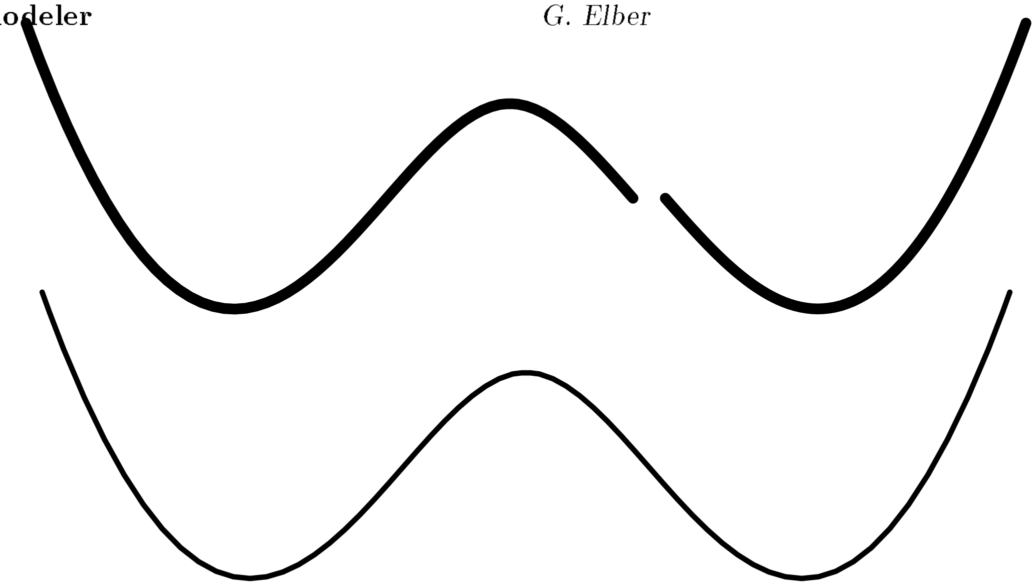


Figure 9: A Bspline curve is subdivided into two distinct regions using CDIVIDE.

10.2.15 CEDITPT

`CurveType CEDITPT( CurveType Curve, CtlPtType CtlPt, NumericType Index )`

Provides a simple mechanism to manually modify a single control point number **Index** (base count is 0) in **Curve**, by substituting **CtlPt** instead. **CtlPt** must have the same point type as the control points of the **Curve**. Original curve **Curve** is not modified.

Example:

```
CPt = ctlpt( E3, 1, 2, 3 );
NewCrv = CEDITPT( Curve, CPt, 1 );
```

constructs a **NewCrv** with the second control point of **Curve** being **CPt**.

10.2.16 CEVAL

`CtlPtType CEVAL( CurveType Curve, NumericType Param )`

Evaluates the provided **Curve** at the given **Param** value. **Param** should be in the curve's parametric domain if **Curve** is a Bspline curve, or between zero and one if **Curve** is a Bezier curve. The returned control point has the same point type as the control points of the **Curve**.

Example:

```
CPt = CEVAL( Crv, 0.25 );
```

evaluates **Crv** at the parameter value of 0.25.

10.2.17 CEXTREMES

`ListType CEXTREMES( CurveType Crv, NumericType Epsilon, NumericType Axis )`

Computes the extreme set of the given **Crv** in the given axis (1 for X, 2 for Y, 3 for Z). Since this computation is numeric, an **Epsilon** is also required to specify the desired tolerance. It returns a list of all the parameter values (NumericType) in which the curve takes an extreme value.

Example:

```
extremes = CEXTREMES( Crv, 0.0001, 1 );
```

Computes the extreme set of curve **crv**, in the **X** axis, with error tolerance of **0.0001**. See also CZERO. See Figure 10.

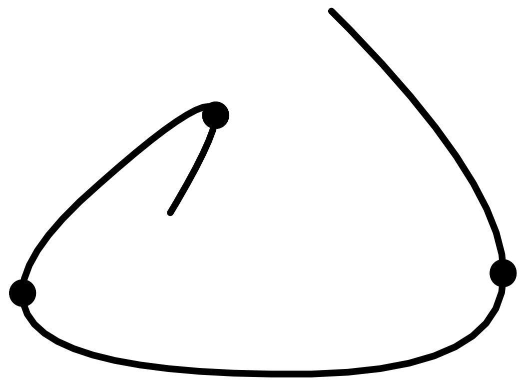


Figure 10: The X local extremums of a freeform curve are isolated using CEXTREMES.

10.2.18 CINFLECT

```
ListType CINFLECT( CurveType Crv, NumericType Epsilon )
```

or

```
CurveType CINFLECT( CurveType Crv, NumericType Epsilon )
```

Computes the inflection points of **Crv** in the XY plane. Since this computation is numeric, an **Epsilon** is also required to specify the desired tolerance. It returns a list of all the parameter values (NumericType) in which the curve has an inflection point. If, however, **Epsilon** is negative, a scalar field curve representing the sign of the curvature of the curve is returned instead.

The sign of curvature scalar field is equal to,

$$\sigma(t) = x'(t)y''(t) - x''(t)y'(t). \tag{5}$$

Example:

```
inflect = CINFLECT( crv, 0.001 );
pt_inflect = nil();
pt = nil();
for ( i = 1, 1, sizeof( inflect ),
        pt = ceval( crv, nth( inflect, i ) ):
        snoc( pt, pt_inflect )
    );
interact( list( axes, crv, pt_inflect ), 0);
```

Computes the set of inflection points of curve **crv** with error tolerance of **0.001**. This set is then scanned in a loop and evaluated to the curve's locations which are then displayed with the **crv**. See also CZEROS, CEXTREMES, and CCRVTR. See Figure 11.

10.2.19 CINTERP

```
CurveType CINTERP( ListType PtList, NumericType Order, NumericType Size,
                                    ConstantType Param, NumericType Periodic)
```

Computes a Bspline polynomial curve that interpolates or approximates the list of points in **PtList**. The Bspline curve will have order **Order** and **Size** control points, and will be periodic if **periodic** is none zero. The knots will be spaced according to **Param** which can be one of PARAM_UNIFORM,

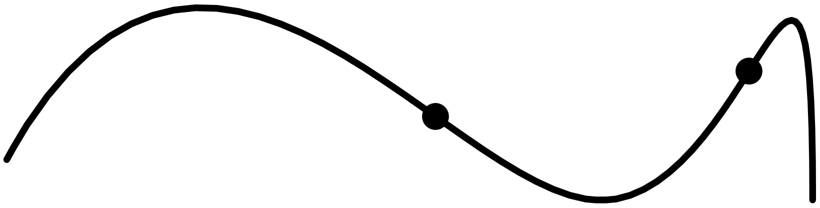


Figure 11: The Inflection points of a freeform curve can be isolated using CINFLECT.

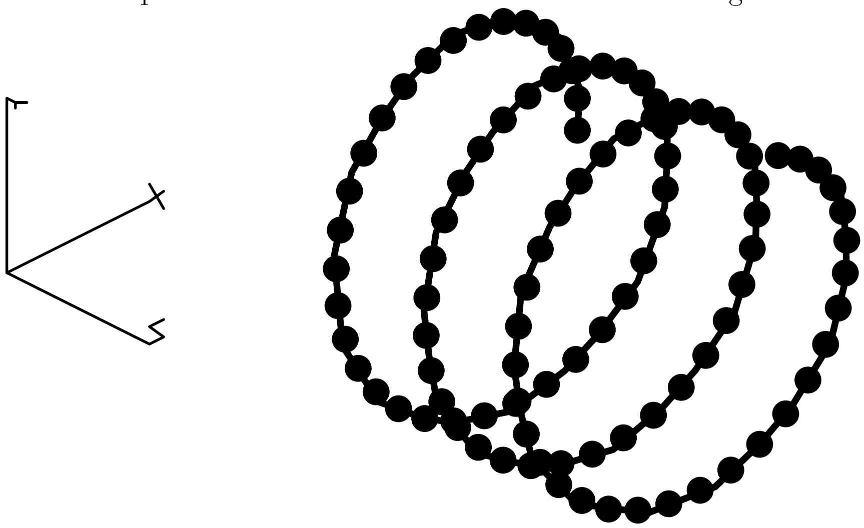


Figure 12: A Helix is sampled at 100 locations is least square fit using CINTERP by a quadratic Bspline curve and 21 control points.

PARAM_CHORD or PARAM_CENTRIP. The former prescribes a uniform knot sequence and the latters specify knot spacing according to the chord length and a square root of the chord length. A periodic curve will be coerced to have PARAM_UNIFORM knot sequence. Use of **Periodic** end conditions can create cases with degenerated linear systems (determinant equal zero). Increase or decrease of the **Order** of the Bspline by one will resolve the problem. All points in **PtList** must be of type (E1-E5, P1-P5) control point, or regular PointType. If **Size** is equal to the number of points in **PtList** the resulting curve will *interpolate* the data set. Otherwise, if **Size** is less than the number of points in **PtList** the point data set will be least square approximated. In no time can **Size** be lower than **Order**. **Size** of zero forces interpolation by setting **Size** to the data set size. All interior knots will be distinct preserving maximal continuity. The resulting Bspline curve will have open end conditions.

Example:

```
pl = nil();
for ( x = 0, 1, 100,
      snoc(point(cos(x / 5), sin(x / 5), x / 50 - 1), pl)
);
c = CINTERP( pl, 3, 21, PARAM_UNIFORM );
```

Samples a helical curve at 100 points and least square fit a quadratic Bspline curve with 21 point to the data set. The curve will have a uniform knot spacing. See Figure 12.

10.2.20 CIRCLE

`CurveType CIRCLE( VectorType Center, NumericType Radius )`

Constructs a circle at the specified **Center** with the specified **Radius**. The returned circle is a Bspline curve of four piecewise Bezier 90 degree arcs. The construced circle is always parallel to the XY plane. Use the linear transformation routines to place the circle in the appropriate orientation and location.

10.2.21 CIRCPOLY

```
PolygonType CIRCPOLY( VectorType Normal, VectorType Trans, NumericType Radius )
```

Defines a circular polygon in a plane perpendicular to **Normal** that contains the **Trans** point. Constructed polygon is centered at **Trans**. RESOLUTION vertices will be defined with **Radius** from distance from **Trans**.

Alternative ways to construct a polygon are manual construction of the vertices using POLY, or the construction of a flat ruled surface using RULEDSRF.

10.2.22 CLNTREAD

```
AnyType CLNTREAD( NumericType Handler, NumericType Block )
```

Reads one object from a communication channel of a client. **Handler** contains the index of the communication channel opened via CLNTEXEC. If no data is available in the communication channel, this function will block for at most **Block** millisecond until data is found or timeout occurs. In the latter, a single StringType object is returned with the content of "no data (timeout)". See also CLNTWRITE, CLNTCLOSE, and CLNTEXEC.

Example:

```
h2 = clntexec( "xmtdrvs -s-" );
    .
    .

Model = CLNTREAD( h2 );
    .
    .

clntclose( h2,TRUE );
```

reads one object from client through communication channel h2 and save it in variable Model.

10.2.23 CMESH

```
CurveType CMESH( SurfaceType Srf, ConstantType Direction, NumericType Index )
```

Returns a single ROW or COLumn as specified by the **Direction** and **Index** (base count is 0) of the control mesh of surface **Srf**.

The returned curve will have the same knot vector as **Srf** in the appropriate direction. See also CSURFACE.

This curve is *not* necessarily in the surface **Srf**. It is equal to,

$$C(t) = \sum_{i=0}^{m} P_{ij} B_i(t), \tag{6}$$

and similar for the other parametric direction.

Example:

```
Crv = CMESH( Srf, COL, 0 );
```

extracts the first column of surface **Srf** as a curve. See also CSURFACE.

10.2.24 CMORPH

```
CurveType CMORPH( CurveType Crv1, CurveType Crv2,
                  NumericType Method, NumericType Blend )
```

or

```
ListType CMORPH( CurveType Crv1, CurveType Crv2,
                 NumericType Method, NumericType Blend )
```

Creates a new curve which is a *metamorph* of the two given curves. The two given curves must be compatible (see FFCOMPAT) before this blend is invoked. Very useful if a sequence that "morphs" one curve to another is to be created. Several methods of metamorphosis are supported according to the value of **Method**,

0	Simple convex blend.
1	Corner/Edge cutting scheme, scaled to same curve length.
2	Corner/Edge cutting scheme, scaled to same bounding box.
3	Same as 1 but with filtering out of tangencies.
4	Same as 2 but with filtering out of tangencies.
5	Multiresolution decompsition based metamorphosis. See CMULTRES.

In **Method** 1, **Blend** is a number between zero (**Crv1**) and one (**Crv2**) defining the similarity to **Crv1** and **Crv2**, respectively. A single curve is returned.

In **Method**s 2 to 5, **Blend** is a step size for the metamorphosis operation and a whole list describing the entire metamorphosis operation is returned.

Examples:

```
for ( i = 0, 1, 300,
    c = CMORPH( crv1a, crv1b, 0, i / 300.0 ):
    color( c, yellow ):
    viewobj( c )
);

crvs = CMORPH( crv1a, crv1b, 2, 0.003 );
snoc( crv1b, crvs );
for ( i = 1, 1, sizeof( crvs ),
    c = nth( crvs, i ):
    color( c, yellow ):
    viewobj( c )
);

Turtle2 = ffmatch( Wolf, Turtle, 20, 100, 2, false, 2 );
ffcompat( Wolf, Turtle2 );
for ( i = 0, 1, 25,
    c = CMORPH( Wolf, Turtle2, 0, i / 25 ):
    color( c, yellow ):
    viewobj( c )
);
```

creates three metamorphosis animation sequences, one that is based on a convex blend, one that is based on corner/edge cutting scheme. See alost SMORPH, and a third that employs FFMATCH. See Figure 13.

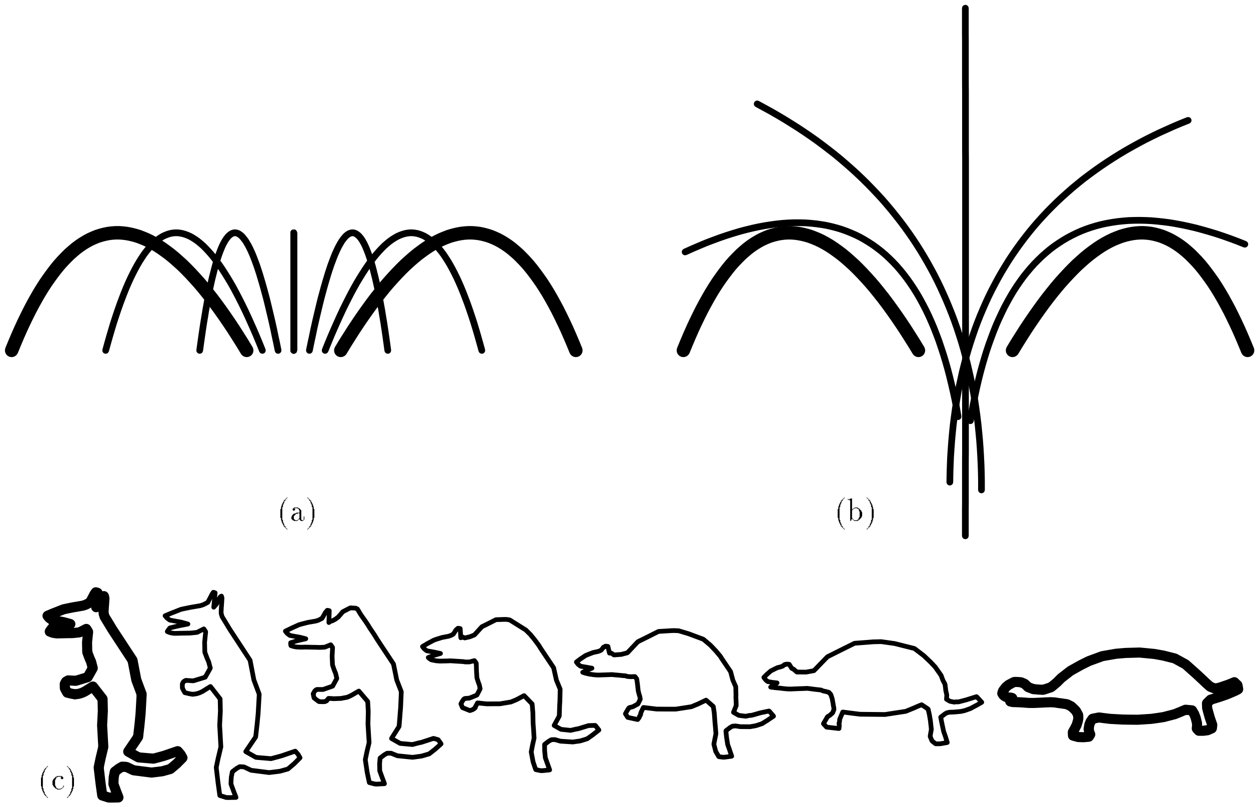


Figure 13: A morphing sequence using convex blend (a), edge cutting (b), and using FFMATCH and convex blend (c).

10.2.25 CMULTIRES

```
ListType CMULTIRES( CurveType Crv, NumericType Discont )
```

Computes a multiresolution decomposition of curve **Crv** using least squares approximation. The resulting list of curves describes an hierarchy of curves in linear subspaces of the space **Crv** was in that can be sum algebraically to form **Crv**. Each of the curves in the hierarchy is a least squares approximation of **Crv** in the subspace it is defined in. **Discont** is a boolean flat that controls the way tangent discontinuities are computed throughout the decomposition.

Example:

```
MRCrv = CMULTIRES( Animal, false );

sum = nth( MRCrv, 1 );
MRCrvs = list( sum * tx( 3.0 ) );
for ( ( i = 2 ), 1, sizeof( MRCrv ),
    sum = symbsum( sum, nth( MRCrv, i ) ):
    snoc( sum * tx( ( 3 - i ) * 1.5 ), MRCrvs )
);

All = MRCrvs * sc ( 0.25 );
view( All, on );
```

Computes a multiresolution decomposition to curve **CrossSec** as **MRCrv** and display all the decomposition levels by summing them all up. The use of **none** as on object name allows one to

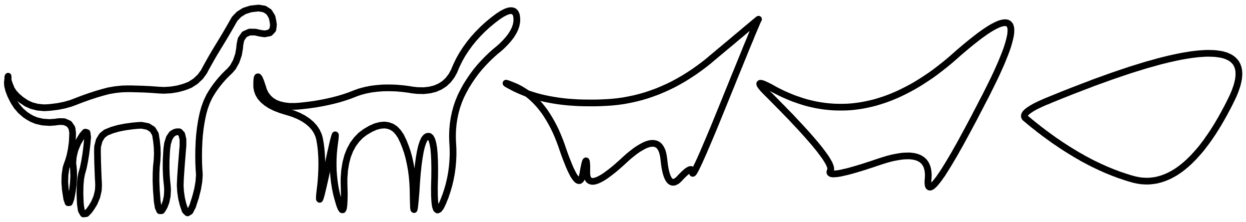


Figure 14: A multiresolution decomposition of a curve of an animal using list squares. Original curve is shown on the left.

display an object in the display device without replacing the previous object in the display device, carrying the same name.

creates two metamorphosis animation sequences, one that is based on a convex blend and one that is based on corner/edge cutting scheme. See Figure 14.

10.2.26 CNORMAL

```
VectorType CNORMAL( CurveType Crv, NumericType TParam )
```

Computes the normal vector to curve **Crv** at the parameter values **TParam**. The returned vector has a unit length.

Example:

```
Normal = CNORMAL( Crv, 0.5 );
```

computes the normal to **Crv** at the parameter value 0.5. See also CNRMLCRV.

10.2.27 CNRMLCRV

```
CurveType CNRMLCRV( CurveType Crv )
```

Symbolically computes a vector field curve representing the non-normalized normals of the given curve. That is a normal vector filed, evaluated at t, provides a vector in the direction of the normal of the original curve at t. The normal curve computed is in fact equal to kN where k is the curvature of **Crv** and N is its normal.

Example:

```
NrmlCrv = CNRMLCRV( Crv );
```

10.2.28 COERCE

```
AnyType COERCE( AnyType Object, ConstantType NewType )
```

Provides a coercion mechanism between different objects or object types. PointType, VectorType, PlaneType, CtlPtType can be all coerced to each other by using the **NewType** of POINT_TYPE, VECTOR_TYPE, PLANE_TYPE, or one of E1-E5, P1-P5 (CtlPtType). Similarly, CurveType, SurfaceType, TrimSrfType, and TrivarType can all be coerced to hold different CtlPtType of control points, or even different open end conditions from KV_PERIODIC to KV_FLOAT to KV_OPEN. If a scalar (E1 or P1) curve is coerced to E2 or P2 curve or a scalar (E1 or P1) surface is coerced to E3 or P3 surface, the Y (YZ) coordinate(s) is (are) updated to hold the parametric domain of the curve (surface).

Example:

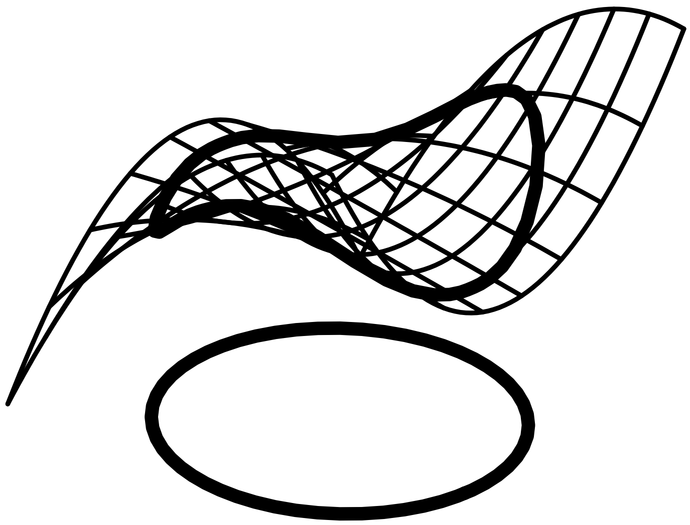


Figure 15: A circle in the parametric space of the freefrom surface is composed to create a closed loop curve on the surface using COMPOSE.

```
CrvE2 = COERCE( Crv, E2 );
```

coerce **Crv** to a new curve that will have an E2 CtlPtType control points. Coerction of a projective curve (P1-P5) to a Euclidean curve (E1-E5) does not preseve the shape of the curve.

10.2.29 COMPOSE

```
CurveType COMPOSE( CurveType Crv1, CurveType Crv2 )
```

or

```
CurveType COMPOSE( SurfaceType Srf, CurveType Crv )
```

Symbolically compute the composition curve **Crv1(Crv2(t))** or **Srf(Crv(t))**. In **Crv1(Crv2(t)**, **Crv1** can be any curve while **Crv2** must be a one-dimensional curve that is either E1 or P1. In **Srf(Crv(t))**, **Srf** can be any surface, while **Crv** must be a two-dimensional curve, that is either E2 or P2. Both **Crv2** in the curve's composition, and **Crv** is the surface's composition must be contained in the curve or surface parametric domain.

Example:

```
srf = sbezier( list( list( ctlpt( E3, 0.0, 0.0, 0.0 ),
                           ctlpt( E3, 0.0, 0.5, 1.0 ),
                           ctlpt( E3, 0.0, 1.0, 0.0 ) ),
                     list( ctlpt( E3, 0.5, 0.0, 1.0 ),
                           ctlpt( E3, 0.5, 0.5, 0.0 ),
                           ctlpt( E3, 0.5, 1.0, 1.0 ) ),
                     list( ctlpt( E3, 1.0, 0.0, 1.0 ),
                           ctlpt( E3, 1.0, 0.5, 0.0 ),
                           ctlpt( E3, 1.0, 1.0, 1.0 ) ) ) );
crv = coerce( circle( vector( 0.0, 0.0, 1.0 ), 0.4 ), p2 ) *
      trans( vector( 0.5, 0.5, 0.0 ) );
comp_crv = COMPOSE( srf, crv );
```

compose a circle **Crv** to be on the surface **Srf**. See Figure 15.

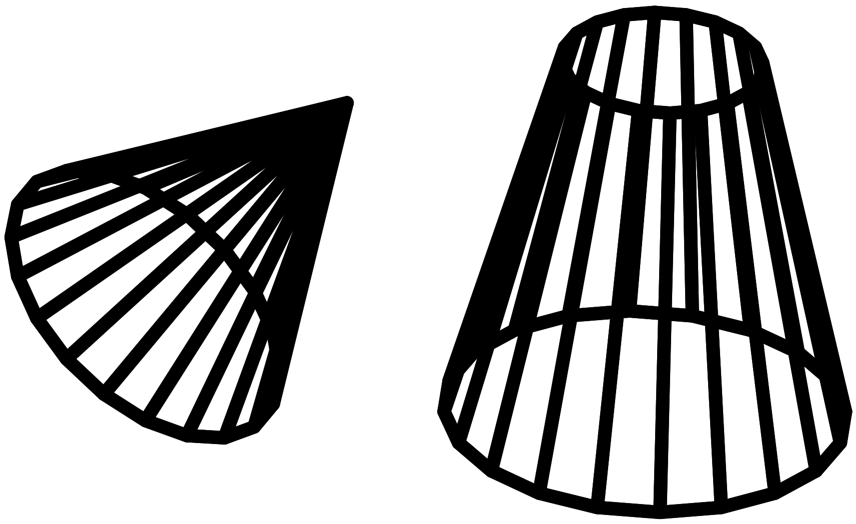


Figure 16: A cone (left) can be constructed using the CONE constructor and a truncated cone (right) using the constructor CONE2.

10.2.30 CON2

```
PolygonType CON2( VectorType Center, VectorType Direction,
                  NumericType Radius1, NumericType Radius2 )
```

Creates a truncated CONE geometric object, defined by **Center** as the center of the main base of the CONE, **Direction** as both the CONE's axis and the length of CONE, and the two radii **Radius1/2** of the two bases of the CONE.

Unlike the regular cone (CONE) constructor which has inherited discontinuities in its generated normals at the apex, CON2 can be used to form a (truncated) cone with continuous normals. See RESOLUTION for the accuracy of the CON2 approximation as a polygonal model.

Example:

```
Cone2 = CON2( vector( 0, 0, -1 ), vector( 0, 0, 4 ), 2, 1 );
```

constructs a truncated cone with bases parallel to the XY plane at $Z = -1$ and $Z = 3$, and with radii of 2 and 1 respectively. See Figure 16.

10.2.31 CONE

```
PolygonType CONE( VectorType Center, VectorType Direction,
                  NumericType Radius )
```

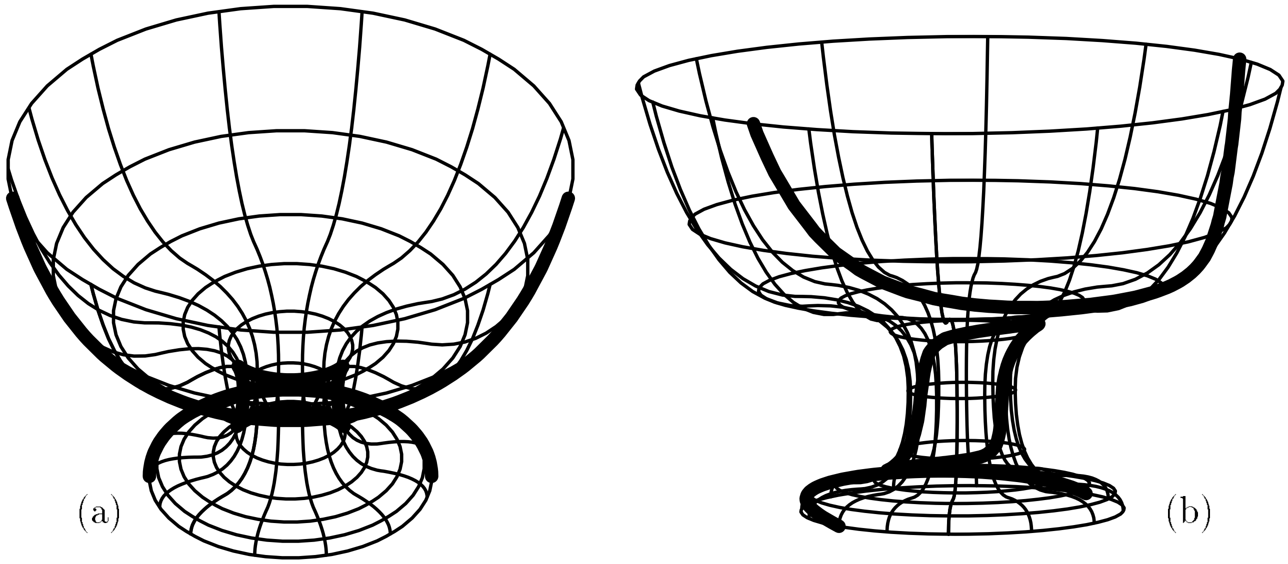
Creates a CONE geometric object, defined by **Center** as the center of the base of the CONE, **Direction** as the CONE's axis and height, and **Radius** as the radius of the base of the CONE. See RESOLUTION for accuracy of the CONE approximation as a polygonal model.

Example:

```
Cone1 = CONE( vector( 0, 0, 0 ), vector( 1, 1, 1 ), 1 );
```

constructs a cone based in an XY parallel plane, centered at the origin with radius 1 and with tilted apex at (1, 1, 1).

See also CON2. See Figure 16.



(a) (b)

Figure 17: Computes the silhouette of a freeform glass surface from viewing direction (1, 1, 1). In (a), the original view is seen. In (b), a different, general, view is provided.

10.2.32 CONTOUR

```
PolygonType CONTOUR( SurfaceType ContouredSrf, PlaneType ContourPlane )
```

or

```
PolygonType CONTOUR( SurfaceType ContouredSrf, PlaneType ContourPlane,
                     SurfaceType MappedSrf )
```

Contours surface **ContouredSrf** by intersecting plane **ContourPlane** with a polygonal approximating of **ContouredSrf** with resolution set via variable RESOLUTION. If **ContouredSrf** is a scalar field surface of type E1 or P1 and **MappedSrf** is provided, **ContouredSrf** is contoured above its parametric domain (U is X, V is Y) and the resulting parametric curve is composed with **MappedSrf** to yield the returned value.

Example:

```
resolution = 20;
nglass = snrmlsrf( glass ) * vector( 1, 1, 1 );

sils = contour( nglass, plane( 1, 0, 0, 0 ), glass );
color( sils, cyan );
attrib( sils, "dwidth", 4 );

view( list( axes, glass, sils ), on );
```

Computes the normal field of the surface **glass**, project it onto the viewing direction of (1, 1, 1) and contour the resulting scalar field with the plane X = 0, to extract the silhouette curves from viewing direction (1, 1, 1). See Figure 17.

10.2.33 CONVEX

```
PolygonType CONVEX( PolygonType Object )
```

```
or

ListType CONVEX( ListType Object )
```

Converts non-convex polygons in **Object**, into convex ones. New vertices are introduced into the polygonal data during this process. The Boolean operations require the input to have convex polygons only (although it may return non convex polygons...) and it automatically converts non-convex input polygons to convex ones, using this same routine.

However, some external tools (like irit2ray and poly3d-h) require convex polygons. This function must be used on the objects to guarantee that only convex polygons are saved into data files for these external tools.

Example:

```
CnvxObj = CONVEX( Obj );
save( "data", CnvxObj );
```

converts non-convex polygons into convex ones, so that the data file can be used by external tools requiring convex polygons.

10.2.34 COORD

```
AnyType COORD( AnyType Object, NumericType Index )
```

Extracts an element from a given **Object**, at index **Index**. From a PointType, VectorType, PlaneType, CtlPtType and MatrixType, a NumericType is returned with **Index** 0 for the X axis, 1 for the Y axis etc. **Index** 0 denotes the weight of CtlPtType. For a PolygonType that contains more than one polygon, the **Index**th polygon is returned. For a PolygonType that contains a single Polygon, the **Index**th vertex is returned. For a CurveType or a SurfaceType, the **Index**th CtlPtType is returned. For a ListType, COORD behaves like NTH and returns the **Index**th object in the list. For a StringType, the **Index**th character is returned as its ASCII numeric code.

Example:

```
a = vector( 1, 2, 3 );
vector( COORD( a, 0 ), COORD( a, 1 ), COORD( a, 2 ) );

a = ctlpt( P2, 6, 7, 8, 9 );
ctlpt( P3, coord( a, 0 ), coord( a, 1 ), coord( a, 2 ), coord( a, 3 ) );

a = plane( 10, 11, 12, 13 );
plane( COORD( a, 0 ), COORD( a, 1 ), COORD( a, 2 ), COORD( a, 3 ) );
```

constructs a vector/ctlpt/plane and reconstructs it by extracting the constructed scalar components of the objects using COORD.

See also COERCE.

10.2.35 CRAISE

```
CurveType CRAISE( CurveType Curve, NumericType NewOrder )
```

Raise **Curve** to the **NewOrder** Order specified.

Example:

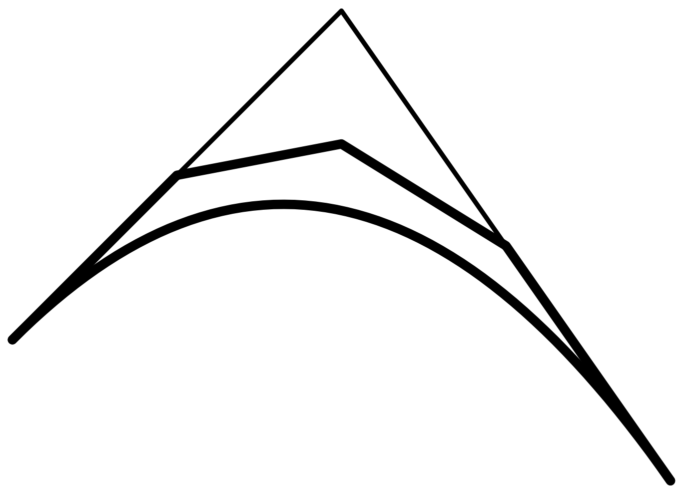


Figure 18: Raises a 90 degrees corner quadratic Bezier curve to a quintic using CRAISE. The control polygons are also shown.

```
Crv = cbezier( list( ctlpt( E2, -0.7,  0.3 ),
                     ctlpt( E2,  0.0,  1.0 ),
                     ctlpt( E2,  0.7,  0.0 ) ) );
Crv2 = CRAISE( Crv, 5 );
```

raises the 90 degrees corner Bezier curve **Crv** to be a quartic. See Figure 18.

10.2.36 CREFINE

```
CurveType CREFINE( CurveType Curve, NumericType Replace, ListType KnotList )
```

Provides the ability to **Replace** a knot vector of **Curve**, or refine it. **KnotList** is a list of knots to refine **Curve** at. All knots should be contained in the parametric domain of the **Curve**. If the knot vector is replaced, the length of **KnotList** should be identical to the length of the original knot vector of the **Curve**. If **Curve** is a Bezier curve, it is automatically promoted to be a Bspline curve.

Example:

```
Crv2 = CREFINE( Crv, FALSE, list( 0.25, 0.5, 0.75 ) );
```

refines **Crv** and adds three new knots at 0.25, 0.5, and 0.75. See Figure 19.

10.2.37 CREGION

```
CurveType CREGION( CurveType Curve, NumericType MinParam,
                                                     NumericType MaxParam )
```

Extracts a region from **Curve** between **MinParam** and **MaxParam**. Both **MinParam** and **MaxParam** should be contained in the parametric domain of the **Curve**.

Example:

```
SubCrv = CREGION( Crv, 0.3, 0.6 );
```

extracts the region from **Crv** from the parameter value 0.3 to the parameter value 0.6. See Figure 20.

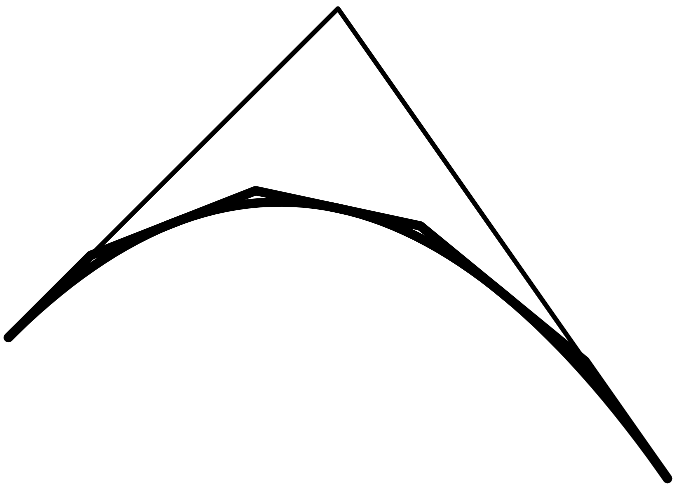


Figure 19: Refines a 90 degrees corner quadratic Bezier curve at three interior knots (result is a Bspline curve) using CREFINE. The control polygons are also shown.

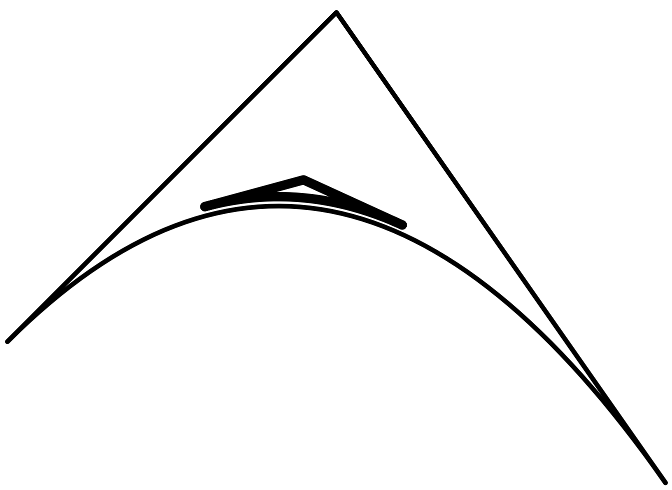


Figure 20: Extracts a sub region from a curve using CREGION.

10.2.38 CREPARAM

```
CurveType CREPARAM( CurveType Curve, NumericType MinParam,
                                                      NumericType MaxParam )
```

Reparametrize **Curve** over a new domain from **MinParam** to **MaxParam**. This operation does not affect the geometry of the curve and only affine transforms its knot vector. A Bezier curve will automatically be promoted into a Bspline curve by this function.

Example:

```
arc1 = arc( vector( 0.0, 0.0, 0.0 ),
            vector( 0.5, 2.0, 0.0 ),
            vector( 1.0, 0.0, 0.0 ) );
crv1 = arc( vector( 1.0, 0.0, 0.75 ),
            vector( 0.75, 0.0, 0.7 ),
            vector( 0.5,  0.0, 0.85 ) ) +
       arc( vector( 0.5,  0.0, 0.75 ),
            vector( 0.75, 0.0, 0.8 ),
            vector( 1.0,  0.0, 0.65 ) );

arc1 = CREPARAM( arc1, 0, 10 );
crv1 = CREPARAM( crv1, 0, 10 );
```

Sets the domain of the given two curves to be from zero to ten. The Bezier curve arc1 is promoted to a Bspline curve.

10.2.39 CROSSEC

```
PolygonType CROSSEC( PolygonType Object )
```

This feature is NOT implemented.

10.2.40 CRVLNDST

```
NumericType CRVLNDST( CurveType Crv, PointType PtOnLine, VectorType LnDir,
                                NumericType IsMinDist, NumericType Epsilon )
```

or

```
ListType CRVLNDST( CurveType Crv, PointType PtOnLine, VectorType LnDir,
                               NumericType IsMinDist, NumericType Epsilon )
```

Computes the closest (if **IsMinDist** is TRUE, farthest if FALSE) point on **Curve** to the line specified by **PtOnLine** and **LnDir** as a point on the line and a line direction. Since this operation is partially numeric, **Epsilon** is used to set the needed accuracy. It returns the parameter value of the location on **Crv** closest to the line. If, however, **Epsilon** is negative, -**Epsilon** is used instead, and all local extrema in the distance function are returned as a list (both minima and maxima). If the line and the curve intersect, the point of intersection is returned as the minimum.

Example:

```
Param = CRVLNDST( Crv, linePt, lineVec, TRUE, 0.001 );
```

finds the closest point on **Crv** to the line defined by **linePt** and **lineVec**. See Figure 21.

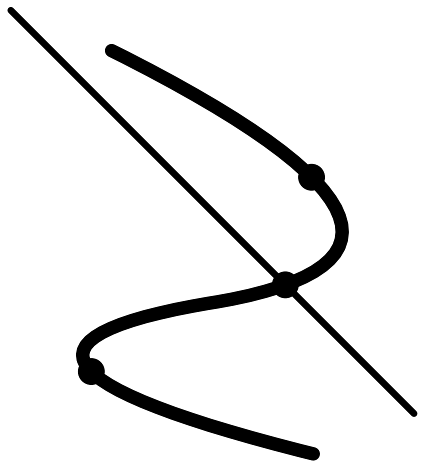


Figure 21: Computes the locations on the freeform curve with local extreme distance to the given line, using CRVLNDST.

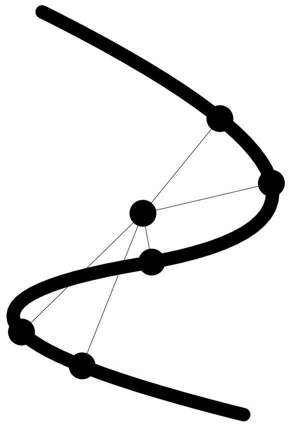


Figure 22: Computes the locations on the freeform curve with local extreme distance to the given point, using CRVPTDST.

10.2.41 CRVPTDST

```
NumericType CRVPTDST( CurveType Crv, PointType Point, NumericType IsMinDist,
                                                       NumericType Epsilon )
```

or

```
ListType CRVPTDST( CurveType Crv, PointType Point, NumericType IsMinDist,
                                                       NumericType Epsilon )
```

Computes the closest (if **IsMinDist** is TRUE, farthest if FALSE) point on **Crv** to **Point**. Since this operation is partially numeric, **Epsilon** is used to set the needed accuracy. It returns the parameter value of the location on **Crv** closest to **Point**. If, however, **Epsilon** is negative, -**Epsilon** is used instead, and all local extrema in the distance function are returned as a list (both minima and maxima).

Example:

```
Param = CRVPTDST( Crv, Pt, FALSE, 0.0001 );
```

finds the farthest point on **Crv** from point **Pt**. See Figure 22.

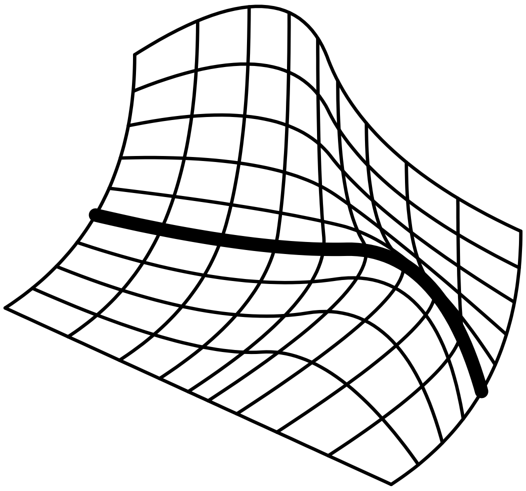


Figure 23: Extracts an isoparametric curve from the given surface, using CSURFACE.

10.2.42 CSURFACE

```
CurveType CSURFACE( SurfaceType Srf, ConstantType Direction,
                                             NumericType Param )
```

Extract an isoparametric curve out of **Srf** in the specified **Direction** (ROW or COL) at the specified parameter value **Param**. **Param** must be contained in the parametric domain of **Srf** in **Direction** direction. The returned curve is *in* the surface **Srf**. It is equal to,

$$C(t) = S(t, v_0) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{ij} B_i(t) B_j(v_0) = \sum_{i=0}^{m} \left(\sum_{j=0}^{n} P_{ij} B_j(u_0) \right) B_i(t) = \sum_{i=0}^{m} Q_i B_i(t), \tag{7}$$

where $Q_i = \sum_{j=0}^{n} P_{ij} B_j(u_0)$ are the coefficients of the returned curve, and similar for the other parametric direction $S(u_0, t)$. **param** is v_0 is equation (7)

Example:

```
Crv = CSURFACE( Srf, COL, 0.45 );
```

extracts an isoparametric curve in the COLumn direction at the parameter value of 0.15 from surface **Srf**. See also CMESH, COMPOSE. See Figure 23.

10.2.43 CTANGENT

```
VectorType CTANGENT( CurveType Curve, NumericType Param )
```

Computes the tangent vector to **Curve** at the parameter value **Param**. The returned vector has a unit length.

Example:

```
Tang = CTANGENT( Crv, 0.5 );
```

computes the tangent vector to **Crv** at the parameter value of 0.5.

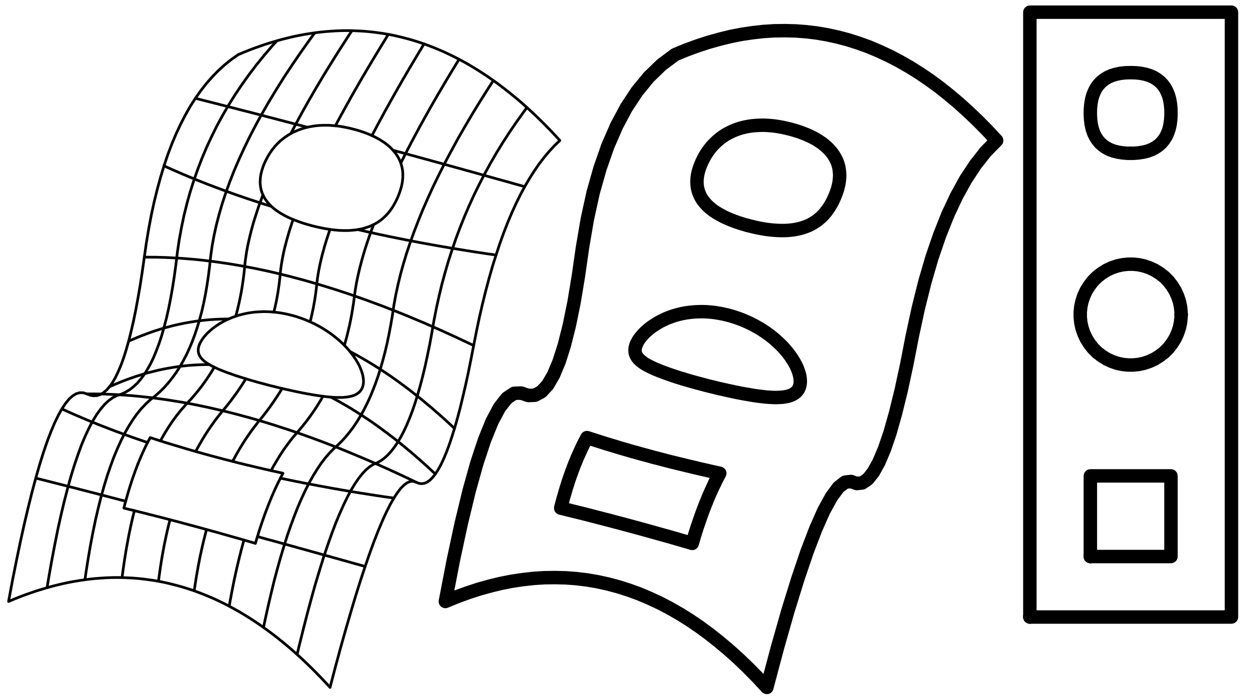


Figure 24: Extracts the trimming curves in Euclidean space (middle) and parametric space (right) of a trimmed surface (left), using CTRIMSRF.

10.2.44 CTLPT

```
CPt = CTLPT( ConstantType PtType, NumericType Coord1, ... )
```

Constructs a single control point to be used in the construction of curves and surfaces. Points can have from one to five dimensions, and may be either Euclidean or Projective (rational). Points' type is set via the constants E1 to E5 and P1 to P5. The coordinates of the point are specified in order, weight is first if rational.

Examples:

```
CPt1 = CTLPT( E3, 0.0, 0.0, 0.0 );
CPt2 = CTLPT( P2, 0.707, 0.707, 0.707 );
```

constructs an **E3** point at the origin and a P2 rational point with a weight of 0.707. The Projective Pi points are specified as CTLPT(Pn, W, W X1, ... , W Xn).

10.2.45 CTRIMSRF

```
ListType CTRIMSRF( TrimSrfType TSrf, NumericType Parametric )
```

Extract the trimming curves of a trimmed surface **TSrf**. If **Parametric** is not zero, then the trimming curves are extracted as parametric space curves of **TSrf**. Otherwise, the trimming curves are evaluated into Euclidean space as curves on surface **TSrf**.

Example:

```
TrimCrvs = CTRIMSRF( TrimSrf, FALSE );
```

extracts the trimming curves of **TrimSrf** as Euclidean curves on **TrimSrf**. See Figure 24.

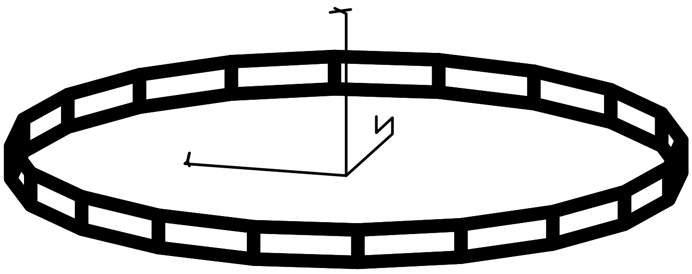


Figure 25: A cylinder primitive can be constructor using the CYLIN constructor.

10.2.46 CYLIN

```
PolylineType CYLIN( VectorType Center, VectorType Direction,
                    NumericType Radius )
```

Creates a CYLINder geometric object, defined by **Center** as center of the base of the CYLINder, **Direction** as the CYLINder's axis and height, and **Radius** as the radius of the base of the CYLINder. See RESOLUTION for the accuracy of the CYLINder approximation as a polygonal model.

Example:

```
Cylinder1 = CYLIN( vector( 0, 0, 0 ), vector( 1, 0, 0 ), 10 );
```

constructs a cylinder along the X axis from the origin to $X = 10$. See Figure 25.

10.2.47 CZEROS

```
ListType CZEROS( CurveType Crv, NumericType Epsilon, NumericType Axis )
```

Computes the zero set of the given **Crv** in the given axis (1 for X, 2 for Y, 3 for Z). Since this computation is numeric, an **Epsilon** is also required to specify the desired tolerance. It returns a list of all the parameter values (NumericType) the curve is zero.

Example:

```
xzeros = CZEROS( cb, 0.001, 1 );
pt_xzeros = nil();
pt = nil();
for ( i = 1, 1, sizeof( xzeros ),
        pt = ceval( cb, nth( xzeros, i ) ):
        snoc( pt, pt_xzeros )
    );
interact( list( axes, cb, pt_xzeros ), 0 );
```

Computes the **X** zero set of curve **cb** with error tolerance of **0.001**. This set is then scanned in a loop and evaluated to the curve's locations, which are then displayed. See also CINFLECT. See Figure 26.

10.2.48 EVOLUTE

```
CurveType EVOLUTE( CurveType Curve )
```

or

```
SurfaceType EVOLUTE( SurfaceType Curve )
```

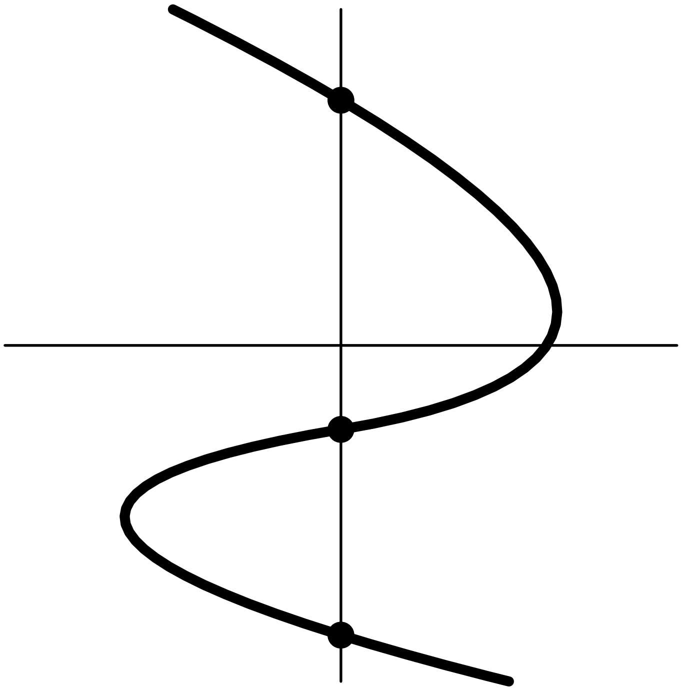


Figure 26: Computes the zero set of a given freeform curve, in the given axis, using CZEROS.

Computes the evolute of a curve or a surface. For curves, the evolute is defined as,

$$E(t) = C(t) + \frac{N(t)}{\kappa(t)}, \tag{8}$$

where $N(t)$ is the unit normal of $C(t)$ and $k(t)$ is its curvature.

$E(t)$ is computed symbolically as the symbolic sum of $C(t)$ and $\frac{N(t)}{\kappa(t)}$ where the latter is,

$$\begin{aligned} \frac{N(t)}{\kappa(t)} &= \frac{\kappa(t)N(t)}{k^2(t)} \\ &= \frac{(C'(t) \times C''(t)) \times C'(t)}{\|C'(t)\|^4} \frac{\|C'(t)\|^6}{(C'(t) \times C''(t))^2} \\ &= \frac{((C'(t) \times C''(t)) \times C'(t)) \, \|C'(t)\|^2}{(C'(t) \times C''(t))^2} \end{aligned} \tag{9}$$

For surfaces, this function computes the mean evulate which is equal to,

$$E(u, v) = S(u, v) + \frac{n(u, v)}{2H(u, v)}, \tag{10}$$

where $n(u, v)$ is the unit normal of $S(u, v)$ and $H(u, v)$ is the mean curvature.

$E(u, v)$ is computed symbolically.

The result of this symbolic computation is exact (upto machine precision) unlike a similar operations that are only approximated, like the OFFSET or the AOFFSET.

Example:

```
crv = cbspline( 3,
                list( ctlpt( E3, -1.0,  0.1,  0.2 ),
                      ctlpt( E3, -0.1,  1.0,  0.1 ),
```

Figure 27: The evolute (thick) of a freeform curve (thin) can be computed using EVOLUTE.

```
                      ctlpt( E3,  0.1,  0.1,  1.0 ),
                      ctlpt( E3,  1.0,  0.1,  0.1 ),
                      ctlpt( E3,  0.1,  1.0,  0.2 ) ),
                list( KV_OPEN ) );
cev = EVOLUTE( Crv );
```

See Figure 27.

10.2.49 EXTRUDE

```
PolygonType EXTRUDE( PolygonType Object, VectorType Dir )
```

or

```
SurfaceType EXTRUDE( CurveType Object, VectorType Dir )
```

Creates an extrusion of the given **Object**. If **Object** is a PolygonObject, its first polygon is used as the base for the extrusion in **Dir** direction, and a closed PolygonObject is constructed. If **Object** is a CurveType, an extrusion surface is constructed instead, which is *not* a closed object (the two bases of the extrusion are excluded, and the curve may be open by itself).

Direction **Dir** cannot be coplanar with the polygon plane. The curve may be nonplanar.

Example:

```
Cross = cbspline( 3,
                  list( ctlpt( E2, -0.018, 0.001 ),
                        ctlpt( E2,  0.018, 0.001 ),
                        ctlpt( E2,  0.019, 0.002 ),
                        ctlpt( E2,  0.018, 0.004 ),
                        ctlpt( E2, -0.018, 0.004 ),
                        ctlpt( E2, -0.019, 0.001 ) ),
                  list( KV_OPEN ) );
Cross = Cross + -Cross * scale( vector( 1, -1, 1 ) );
Napkin = EXTRUDE( Cross * scale( vector( 1.6, 1.6, 1.6 ) ),
                  vector( 0.02, 0.03, 0.2 ) );
```

constructs a closed cross section **Cross** by duplicating one half of it in reverse and merging the two sub-curves. **Cross** is then used as the cross-section for the extrusion operation. See Figure 28.

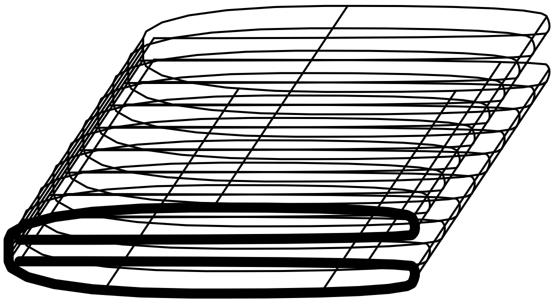


Figure 28: An extrusion of a freeform curve using EXTRUDE to create a freeform surface.

10.2.50 FFCOMPAT

```
FFCOMPAT( CurveType Crv1, CurveType Crv2 )
```

or

```
FFCOMPAT( SurfaceType Srf1, SurfaceType Srf2 )
```

Makes the given two curves or surfaces compatible by making them share the same point type, same curve type, same degree, and the same continuity. Same point type is gained by promoting a lower dimension into a higher one, and non-rational to rational points. Bezier curves are promoted to Bspline curves if necessary, for curve type compatibility. Degree compatibility is achieved by raising the degree of the lower order curve. Continuity is achieve by refining both curves to the space with the same (unioned) knot vector. This function returns nothing and compatibility is made *in place*.

Example:

```
FFCOMPAT( Srf1, Srf2 );
```

See also CMORPH and SMORPH.

10.2.51 FFEXTREME

```
CtlPtType FFEXTREME( CurveType Crv, NumericType Minimum )
```

or

```
CtlPtType FFEXTREME( SurfaceType Srf, NumericType Minimum )
```

Computes a bound on the extreme values a curves **Crv** or surface **Srf** can assume. Returned control points provides a bound on the minimum (maximum) values that can be assumed if **Minimum** is TRUE (FALSE).

Example:

```
Bound = FFEXTREME( Srf, false );
```

Computes a bound on the maximal values Srf can assume.

10.2.52 FFMATCH

```
FFMATCH( CurveType Crv1, CurveType Crv2, NumericType Reduce,
         NumericType Samples, NumericType ReparamOrder,
         NumericType Rotate, NumericType NormType )
```

Computes a reparametrization to **Crv2** so it fits **Crv1**, the best under some prescribed norm, **NormType**. Currently the following norms are valid for **NormType**

Value	Description
1	Suitable for ruled and blended curves, for modeling. See RULEDSRF.
2	Suitable for metamorphosis of curves. See CMORPH.
3	Distance norm in "walking the dog" notion.
4	Bisector (skeleton) matching norm for two curves.

Whenever negative norms can result (for example, in cases were self intersection cannot be prevented in ruled surface constructions), one can allow negativity with no extra penalty by applaying neative **NormType**. Use of positive only norms would yield no output at all if no matching with positive weights can be established whereas allowing negative norm values would result in the globally optimal result, but with possibly self intersectiions.

The reparametrization is computed by sampling a fix set of size **Samples** off both curves, and fitting a Bspline curve of length **Reduce** as the reparametrization curve. Hence, **Reduce** must be less than or equal to **Samples**. The reparametrization curve will have order of **ReparamOrder**. If **Rotate** is TRUE or ON, then attempt is made to rotate the reparametrization of the curves. Rotation can be used on closed curves only.

See RULEDSRF and CMORPH for examples.

10.2.53 FFMERGE

```
CurveType FFMERGE( ListType E1Curves, NumericType PointType )@
```

or

```
SurfaceType FFMERGE( ListType E1Surfaces, NumericType PointType )
```

Merges the scalar curves in the list of curves **E1Curves** or list of surfaces **E1Surfaces** to one vector curve/surface of point type **PointType**.

Example:

```
Srf = FFMERGE( list( SrfW, SrfX, SrfY ), P2 );
```

merges three scalar surfaces into a single surface with point type P2. See also FFSPLIT, FFPTTYPE.

10.2.54 FFPTTYPE

```
NumericType FFMERGE( CurveType Crv )
```

or

```
NumericType FFMERGE( SurfaceType Srf )
```

or

```
NumericType FFMERGE( TrivarType )
```

Returns the point type (E2, P4 etc.) of the given freeform.

10.2.55 FFSPLIT

```
ListType FFSPLIT( CurveType Crv )
```

or

```
ListType FFSPLIT( SurfaceType Srf )
```

Splits the given curve **Crv** or surface **Srf** into its scalar components that are returned as a list of curves/surfaces.
Example:

```
E1Srfs = FFSPLIT( circle( vector( 0, 0, 0 ), 1 ) );
```

splits the circle which is a curve in P3 into four scalar curves (W, X, Y, Z) that are returned in a single list. See also FFMERGE, FFPTTYPE.

10.2.56 GBOX

```
PolygonType GBOX( VectorType Point,
                  VectorType Dx, VectorType Dy, VectorType Dz )
```

Creates a parallelepiped - Generalized BOX polygonal object, defined by **Point** as base position, and **Dx, Dy, Dz** as 3 3D vectors to define the 6 faces of this generalized BOX. The regular BOX object is a special case of GBOX where **Dx** = vector(Dx, 0, 0), **Dy** = vector(0, Dy, 0), and **Dz** = vector(0, 0, Dz).
Dx, **Dy**, **Dz** must all be independent in order to create an object with positive volume.
Example:

```
GB = GBOX(vector(0.0, -0.35, 0.63), vector(0.5, 0.0, 0.5),
                                    vector(-0.5, 0.0, 0.5),
                                    vector(0.0, 0.7, 0.0));
```

See Figure 29.

10.2.57 GETLINE

```
AnyType GETLINE( NumericType RequestedType )
```

Provides a method to get input from keyboard within functions and or subroutines. **RequestedType** can be one of NUMERIC_TYPE, POINT_TYPE, VECTOR_TYPE, or PLANE_TYPE in which the entered line will be parsed into one, three, or four numeric values (sperated by either spaces or commas) and the proper object will be created and returned. In any other case, including failure to parse the numeric input, a STRING_TYPE object will be constructed from the entered line.
Example:

```
Pt = GETLINE( point_type );
```

to read one point (three numeric values) from stdin.

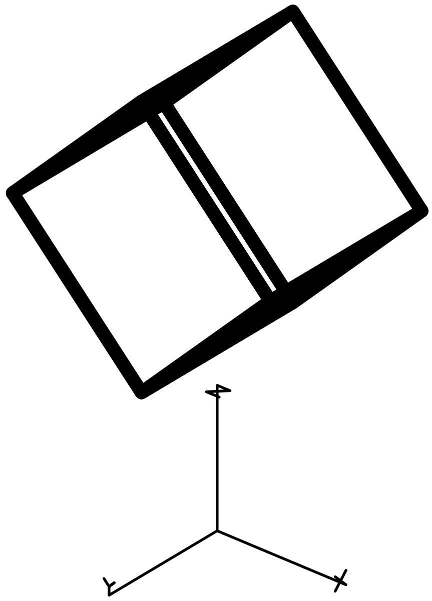


Figure 29: A warped box in a general position can be constructed using the GBOX constructor.

10.2.58 GPOLYGON

```
PolygonType GPOLYGON( GeometryTreeType Object, NumericType Normals )
```

Approximates all Surface(s)/Trimmed surface(s)/Trivariate(s) in **Object** with polygons using the RESOLUTION and FLAT4PLY variables. The larger the RESOLUTION is, the finer (more polygons) the resulting approximation will be.

FLAT4PLY is a Boolean flag controlling the conversion of an (almost) flat patch into four (TRUE) or two (FALSE) polygons. Normals are computed to polygon vertices using surface normals, so Gouraud or Phong shading can be exploited. It returns a single polygonal object.

If **Normals** is set, surface normals will be evaluated at the vertices. Otherwise flat shading and constant normals across polygons are assumed.

Example:

```
Polys = GPOLYGON( list( Srf1, Srf2, Srf3 ), off );
```

Converts to polygons the three surfaces **Srf1**, **Srf2**, and **Srf3** with no normals.

10.2.59 GPOLYLINE

```
PolylineType GPOLYLINE( GeometryTreeType Object, NumericType Optimal )
```

Converts all Curves(s), (Trimmed) Surface(s), and Trivariate(s) **Object** into polylines using the RESOLUTION variable. The larger the RESOLUTION is, the finer the resulting approximation will be. It returns a single polyline object.

If **Optimal** is false, the points are sampled at equally spaced interval in the parametric space. If **Optimal** true, a better, more expensive computationally, algorithm is used to derive optimal sampling locations as to minimize the maximal distance between the curve and piecewise linear approximation (L infinity norm).

Example:

```
Polys = GPOLYLINE( list( Srf1, Srf2, Srf3, list( Crv1, Crv2, Crv3 ) ),
                   on );
```

converts to polylines the three surfaces **Srf1**, **Srf2**, and **Srf3** and the three curves **Crv1**, **Crv2**, and **Crv3**.

10.2.60 HERMITE

```
SurfaceType HERMITE( CurveType Bndry1, CurveType Bndry2,
                     CurveType Tan1, CurveType Tan2 )
```

or

```
CurveType HERMITE( PointType Bndry1, PointType Bndry2,
                   VectorType Tan1, VectorType Tan2 )
```

Constructs a cubic fit between **Bndry1** and **Bndry2** so that first derivative continuity constraints, as prescribed by **Tan1** at **Bndry1** and **Tan2** at **Bndry2**, are preserved.

Returns either a curve or a surface, according to type of input parameters.

Example:

```
h00 = HERMITE( point( 0, 0, 0 ),
               point( 1, 1, 0 ),
               vector( 1, 0, 0 ),
               vector( 1, 0, 0 ) );
```

Constructs a curve in the shape of the first basis function of the cubic Hermite basis functions.

10.2.61 LOFFSET

```
CurveType LOFFSET( CurveType Crv, NumericType OffsetDistance,
                   NumericType NumOfSamples, NumericType NumOfDOF,
                   NumericType Order )
```

Approximate an offset of **OffsetDistance** by sampling **NumOfSamples** samples along the offset curve and least square fitting them using a Bspline curve of order **Order** and **NumOfDOF** control points.

Example:

```
OffCrv1 = LOFFSET( Crv, -0.4, 100, 10, 4 );
```

See also OFFSET, AOFFSET, and MOFFSET.

10.2.62 MERGPOLY

```
PolygonType MERGEPOLY( ListType PolyList )
```

Merges a set of polygonal objects in **PolyList** list to a single polygonal object. All elements in **ObjectList** must be of PolygonType type. This function performs the same operation as the overloaded ^ operator would, but might be more convenient to use under some circumstances.

Example:

```
Vrtx1 = vector( -3, -2, -1 );
Vrtx2 = vector( 3, -2, -1 );
Vrtx3 = vector( 3, 2, -1 );
Vrtx4 = vector( -3, 2, -1 );
Poly1 = poly( list( Vrtx1, Vrtx2, Vrtx3, Vrtx4 ), false );

Vrtx1 = vector( -3, 2, 1 );
```

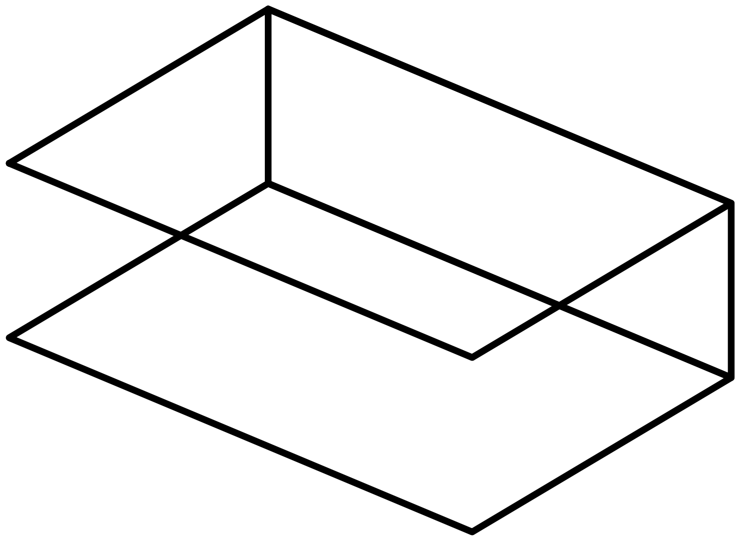


Figure 30: Individual polygons can be merged into a complete model using MERGEPOLY.

```
Vrtx2 = vector( 3, 2, 1 );
Vrtx3 = vector( 3, -2, 1 );
Vrtx4 = vector( -3, -2, 1 );
Poly2 = poly( list( Vrtx1, Vrtx2, Vrtx3, Vrtx4 ), false );

Vrtx1 = vector( -3, -2, 1 );
Vrtx2 = vector( 3, -2, 1 );
Vrtx3 = vector( 3, -2, -1 );
Vrtx4 = vector( -3, -2, -1 );
Poly3 = poly( list( Vrtx1, Vrtx2, Vrtx3, Vrtx4 ), false );

PolyObj = MERGEPOLY( list( Poly1, Poly2, Poly3 ) );
```

See Figure 30.

10.2.63 MOFFSET

```
CurveType MOFFSET( CurveType Crv, NumericType OffsetDistance,
                   NumericType AngularError )
```

Computes an offset of **OffsetDistance** with globally bounded error (controlled by **AngularError**). The smaller **AngularError** is, the better the approximation to the offset. The bounded error is achieved by adaptive refinement of the **Crv**. The offset is computed via matching of the tangent fields of the given curve **Crv** and an arc spanning the same angular domain. Further, **AngularError** measures the angular deviation allowed between the two tangent fields.

Example:

```
OffCrv1 = MOFFSET( Crv, -0.4, 10 );
OffCrv2 = MOFFSET( Crv, -0.4, 5 );
```

computes an offset approximation to **Crv** with **OffsetDistance** of -0.4 and **AngularError** of 10 and 5 degrees, respectively. See also OFFSET, AOFFSET, LOFFSET, and FFMATCH.

10.2.64 MOMENT

```
PointType MOMENT( CurveType Crv, 0 );
```

or

```
VectorType MOMENT( CurveType Crv, 1 );
```

Approximates the zero and first moment of curve **Crv**.
Example:

```
a = circle( vector( 0, 0, 0 ), 1 );
a = cregion( a, 0, 1 );
p = moment( a, 0 );
v = moment( a, 1 );
view(list(a, p, v), on);

a = cregion( a, 0, 1 ) * rz( 45 );
p = moment( a, 0 );
v = moment( a, 1 );
view(list(a, p, v), on);
```

computes and displays the zero and first moment of a quarter of a circle in two orientations.

10.2.65 NIL

```
ListType NIL()
```

Creates an empty list so data can be accumulated in it. See CINFLECT or CZEROS for examples. See also LIST and SNOC.

10.2.66 OFFSET

```
CurveType OFFSET( CurveType Crv, NumericType OffsetDistance,
                  NumericType Tolerance, NumericType BezInterp )
```

or

```
SurfaceType OFFSET( SurfaceType Srf, NumericType OffsetDistance,
                    NumericType Tolerance, NumericType BezInterp )
```

or

```
TrimSrfType OFFSET( TrimSrfType TrimSrf, NumericType OffsetDistance,
                    NumericType Tolerance, NumericType BezInterp )
```

Offsets **Crv**, **Srf** or a **TrimSrf**, by translating all the control points in the direction of the normal of the curve or the (trimmed) surface by an **OffsetDistance** amount. Each control point has a *node* parameter value associated with it, which is used to compute the normal. The returned curve or surface only approximates the real offset. If the resulting approximation does not satisfy the accuracy required by **Tolerance**, **Crv** or **Srf** or **TrimSrf** is subdivided and an offset approximation fit is computed to the two halfs. For curves, one can request a Bezier interpolation scheme in the offset

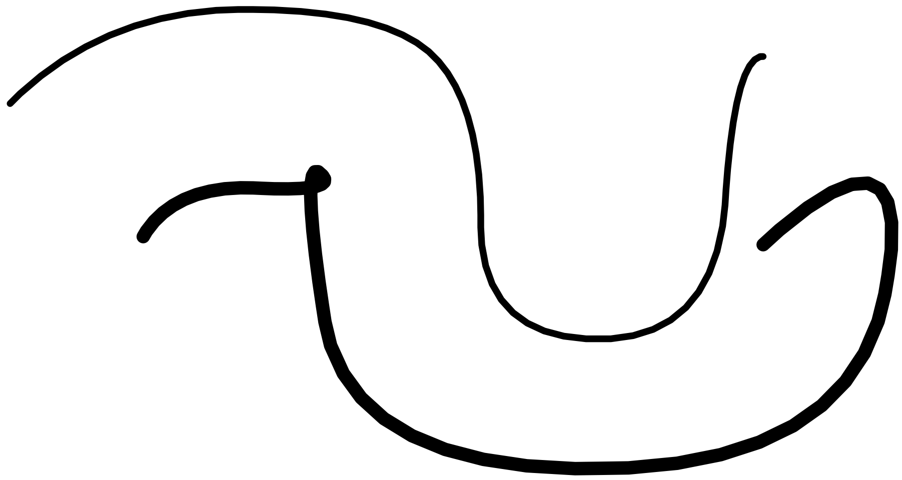


Figure 31: Offset approximation (thick) of a Bspline curve (thin) (See also Figure 3.)

approximation by setting **BezInterp**. The **BezInterp** is not supported yet for (trimmed) surfaces. Negative **OffsetDistance** denotes offset in the reversed direction of the normal.

Example:

```
OffCrv = OFFSET( Crv, -0.4, 0.1, off );
```

offsets **Crv** by the amount of -0.4 in the reversed normal direction, **Tolerance** of 0.1 and no Bezier interpolation. See also AOFFSET and LOFFSET. See Figure 31.

10.2.67 PCIRCLE

```
CurveType PCIRCLE( VectorType Center, NumericType Radius )
```

Same as CIRCLE but approximates the circle as a *polynomial* curve. See CIRCLE.

10.2.68 PDOMAIN

```
ListType PDOMAIN( CurveType Crv )
```

or

```
ListType PDOMAIN( SurfaceType Srf )
```

or

```
ListType PDOMAIN( TrimSrfType TrimSrf )
```

or

```
ListType PDOMAIN( TrivarType TV )
```

Returns the parametric domain of the curve (TMin, TMax) or of a (trimmed) surface (UMin, UMax, VMin, VMax) or of a trivariate function (UMin, UMax, VMin, VMax, WMin, WMax) as a list object.

Example:

```
circ_domain = PDOMAIN( circle( vector( 0.0, 0.0, 0.0 ), 1.0 ) );
```

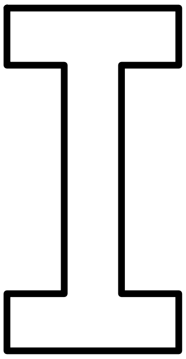


Figure 32: Polygons or polylines can be manually constructed using the POLY constructor.

10.2.69 PLN3PTS

```
PlaneType PLN3PTS( PointType Pt1, PointType Pt2, PointType Pt3 )
```

Computes a plane out of three points.
Example:

```
Pl1 = PLN3PTS( point( 0, 0, 0 ), point( 0, 1, 0 ), point( 1, 0, 0 ) );
```

10.2.70 POLY

```
PolygonType POLY( ListType VrtxList, NumericType IsPolyline )
```

Creates a single polygon/polyline (and therefore open) object, defined by the vertices in **VrtxList** (see LIST). All elements in **VrtxList** must be of VectorType type. If **IsPolyline**, a polyline is created, otherwise a polygon.
Example:

```
V1  = vector( 0.0, 0.0, 0.0 );
V2  = vector( 0.3, 0.0, 0.0 );
V3  = vector( 0.3, 0.0, 0.1 );
V4  = vector( 0.2, 0.0, 0.1 );
V5  = vector( 0.2, 0.0, 0.5 );
V6  = vector( 0.3, 0.0, 0.5 );
V7  = vector( 0.3, 0.0, 0.6 );
V8  = vector( 0.0, 0.0, 0.6 );
V9  = vector( 0.0, 0.0, 0.5 );
V10 = vector( 0.1, 0.0, 0.5 );
V11 = vector( 0.1, 0.0, 0.1 );
V12 = vector( 0.0, 0.0, 0.1 );
I = POLY( list( V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12 ),
          FALSE );
```

constructs an object with a single polygon in the shape of the letter I. See Figure 32.

10.2.71 PRISA

```
ListType PRISA( SurfaceType Srfs, NumericType SamplesPerCurve,
                NumericType Epsilon, ConstantType Dir, VectorType Space )
```

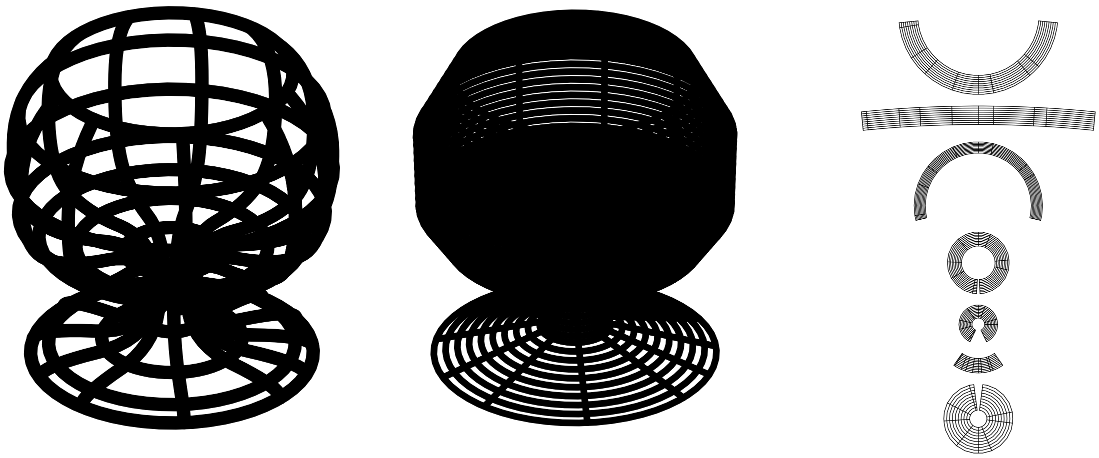


Figure 33: The layout (prisa in hebrew...) of a freeform surface can be approximated using the PRISA function.

Computes a layout (prisa) of the given surface(s) **Srfs**, and returns a list of surface objects representing the layout. The surface is approximated to within **Epsilon** in direction **Dir** into a set of ruled surfaces and then developable surfaces that are laid out flat onto the XY plane. If **Epsilon** is negative, the piecewise ruled surface approximation in 3-space is returned. **SamplesPerCurve** controls the piecewise linear approximation of the boundary of the ruled/developable surfaces. **Space** is a vector whose X component controls the space between the different surfaces' layout, and whose Y component controls the space between different layout pieces.

Example:

```
cross = cbspline( 3,
                  list( ctlpt( E3, 0.7, 0.0, 0. ),
                        ctlpt( E3, 0.7, 0.0, 0.06 ),
                        ctlpt( E3, 0.1, 0.0, 0.1 ),
                        ctlpt( E3, 0.1, 0.0, 0.6 ),
                        ctlpt( E3, 0.6, 0.0, 0.6 ),
                        ctlpt( E3, 0.8, 0.0, 0.8 ),
                        ctlpt( E3, 0.8, 0.0, 1.4 ),
                        ctlpt( E3, 0.6, 0.0, 1.6 ) ),
                  list( KV_OPEN ) );
wglass = surfrev( cross );
wgl_ruled = PRISA( wglass, 6, -0.1, COL, vector( 0, 0.25, 0.0 ) );
wgl_prisa = PRISA( wglass, 6, 0.1, COL, vector( 0, 0.25, 0.0 ) );
```

Computes a layout of a wine glass in **wgl_prisa** and a three-dimensional ruled surface approximation of wglass in **wgl_ruled**. See Figure 33.

10.2.72 PT3BARY

```
VectorType PT3BARY( PointType Pt1, PointType Pt2, PointType Pt3,
                    PointType InteriorPt )
```

Computes the barycentric coordinates of **InterPt** with respect to the triangle defined by **Pt1**, **Pt2**, **Pt3**. Returned is a vector of three coefficents, which are the weights of the three points of the triangle. **InteriorPt** is assumed to be in the triangle.

Example:

```
Coeffs = PT3BARY( point( 0, 0, 0 ),
                  point( 1, 0, 0 ),
                  point( 0, 1, 0 ),
                  point( 0.25, 0.25, 0.0 ) );
```

10.2.73 PTLNPLN

```
VectorType PTLNPLN( PointType LineOrig, VectorType LineRay, PlaneType Plane )
```

Computes the point of intersection of given line **LineOrig**, **LineRay** with plane **Plane**.
Example:

```
InterPt = PtLnPln( point( 1, 0, 1 ), vector( 1, 1, 1 ), Plane( 0, 0, 1, 0 ) );
```

10.2.74 PTPTLN

```
VectorType PTPTLN( PointType Point, PointType LineOrig, VectorType LineRay )
```

Computes the point on line **LineOrig**, **LineRay** that is closest to point **Point**. See also DSTPTLN
Example:

```
ClosestPt = PTPTLN( point( 0, 0, 0 ), point( 1, 1, 0 ), vector( 1, 1, 1 ) );
```

10.2.75 PTSLNLN

```
VectorType PTSLNLN( PointType Line1Orig, VectorType Line1Ray,
                    PointType Line2Orig, VectorType Line2Ray )
```

Computes the closest two points on the two lines defined by point **LineiOrig** and ray **LineiRay**. See also DSTLNLN. Returned is a list object with the two points.
Example:

```
ClosestPts = PtsLnLn( point( 1, 0, 0 ), vector( 0, 1, 0 ),
                      point( 0, 1, 0 ), vector( 1, 0, 0 ) );
```

10.2.76 RULEDSRF

```
SurfaceType RULEDSRF( CurveType Crv1, CurveType Crv2 )
```

Constructs a ruled surface between the two curves **Crv1** and **Crv2**. The curves do not have to have the same order or type, and will be promoted to their least common denominator.
Example:

```
c1 = cbspline( 3,
               list( ctlpt(E3, 1.7, 0.0 , 0  ),
                     ctlpt(E3, 0.7, 0.7 , 0  ),
                     ctlpt(E3, 1.7, 0.3 , 0  ),
                     ctlpt(E3, 1.5, 0.8 , 0  ),
                     ctlpt(E3, 1.6, 1.0 , 0  ) ),
          list( KV_OPEN ) );
c2 = cbspline( 3,
               list( ctlpt(E3, 0.7, 0.0 , 0  ),
```

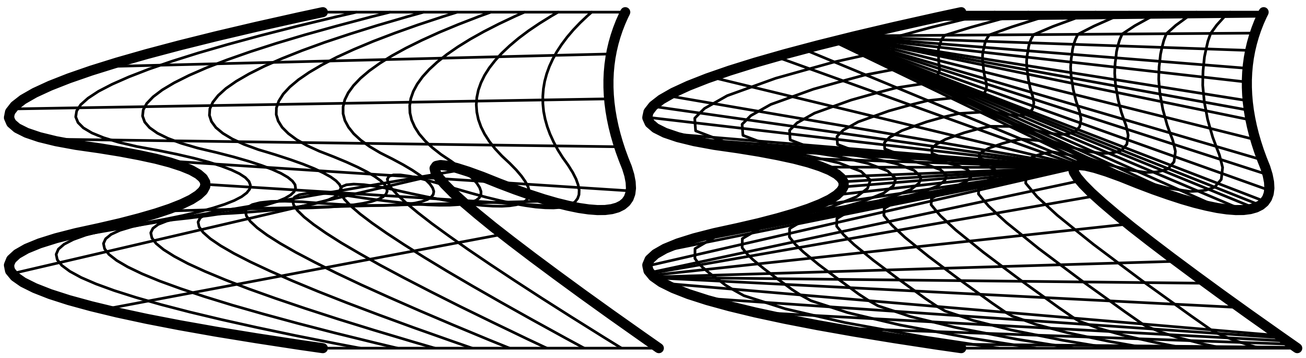


Figure 34: A naive construction of a ruled surface (left) using RULEDSRF result with self intersection. FFMATCH is employed (right) to automatically resolve this self intersection.

```
                      ctlpt(E3,-0.7, 0.2 , 0  ),
                      ctlpt(E3, 0.7, 0.5 , 0  ),
                      ctlpt(E3,-0.7, 0.7 , 0  ),
                      ctlpt(E3, 0.7, 1.0 , 0  ) ) ,
           list( KV_OPEN ) );

    srf1 = RULEDSRF( c1, c2 );
    interact( list( c1, c2, srf1 ), on );

    c2a = ffmatch( c1, c2, 50, 100, 2, false, 1 );
    srf2 = RULEDSRF( c1, c2a );
    interact( list( c1, c2, srf2 ), on );
```

Constructs a planar ruled surface between two curves, **c1** and **c2**. The naive construction causes self intersection, but by employing FFMATCH the self intersection can be resloved. See Figure 34.

See also FFMATCH.

10.2.77 SBEZIER

```
SurfaceType SBEZIER( ListType CtlMesh )
```

Creates a Bezier surface using the provided control mesh. **CtlMesh** is a list of rows, each of which is a list of control points. All control points must be of type (E1-E5, P1-P5), or regular PointType defining the surface's control polygon. Surface's point type will be of a space which is the union of the spaces of all points.

The created surface is the piecewise polynomial (or rational) surface,

$$S(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} P_{ij} B_i(u) B_j(v) \tag{11}$$

where P_{ij} are the control points **CtlMesh**. and m and n are the degrees of the surface, which are one less than the number of points in the appropriate direction.

Example:

```
  Srf = SBEZIER( list ( list( ctlpt( E3, 0.0, 0.0, 1.0 ),
                              ctlpt( E3, 0.0, 1.0, 0.0 ),
                              ctlpt( E3, 0.0, 2.0, 1.0 ) ),
```

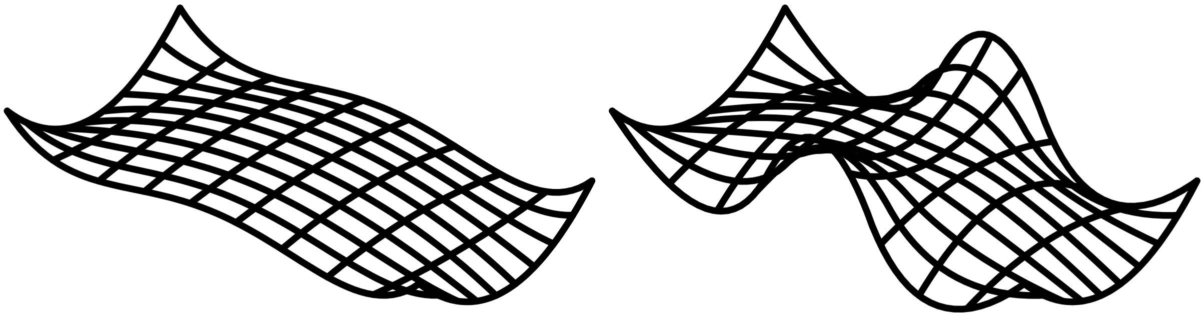


Figure 35: A bezier surface (left) of degree 3 by 5 and a Bspline surface (right) of degree 3 by 3 (bi-quadratic). Both share the same control mesh.

```
list( ctlpt( E3, 1.0, 0.0, 0.0 ),
      ctlpt( E3, 1.0, 1.0, 2.0 ),
      ctlpt( E3, 1.0, 2.0, 0.0 ) ),
list( ctlpt( E3, 2.0, 0.0, 2.0 ),
      ctlpt( E3, 2.0, 1.0, 0.0 ),
      ctlpt( E3, 2.0, 2.0, 2.0 ) ),
list( ctlpt( E3, 3.0, 0.0, 0.0 ),
      ctlpt( E3, 3.0, 1.0, 2.0 ),
      ctlpt( E3, 3.0, 2.0, 0.0 ) ),
list( ctlpt( E3, 4.0, 0.0, 1.0 ),
      ctlpt( E3, 4.0, 1.0, 0.0 ),
      ctlpt( E3, 4.0, 2.0, 1.0 ) ) ) );
```

See Figure 35.

10.2.78 SBSPLINE

```
SurfaceType SBSPLINE( NumericType UOrder, NumericType VOrder,
                      ListType CtlMesh, ListType KnotVectors )
```

Creates a Bspline surface from the provided **UOrder** and **VOrder** orders, the control mesh **CtlMesh**, and the two knot vectors **KnotVectors**. **CtlMesh** is a list of rows, each of which is a list of control points. All control points must be of point type (E1-E5, P1-P5), or regular PointType defining the surface's control mesh. Surface's point type will be of a space which is the union of the spaces of all points. **KnotVectors** is a list of two knot vectors. Each knot vector is a list of NumericType knots of length **#CtlPtList** plus the **Order**. If, however, the length of the knot vector is equal to **#CtlPtList + Order + Order - 1** the curve is assumed *periodic.* The knot vector may also be a list of a single constant KV_OPEN or KV_FLOAT or KV_PERIODIC, in which a uniform knot vector with the appropriate length and with open, floating or periodic end condition will be constructed automatically.

The created surface is the piecewise polynomial (or rational) surface,

$$S(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{ij} B_{i,\chi}(u) B_{j,\xi}(v) \tag{12}$$

where P_{ij} are the control points **CtlMesh**, and m and n are the degrees of the surface, which are one less than **UOrder** and **VOrder**. χ and ξ are the two knot vectors of the surface.

Example:

```
Mesh = list ( list( ctlpt( E3, 0.0, 0.0, 1.0 ),
                    ctlpt( E3, 0.0, 1.0, 0.0 ),
                    ctlpt( E3, 0.0, 2.0, 1.0 ) ),
              list( ctlpt( E3, 1.0, 0.0, 0.0 ),
                    ctlpt( E3, 1.0, 1.0, 2.0 ),
                    ctlpt( E3, 1.0, 2.0, 0.0 ) ),
              list( ctlpt( E3, 2.0, 0.0, 2.0 ),
                    ctlpt( E3, 2.0, 1.0, 0.0 ),
                    ctlpt( E3, 2.0, 2.0, 2.0 ) ),
              list( ctlpt( E3, 3.0, 0.0, 0.0 ),
                    ctlpt( E3, 3.0, 1.0, 2.0 ),
                    ctlpt( E3, 3.0, 2.0, 0.0 ) ),
              list( ctlpt( E3, 4.0, 0.0, 1.0 ),
                    ctlpt( E3, 4.0, 1.0, 0.0 ),
                    ctlpt( E3, 4.0, 2.0, 1.0 ) ) );
Srf = SBSPLINE( 3, 3, Mesh, list( list( KV_OPEN ),
                                  list( 3, 3, 3, 4, 5, 6, 6, 6 ) ) );
```

constructs a bi-quadratic Bspline surface with its first knot vector having uniform knot spacing with open end conditions. See Figure 35.

10.2.79 SCRVTR

`SurfaceType SCRVTR( SurfaceType Srf, ConstType PtType, ConstType Dir )`

Symbolically computes the extreme curvature bound on **Srf**. If **Dir** is either ROW or COL, then the normal curvature square of **Srf** in **Dir** is computed symbolically and returned. Otherwise, a upper bound on the sum of the squares of the two principal curvatures is symbolically computed and returned.

Returned value is a surface that can be evaluated to the curvature bound, given a UV location. The returned surface value is a scalar field of point type P1 (scalar rational). However, if **PtType** is one of E1, P1, E3, P3 the returned surface is coerced to this given type. If the types are one of E3, P3, then the Y and Z axes are set to be equivalent to the U and V parametric domains.

This function computes the square of the normal curvature scalar field for surfaces as (in the U parametric direction, same for V),

$$\kappa_n^u(u,v) = \frac{\left\langle n, \frac{\partial^2 S}{\partial u^2} \right\rangle}{\left\langle \frac{\partial S}{\partial u}, \frac{\partial S}{\partial u} \right\rangle} \tag{13}$$

and computes $\xi(u,v) = k_1(u,v)^2 + k_2(u,v)^2$ as the scalar field of

$$\xi(u,v) = \frac{(g_{11}l_{22} + l_{11}g_{22} - 2g_{12}l_{12})^2 - 2\,|G|\,|L|}{|G|^2\,\|n\|^2}, \tag{14}$$

where g_{ij} and l_{ij} are the coefficients of the first and second fundamental forms G and L.

See also CCRVTR.

Example:

```
cross = cbspline( 3,
                  list( ctlpt( E2,  0.0,  0.0 ),
                        ctlpt( E2,  0.8,  0.0 ),
                        ctlpt( E2,  0.8,  0.2 ),
                        ctlpt( E2,  0.07, 1.4 ),
                        ctlpt( E2, -0.07, 1.4 ),
```

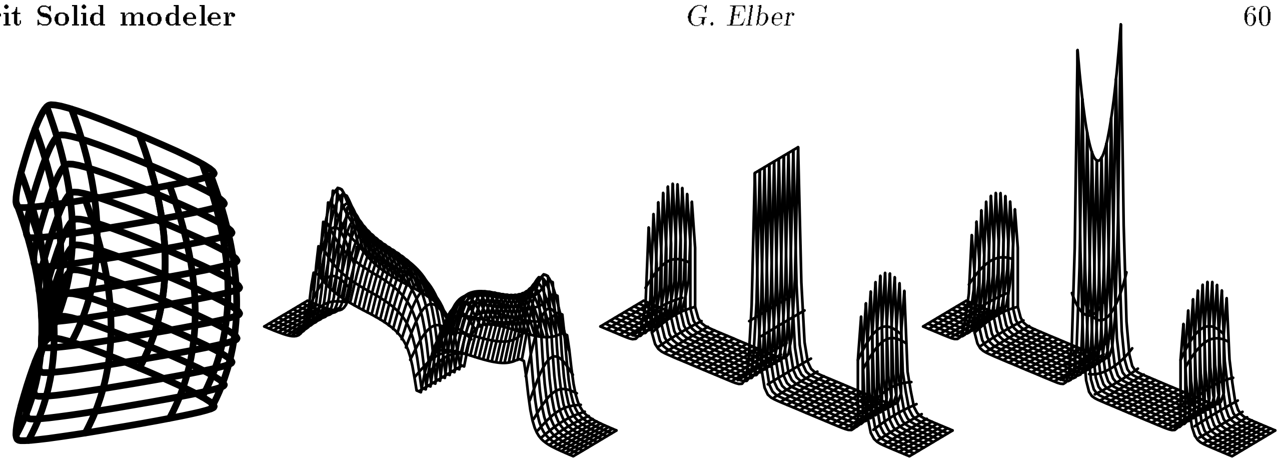


Figure 36: From left to right: original surface, normal curvature in the U direction, normal curvature in the V direction, sum of square of principle curvatures (different scales). All computed using SCRVTR.

```
                        ctlpt( E2, -0.8,  0.2 ),
                        ctlpt( E2, -0.8,  0.0 ),
                        ctlpt( E2,  0.0,  0.0 ) ),
                  list( KV_OPEN ) );
cross = coerce( cross, e3 );
s = sFromCrvs( list( cross,
                     cross * trans( vector( 0.5, 0, 1 ) ),
                     cross * trans( vector( 0, 0, 2 ) ) ), 3 );
view( list( s, axes ), on );

UCrvtrZXY = scrvtr( s, E3, row );
VCrvtrZXY = scrvtr( s, E3, col );
UCrvtrXYZ = UCrvtrZXY * rotx( -90 ) * roty( -90 ) * scale( vector( 1, 1, 0.001 ) );
VCrvtrXYZ = VCrvtrZXY * rotx( -90 ) * roty( -90 ) * scale( vector( 1, 1, 10 ) );
color( UCrvtrXYZ, red );
color( VCrvtrXYZ, magenta );

view( list( UCrvtrXYZ, VCrvtrXYZ ), off );

CrvtrZXY = scrvtr( s, E3, off );
CrvtrXYZ = CrvtrZXY * rotx( -90 ) * roty( -90 ) * scale( vector( 1, 1, 0.001 ) );
color( CrvtrXYZ, green );

view( CrvtrXYZ, off );
```

Computes the sqaure of the normal curvature in the U and V direction, flips its scalar value from X to Z using rotations and scale the fields to reasonable values and display them. Then, display a total bound on the normal curvature as well.

Due to the large degree of the resulting fields be warned that rational surfaces will compute into large degree curvature bound fields. See also IRITSTATE("InterpProd", FALSE); for faster symbolic computation. See Figure 36.

10.2.80 SDERIVE

```
SurfaceType SDERIVE( SurfaceType Srf, NumericType Dir )
```

Returns a vector field surface representing the differentiated surface in the given direction (ROW or COL). Evaluation of the returned surface at a given parameter value will return a vector *tangent* to **Srf** in **Dir** at that parameter value.

```
DuSrf = SDERIVE( Srf, ROW );
DvSrf = SDERIVE( Srf, COL );
Normal = coerce( seval( DuSrf, 0.5, 0.5 ), VECTOR_TYPE ) ^
         coerce( seval( DvSrf, 0.5, 0.5 ), VECTOR_TYPE );
```

computes the two partial derivatives of the surface **Srf** and computes its normal as their cross product, at the parametric location (0.5, 0.5).

10.2.81 SDIVIDE

```
SurfaceType SDIVIDE( SurfaceType Srf, ConstantType Direction,
                                                        NumericType Param )
```

or

```
TrimSrfType SDIVIDE( TrimSrfType Srf, ConstantType Direction,
                                                        NumericType Param )
```

Subdivides a (possibly trimmed) surface into two at the specified parameter value **Param** in the specified **Direction** (ROW or COL). **Srf** can be either a Bspline surface in which **Param** must be conatined in the parametric domain of the surface, or a Bezier surface in which **Param** must be in the range of zero to one.

It returns a list of upto two sub-surfaces. The individual surfaces may be extracted from the list using the **NTH** command. If **Srf** is a trimmed surface, it can be the case that one of the two subdivided surfaces is completely trimmed out, and hence only one surface will be returned.

Example:

```
SrfLst = SDIVIDE( Srf, ROW, 0.5 );
Srf1 = nth( SrfLst, 1 );
Srf2 = nth( SrfLst, 2 );
```

Subdivides **Srf** at the parameter value of 0.5 in the ROW direction. See Figure 37.

10.2.82 SEDITPT

```
SurfaceType SEDITPT( SurfaceType Srf, CtlPtType CPt, NumericType UIndex,
                                                     NumericType VIndex )
```

Provides a simple mechanism to manually modify a single control point number **UIndex** and **VIndex** (base count is 0) in the control mesh of **Srf** by substituting **CtlPt** instead. CtlPt must have the same point type as the control points of **Srf**. Original surface **Srf** is not modified.

Example:

```
CPt = ctlpt( E3, 1, 2, 3 );
NewSrf = SEDITPT( Srf, CPt, 0, 0 );
```

Constructs a **NewSrf** with the first control point of **Srf** being **CPt**.

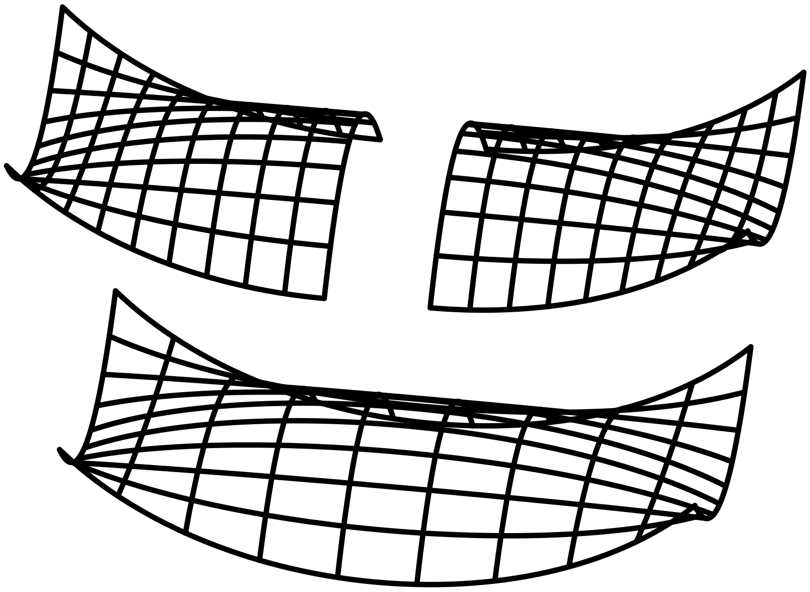


Figure 37: A surface can be subdivided into two distinct regions using SDIVIDE.

10.2.83 SEVAL

```
CtlPtType SEVAL( SurfaceType Srf, NumericType UParam, NumericType VParam )
```

or

```
CtlPtType SEVAL( TrimSrfType Srf, NumericType UParam, NumericType VParam )
```

Evaluates the provided (possibly trimmed) surface **Srf** at the given **UParam** and **VParam** parameters. Both **UParam** and **VParam** should be contained in the surface parametric domain if **Srf** is a Bspline surface, or between zero and one if **Srf** is a Bezier surface. The returned control point has the same type as the control points of **Srf**.

Example:

```
CPt = SEVAL( Srf, 0.25, 0.22 );
```

Evaluates **Srf** at the parameter values of (0.25, 0.22).

10.2.84 SFOCAL

```
SurfaceType SFOCAL( SurfaceType Srf, NumericType Dir )
```

Evaluates the focal surface field of surface **Srf** using the normal curvature in the isoparametric direction as given by **Dir** (either ROW or COL). Note this function is not using the principal curvatures as is generaly the case for focal surfaces.

Example:

```
gcross = cbspline( 3,
                   list( ctlpt( E3, 0.3, 0.0, 0.0 ),
                         ctlpt( E3, 0.1, 0.0, 0.1 ),
                         ctlpt( E3, 0.1, 0.0, 0.4 ),
                         ctlpt( E3, 0.5, 0.0, 0.5 ),
                         ctlpt( E3, 0.6, 0.0, 0.8 ) ),
                   list( KV_OPEN ) );
```

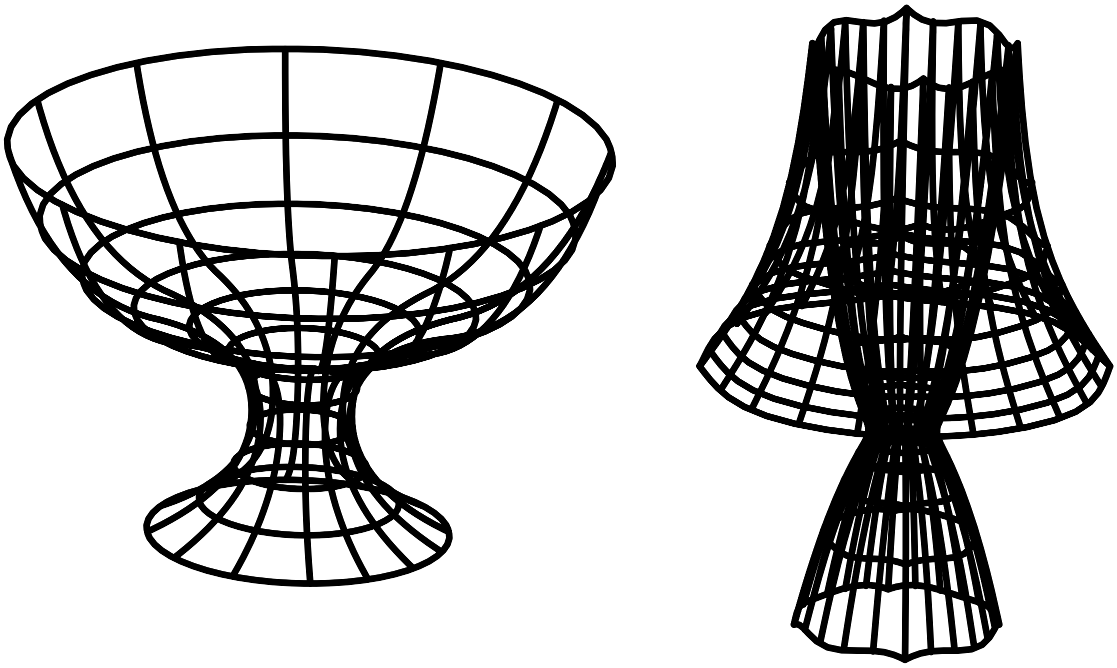


Figure 38: A focal surface (right) of a glass surface (left) can be computed using SFOCAL.

```
glass = surfprev( gcross );
color( glass, red );

gfocal = SFOCAL(glass, col);
```

Evaluates the focal surface using the COL isoparametric direction's normal curvature of the glass surface. See Figure 38.

10.2.85 SFROMCRVS

```
SurfaceType SFROMCRVS( ListType CrvList, NumericType OtherOrder )
```

Constructs a surface by substituting the curves in **CrvList** as rows in a control mesh of a surface. Curves in **CrvList** are made compatible by promoting Bezier curves to Bsplines if necessary, and raising degree and refining as required before substituting the control polygons of the curves as rows in the mesh. The other direction order is set by **OtherOrder**, which cannot be larger than the number of curves.

The surface interpolates the first and last curves only.

Example:

```
Crv1 = cbspline( 3,
                 list( ctlpt( E3, 0.0, 0.0, 0.0 ),
                       ctlpt( E3, 1.0, 0.0, 0.0 ),
                       ctlpt( E3, 1.0, 1.0, 0.0 ) ),
                 list( KV_OPEN ) );
Crv2 = Crv1 * trans( vector( 0.0, 0.0, 1.0 ) );
Crv3 = Crv2 * trans( vector( 0.0, 1.0, 0.0 ) );
Srf = SFROMCRVS( list( Crv1, Crv2, Crv3 ), 3 );
```

See Figure 39.

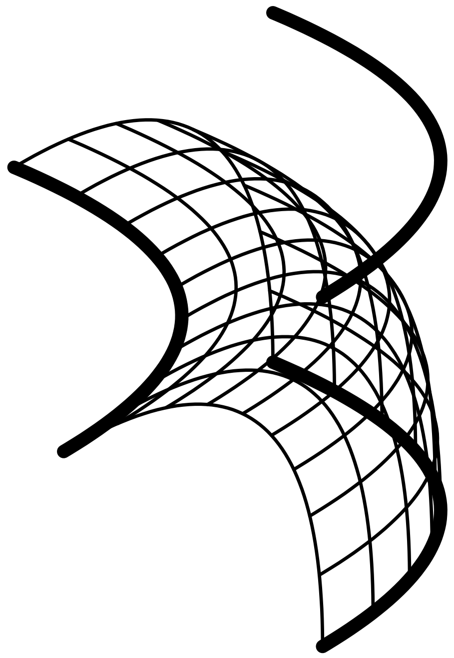


Figure 39: A surface can be constructed from a list of curves substituted as rows into its mesh using SFROMCRVS. The surface does not necessarily interpolate the curves.

10.2.86 SGAUSS

```
SurfaceType SGAUSS( SurfaceType Srf )
```

Evaluates the Gaussian curvature field of surface **Srf**.
Example:

```
Srf1 = hermite( cbezier( list( ctlpt( E3, 0.0, 0.0, 0.0 ),
                               ctlpt( E3, 0.5, 0.2, 0.0 ),
                               ctlpt( E3, 1.0, 0.0, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 1.0, 0.0 ),
                               ctlpt( E3, 0.5, 0.8, 0.0 ),
                               ctlpt( E3, 1.0, 1.0, 0.5 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ) ) ) );

SGauss = SGAUSS( Srf1 );
```

Evaluates the Gaussian curvaure of **Srf1**. See Figure 40.

10.2.87 SINTERP

```
SurfaceType SINTERP( ListType PtList, NumericType UOrder, NumericType VOrder,
                     NumericType USize, NumericType VSize,
                     ConstantType Param)
```

Computes a Bspline polynomial surface that interpolates or approximates the rectangular grid of points in **PtList**. The Bspline surface will have orders **UOrder** and **VOrder** and mesh of size

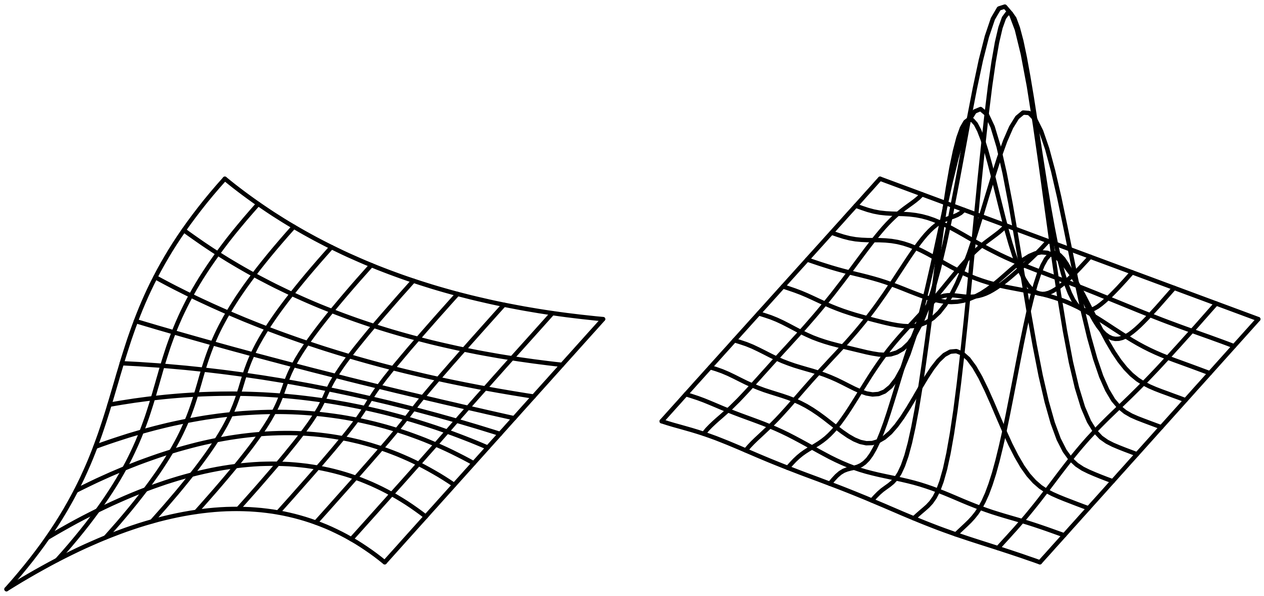


Figure 40: The Gaussian curvature field (right) of the quadratic by cubic surface (left) is computed using SGAUSS. The Gaussian curvature field is scaled down to %1 to fit into the figure. Compare with figure 42.

USize by **VSize** control points. The knots will be spaced according to **Param** which can be one of PARAM_UNIFORM, PARAM_CHORD or PARAM_CENTRIP. The former prescribed a uniform knot sequence and the latters specified a knot spacing accoridng to the chord length and a square root of the chord length. Currently only PARAM_UNIFORM is supported. **PtList** is a list of list of points where all lists should carry the same amount of points in them, defining a rectangular grid. All points in **PtList** must be of type (E1-E5, P1-P5) control point, or regular PointType. If **USize** and **VSize** are equal to the number of points in the grid **PtList** the resulting curve will *interpolate* the data set. Otherwise, if **USize** or **VSize** is less than the number of points in **PtList** the point data set will be least square approximated. In no time can **USize** or **VSize** be larger that the number of points in **PtList** or lower than **UOrder** and **VOrder**, respectively. If **USize** or **VSize** are zero, the grid size is used, forcing an interpolation of the data set.

All interior knots will be distinct preserving maximal continuity. The resulting Bspline surface will have open end conditions.

Example:

```
pl = nil();
pll = nil();
for ( x = -5, 1, 5,
      pl = nil():
      for ( y = -5, 1, 5,
            snoc( point( x, y, sin( x * Pi / 2 ) * cos( y * Pi / 2 ) ),
                  pl )
      ):
      snoc( pl, pll ) );

s1 = sinterp( pll, 3, 3, 8, 8, PARAM_UNIFORM );
s2 = sinterp( pll, 3, 3, 11, 11, PARAM_UNIFORM );
```

Samples an explicit surface sin(x) * cos(y) at a grid of 11 by 11 points, least square fit with a grid of size of 8 by 8 surface **s1**, and interpolate surface **s2** using this data set. See Figure 41.

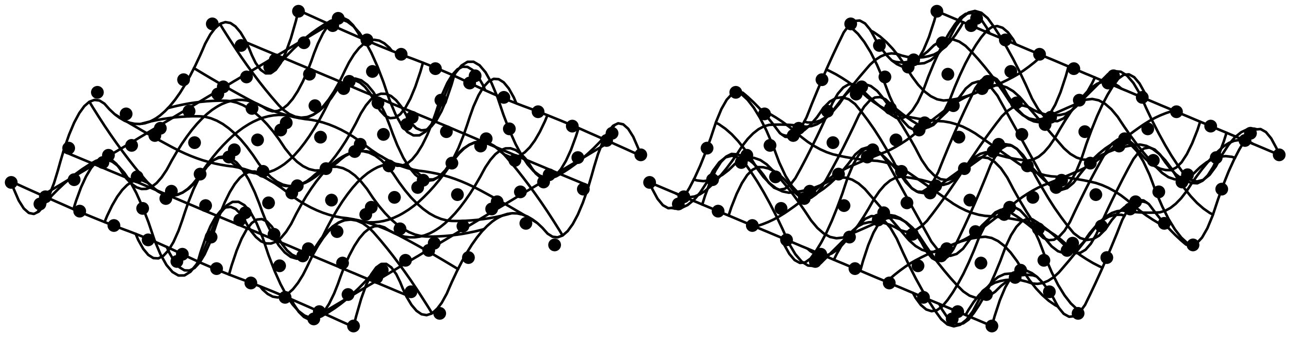


Figure 41: A surface least square fitting a data set with insufficient degrees of freedom (left) and actually interpolating the data set (right), all using SINTERP.

10.2.88 SMEANSQR

```
SurfaceType SMEANSQR( SurfaceType Srf )
```

Evaluates the square of the mean curvature field of surface **Srf**.
Example:

```
Srf1 = hermite( cbezier( list( ctlpt( E3, 0.0, 0.0, 0.0 ),
                               ctlpt( E3, 0.5, 0.2, 0.0 ),
                               ctlpt( E3, 1.0, 0.0, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 1.0, 0.0 ),
                               ctlpt( E3, 0.5, 0.8, 0.0 ),
                               ctlpt( E3, 1.0, 1.0, 0.5 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ),
                               ctlpt( E3, 0.0, 2.0, 0.0 ) ) ) );

SMean = SMEANSQR( Srf1 );
```

Evaluates the square of the mean curvaure of **Srf1**. See Figure 42.

10.2.89 SMERGE

```
SurfaceType SMERGE( SurfaceType Srf1, SurfaceType Srf2,
                    NumericType Dir, NumericType SameEdge )
```

Merges two surfaces along the requested direction (ROW or COL). If SameEdge is non-zero (ON or TRUE), then the common edge is assumed to be identical and copied only once. Otherwise (OFF or FALSE), a ruled surface is constructed between the two surfaces along the (not) common edge.
Example:

```
MergedSrf = SMERGE( Srf1, Srf2, ROW, TRUE );
```

10.2.90 SMORPH

```
SurfaceType SMORPH( SurfaceType Srf1, SurfaceType Srf2, NumericType Blend )
```

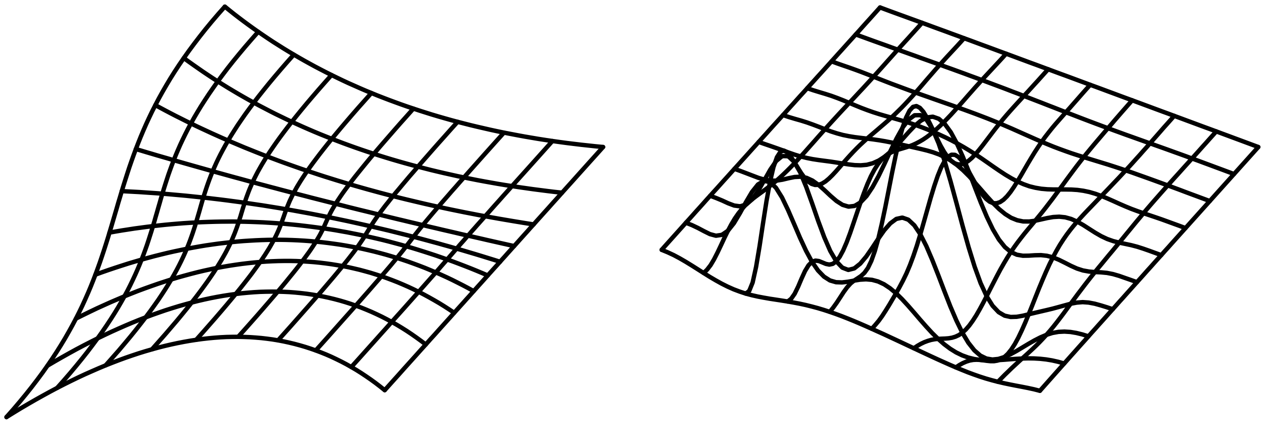


Figure 42: The square of the mean curvature field (right) of the quadratic by cubic surface (left) is computed using SMEANSQR. The square of the mean curvature field is scaled down to %1 to fit into the figure. Compare with figure 40.

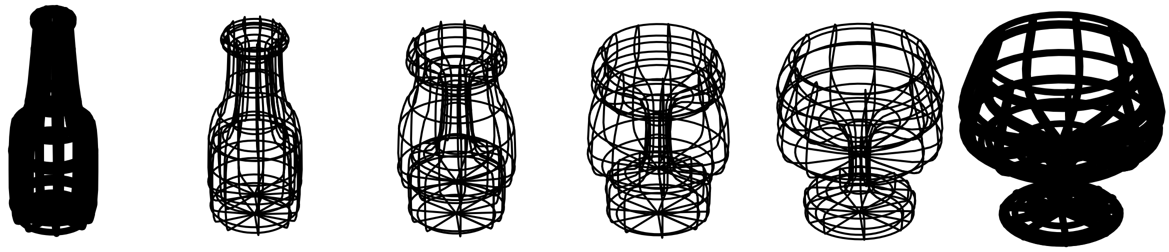


Figure 43: A morphing sequence between a bottle and a glass. Snapshots computed using SMORPH.

Creates a new surface which is a *convex blend* of the two given surfaces. The two given surfaces must be compatible (see FFCOMPAT) before this blend is invoked. Very useful if a sequence that "morphs" one surface to another is to be created.

Example:

```
for ( i = 0.0, 1.0, 11.0,
       Msrf = SMORPH( Srf1, Srf2, i / 11.0 ):
       color( Msrf, white ):
       attrib( Msrf, "rgb", "255,255,255" ):
       attrib( Msrf, "reflect", "0.7" ):
       save( "morp1-" + i, Msrf )
);
```

creates a sequence of 12 surfaces, morphed from **Srf1** to **Srf2** and saves them in the files "morph-0.dat" to "morph-11.dat". See also CMORPH. See Figure 43.

10.2.91 SNORMAL

```
VectorType SNORMAL( SurfaceType Srf, NumericType UParam, NumericType VParam )
```

or

```
VectorType SNORMAL( TrimSrfType Srf, NumericType UParam, NumericType VParam )
```

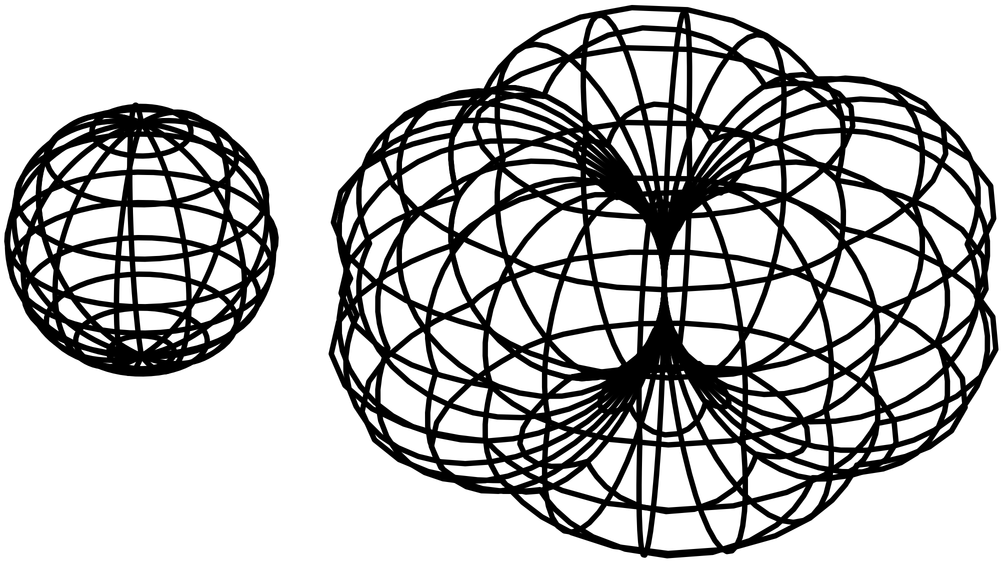


Figure 44: A vector field normal (right) computed for a unit sphere using SNRMLSRF. The normal field degenerates at the norph and south poles because the surface is not regular there.

Computes the normal vector to (possibly trimmed) surface **Srf** at the parameter values **UParam** and **VParam**. The returned vector has a unit length.

Example:

```
Normal = SNORMAL( Srf, 0.5, 0.5 );
```

computes the normal to **Srf** at the parameter values (0.5, 0.5). See also SNRMLSRF.

10.2.92 SNRMLSRF

```
SurfaceType SNRMLSRF( SurfaceType Srf )
```

Symbolically computes a vector field surface representing the non-normalized normals of the given surface. That is the normal surface, evaluated at (u, v), provides a vector in the direction of the normal of the original surface at (u, v). The normal surface is computed as the symbolic cross product of the two surfaces representing the partial derivatives of the original surface.

Example:

```
NrmlSrf = SNRMLSRF( Srf );
```

See Figure 44.

10.2.93 SPHERE

```
PolygonType SPHERE( VectorType Center, NumericType Radius )
```

Creates a SPHERE geometric object, defined by **Center** as the center of the SPHERE, and with **Radius** as the radius of the SPHERE. See RESOLUTION for accuracy of SPHERE approximation as a polygonal model.

10.2.94 SRAISE

```
SurfaceType SRAISE( SurfaceType Srf, ConstantType Direction,
                                                     NumericType NewOrder )
```

Raises **Srf** to the specified **NewOrder** in the specified **Direction**.
Example:

```
Srf = ruledSrf( cbezier( list( ctlpt( E3, -0.5, -0.5, 0.0 ),
                               ctlpt( E3,  0.5, -0.5, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, -0.5,  0.5, 0.0 ),
                               ctlpt( E3,  0.5,  0.5, 0.0 ) ) ) );
Srf = SRAISE( SRAISE( Srf, ROW, 3 ), COL, 3 );
```

constructs a bilinear flat ruled surface and raises both its directions to be a bi-quadratic surface.

10.2.95 SREFINE

```
SurfaceType SREFINE( SurfaceType Srf, ConstantType Direction,
                     NumericType Replace, ListType KnotList )
```

Provides the ability to **Replace** a knot vector of **Srf** or refine it in the specified direction **Direction** (ROW or COL). **KnotList** is a list of knots to refine **Srf** at. All knots should be contained in the parametric domain of **Srf** in **Direction**. If the knot vector is replaced, the length of **KnotList** should be identical to the length of the original knot vector of **Srf** in **Direction**. If **Srf** is a Bezier surface, it is automatically promoted to be a Bspline surface.
Example:

```
Srf = SREFINE( SREFINE( Srf,
                        ROW, FALSE, list( 0.333, 0.667 ) ),
               COL, FALSE, list( 0.333, 0.667 ) );
```

refines **Srf** in both directions by adding two more knots at 0.333 and 0.667.

10.2.96 SREGION

```
SurfaceType SREGION( SurfaceType Srf, ConstantType Direction,
                                 NumericType MinParam, NumericType MaxParam )
```

Extracts a region of **Srf** between **MinParam** and **MaxParam** in the specified **Direction**. Both **MinParam** and **MaxParam** should be contained in the parametric domain of **Srf** in **Direction**.
Example:

```
Srf = ruledSrf( cbezier( list( ctlpt( E3, -0.5, -0.5, 0.5 ),
                               ctlpt( E3,  0.0,  0.5, 0.0 ),
                               ctlpt( E3,  0.5, -0.5, 0.0 ) ) ),
                cbezier( list( ctlpt( E3, -0.5,  0.5, 0.0 ),
                               ctlpt( E3,  0.0,  0.0, 0.0 ),
                               ctlpt( E3,  0.5,  0.5, 0.5 ) ) ) );
SubSrf = SREGION( Srf, ROW, 0.3, 0.6 );
```

extracts the region of **Srf** from the parameter value 0.3 to the parameter value 0.6 along the ROW direction. the COLumn direction is extracted as a whole. See Figure 45.

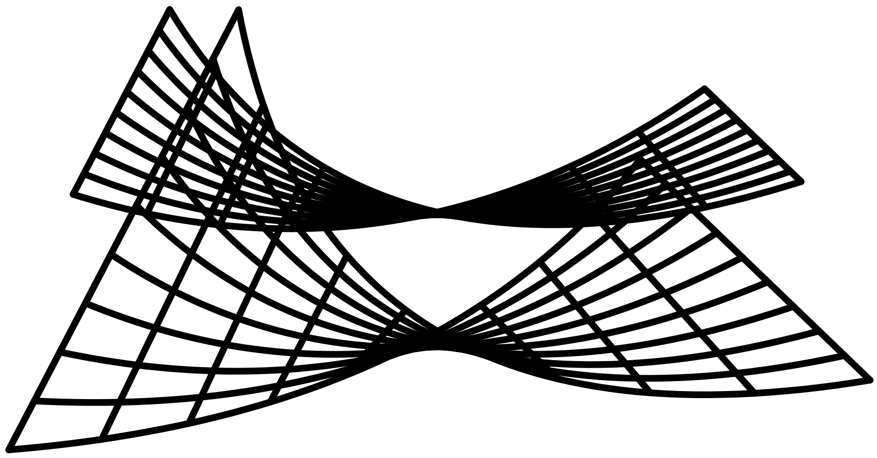


Figure 45: A region can be extracted from a freeform surface using SREGION.

10.2.97 SREPARAM

```
SurfaceType SREPARAM( SurfaceType Srf, ConstantType Direction,
                                 NumericType MinParam, NumericType MaxParam )
```

Reparametrize **Srf** over a new domain from **MinParam** to **MaxParam**, in the prescribed **Direction**. This operation does not affect the geometry of the surface and only affine transforms its knot vectors. A Bezier surface will automatically be promoted into a Bspline surface by this function.

Example:

```
srf = sbspline( 2, 4,
                list( list( ctlpt( E3, 0.0, 0.0, 1.0 ),
                            ctlpt( E2, 0.0, 1.0 ),
                            ctlpt( E3, 0.0, 2.0, 1.0 ) ),
                      list( ctlpt( E2, 1.0, 0.0 ),
                            ctlpt( E3, 1.0, 1.0, 2.0 ),
                            ctlpt( E2, 1.0, 2.0 ) ),
                      list( ctlpt( E3, 2.0, 0.0, 2.0 ),
                            ctlpt( E2, 2.0, 1.0 ),
                            ctlpt( E3, 2.0, 2.0, 2.0 ) ),
                      list( ctlpt( E2, 3.0, 0.0 ),
                            ctlpt( E3, 3.0, 1.0, 2.0 ),
                            ctlpt( E2, 3.0, 2.0 ) ),
                      list( ctlpt( E3, 4.0, 0.0, 1.0 ),
                            ctlpt( E2, 4.0, 1.0 ),
                            ctlpt( E3, 4.0, 2.0, 1.0 ) ) ),
                list( list( KV_OPEN ),
                      list( KV_OPEN ) ) );

srf = sreparam( sreparam( srf, ROW, 0, 1 ), COL, 0, 1 );
```

Ensures that the Bspline surface is defined over the unit size parametric domain.

10.2.98 SRINTER

```
PointType SRINTER( SurfaceType Srf, PointType RayOrigin,
                   VectorType RayDirection )
```

Computes the first intersection, if any, of the prescribed ray originating from **RayOrigin** in direction **RayDirection** with surface **Srf**. Returns the intersection point in the parametric space of **Srf** with the U and V coordinates as the X and Y coefficients of the returned value. The intersection is computed between the ray and a polygonal approximation of the surface **Srf** as set via the RESOLUTION variable.

Example:

```
RayOrigin = point( 2, 0.1, 0.3 );
RayDir = vector( -4, 0, 0 );

RayLine = coerce( RayOrigin, E3 ) + coerce( RayOrigin + RayDir, E3 );
color( RayLine, magenta );
attrib( RayLine, "dwidth", 2 );

resolution = 5;
InterPt = SRINTER( glass, RayOrigin, RayDir );
InterPtE3 = seval( glass, coord( InterPt, 0 ), coord( InterPt, 1 ) );
color( InterPtE3, cyan );
attrib( InterPtE3, "dwidth", 3 );
view( list( InterPtE3, RayLine, glass, axes ), 1 );

resolution = 80;
InterPt = SRINTER( glass, RayOrigin, RayDir );
InterPtE3 = seval( glass, coord( InterPt, 0 ), coord( InterPt, 1 ) );
color( InterPtE3, cyan );
attrib( InterPtE3, "dwidth", 3 );
view( list( InterPtE3, RayLine, glass, axes ), 1 );
```

A complete example of constructing a ray and intersecting it against a surface of a glass at two different resolution, resulting in two different accuracies. See also RESOLUTION.

10.2.99 STANGENT

```
VectorType STANGENT( SurfaceType Srf, ConstantType Direction,
                     NumericType UParam, NumericType VParam )
```

or

```
VectorType STANGENT( TrimSrfType Srf, ConstantType Direction,
                     NumericType UParam, NumericType VParam )
```

Computes the tangent vector to (possibly trimmed) surface **Srf** at the parameter values **UParam** and **VParam** in **Direction**. The returned vector has a unit length.

Example:

```
Tang = STANGENT( Srf, ROW, 0.5, 0.6 );
```

computes the tangent to **Srf** in the ROW direction at the parameter values (0.5, 0.6).

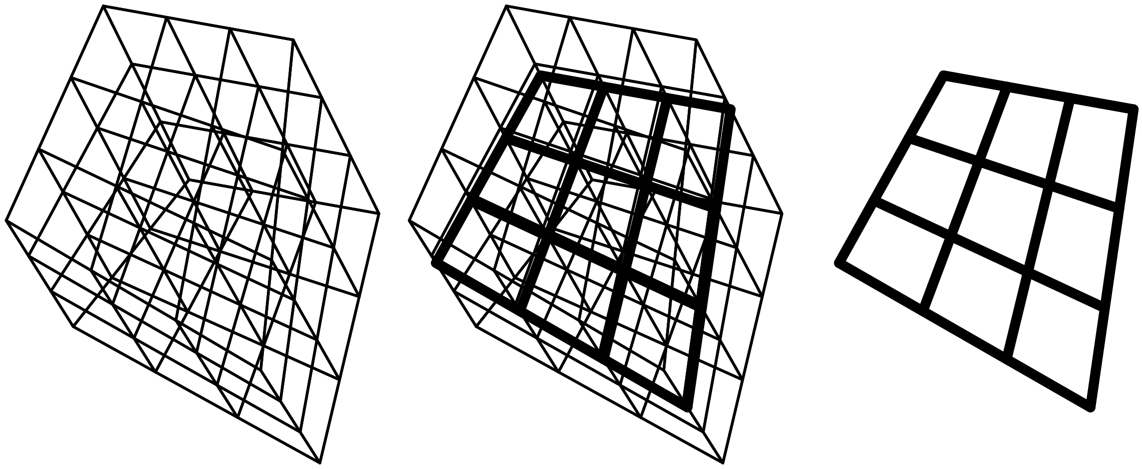


Figure 46: Extracts an iso bilinear surface from a trilinear function, using STRIVAR.

10.2.100 STRIMSRF

```
SurfaceType STRIMSRF( TrimSrfType TSrf )
```

Extracts the surface of a trimmed surface **TSrf**.
Example:

```
Srf = STRIMSRF( TrimSrf );
```

extracts the surface of **TrimSrf**.

10.2.101 STRIVAR

```
SurfaceType STRIVAR( TrivarType TV )
```

Extracts an iso surface from a trivariate function **TV**.
Example:

```
TV1 = tbezier( list( list( list( ctlpt( E3, 0.1, 0.0, 0.8 ),
                                 ctlpt( E3, 0.2, 0.1, 2.4 ) ),
                           list( ctlpt( E3, 0.3, 2.2, 0.2 ),
                                 ctlpt( E3, 0.4, 2.3, 2.0 ) ) ),
                     list( list( ctlpt( E3, 2.4, 0.8, 0.1 ),
                                 ctlpt( E3, 2.2, 0.7, 2.3 ) ),
                           list( ctlpt( E3, 2.3, 2.6, 0.5 ),
                                 ctlpt( E3, 2.1, 2.5, 2.7) ) ) ) );
Srf = STRIVAR( TV1, col, 0.4 );
```

extracts an iso surface of **TV1**, in the **col** direction at parameter value 0.4. See Figure 46.

10.2.102 SURFPREV

```
SurfaceType SURFPREV( CurveType Object )
```

Same as SURFREV but approximates the surface of revolution as a *polynomial* surface. Object must be a polynomial curve. See SURFREV.

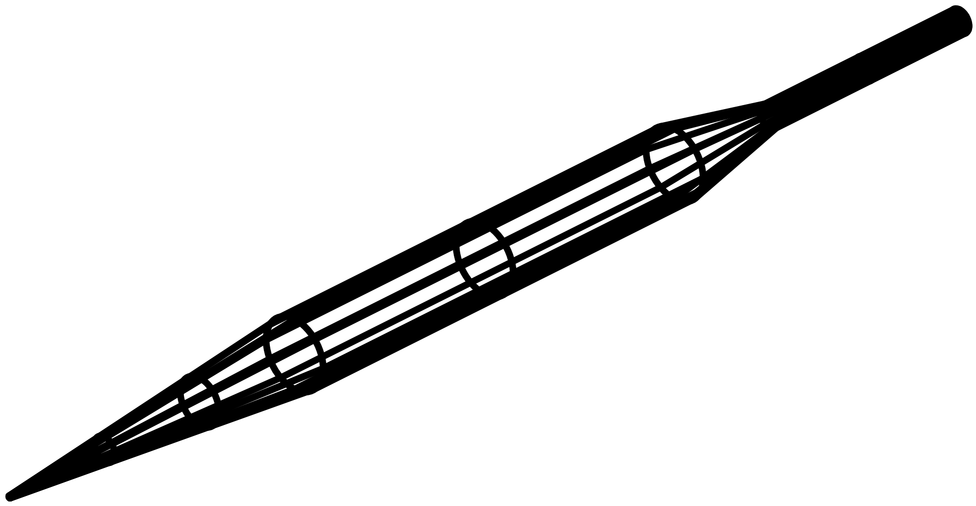


Figure 47: A surface of revolution, VTailAntn in surfrev documentation, can be constructed using SURFREV or SURFPREV.

10.2.103 SURFREV

```
PolygonType SURFREV( PolygonType Object )
```

or

```
SurfaceType SURFREV( CurveType Object )
```

Creates a surface of revolution by rotating the first polygon/curve of the given **Object**, around the Z axis. Use the linear transformation function to position a surface of revolution in a different orientation.

Example:

```
VTailAntn = SURFREV( ctlpt( E3, 0.001, 0.0, 1.0 ) +
                     ctlpt( E3, 0.01,  0.0, 1.0 ) +
                     ctlpt( E3, 0.01,  0.0, 0.8 ) +
                     ctlpt( E3, 0.03,  0.0, 0.7 ) +
                     ctlpt( E3, 0.03,  0.0, 0.3 ) +
                     ctlpt( E3, 0.001, 0.0, 0.0 ) );
```

constructs a piecewise linear Bspline curve in the XZ plane and uses it to construct a surface of revolution by rotating it around the Z axis. See also SURFPREV. See Figure 47.

10.2.104 SWEEPSRF

```
SurfaceType SWEEPSRF( CurveType CrossSection, CurveType Axis,
                      CurveType FrameCrv | VectorType FrameVec | ConstType OFF )
```

Constructs a generalized cylinder surface. This function sweeps a specified cross-section **CrossSection** along the provided **Axis**. By default, when frame specification is **OFF**, the orientation of the cross section is computed using the **Axis** curve tangent and normal. However, unlike the Frenet frame, attempt is made to minimize the normal change, as can happen along inflection points in **Axis**. If a VectorType **FrameVec** is provided as a frame orientation setting, it is used to fix the binormal direction to this value. In other words, the orientation frame has a fixed binormal. If a CurveType **FrameCrv** is specified as a frame orientation setting, this vector field curve is evaluated at each placement of the cross-section to yield the needed binormal.

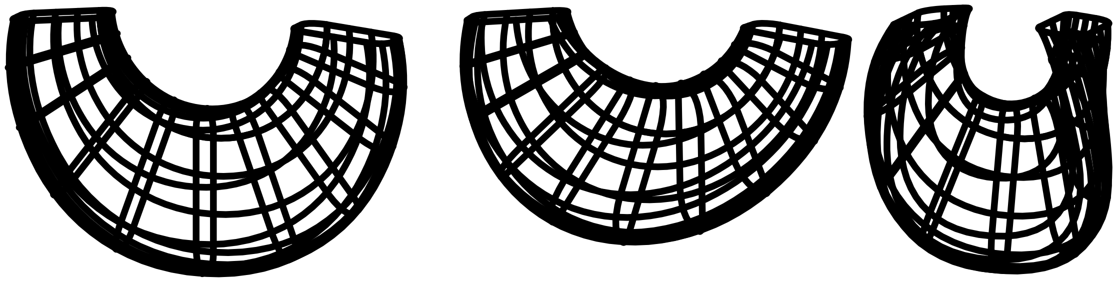


Figure 48: Three examples of the use of SWEEPSRF (Srf1, Srf2, Srf3 from left to right in sweepsrf documentation).

The resulting sweep is only an approximation of the real sweep. The resulting sweep surface will not be exact, in general. Refinement of the axis curve at the proper location, where accuracy is important, should improve the accuracy of the output. The parametric domains of **FrameCrv** do not have to match the parametric domain of **Axis**, and its parametric domain is automatically made compatible by this function.

Example:

```
Cross = arc( vector( 0.2, 0.0, 0.0 ),
             vector( 0.2, 0.2, 0.0 ),
             vector( 0.0, 0.2, 0.0 ) ) +
        arc( vector( 0.0, 0.4, 0.0 ),
             vector( 0.1, 0.4, 0.0 ),
             vector( 0.1, 0.5, 0.0 ) ) +
        arc( vector( 0.8, 0.5, 0.0 ),
             vector( 0.8, 0.3, 0.0 ),
             vector( 1.0, 0.3, 0.0 ) ) +
        arc( vector( 1.0, 0.1, 0.0 ),
             vector( 0.9, 0.1, 0.0 ),
             vector( 0.9, 0.0, 0.0 ) ) +
        ctlpt( E2, 0.2, 0.0 );
Axis = arc( vector( -1.0, 0.0, 0.0 ),
            vector(  0.0, 0.0, 0.1 ),
            vector(  1.0, 0.0, 0.0 ) );
Axis = crefine( Axis, FALSE, list( 0.25, 0.5, 0.75 ) );
Srf1 = SWEEPSRF( Cross, Axis, OFF );
Srf2 = SWEEPSRF( Cross, Axis, vector( 0.0, 1.0, 1.0 ) );
Srf3 = SWEEPSRF( Cross, Axis,
                 cbezier( list( ctlpt( E3,  1.0, 0.0, 0.0 ),
                                ctlpt( E3,  0.0, 1.0, 0.0 ),
                                ctlpt( E3, -1.0, 0.0, 0.0 ) ) ) );
```

constructs a rounded rectangle cross-section and sweeps it along an arc, while orienting it several ways. The axis curve **Axis** is manually refined to better approximate the requested shape.

See also SWPSCLSRF for sweep with scale. See Figure 48.

10.2.105 SWPSCLSRF

```
SurfaceType SWPSCLSRF( CurveType CrossSection, CurveType Axis,
                       NumericType Scale | CurveType ScaleCrv,
                       CurveType FrameCrv | VectorType FrameVec
```

```
                                        | ConstType OFF,
            NumericType ScaleRefine )
```

Constructs a generalized cylinder surface. This function sweeps a specified cross-section **Cross-Section** along the provided **Axis**. The cross-section may be scaled by a constant value **Scale**, or scaled along the **Axis** parametric direction via a scaling curve **ScaleCrv**. By default, when frame specification is **OFF**, the orientation of the cross section is computed using the **Axis** curve tangent and normal. However, unlike the Frenet frame, attempt is made to minimize the normal change, as can happen along inflection points in **Axis**. If a VectorType **FrameVec** is provided as a frame orientation setting, it is used to fix the binormal direction to this value. In other words, the orientation frame has a fixed binormal. If a CurveType **FrameCrv** is specified as a frame orientation setting, this vector field curve is evaluated at each placement of the cross-section to yield the needed binormal. **ScaleRefine** is an integer value to define possible refinement of the **Axis** to reflect the information in **ScalingCrv**. Value of zero will force no refinement while value of $n > 0$ will insert n times the number of control points in **ScaleCrv** into **Axis**, better emulating the scaling requested. The resulting sweep is only an approximation of the real sweep. The scaling and axis placement will not be exact, in general. Manual refinement (in addition to **ScaleRefine**) of the axis curve at the proper location, where accuracy is important, should improve the accuracy of the output. The parametric domains of **ScaleCrv** and **FrameCrv** do not have to match the parametric domain of **Axis**, and their domains are made compatible by this function.

Example:

```
Cross = arc( vector( -0.11, -0.1,  0.0 ),
             vector( -0.1,  -0.1,  0.0 ),
             vector( -0.1,  -0.11, 0.0 ) ) +
        arc( vector(  0.1,  -0.11, 0.0 ),
             vector(  0.1,  -0.1,  0.0 ),
             vector(  0.11, -0.1,  0.0 ) ) +
        arc( vector(  0.11,  0.1,  0.0 ),
             vector(  0.1,   0.1,  0.0 ),
             vector(  0.1,   0.11, 0.0 ) ) +
        arc( vector( -0.1,   0.11, 0.0 ),
             vector( -0.1,   0.1,  0.0 ),
             vector( -0.11,  0.1,  0.0 ) ) +
        ctlpt( E2, -0.11, -0.1 );
scaleCrv = cbspline( 3,
                     list( ctlpt( E2, 0.05, 1.0 ),
                           ctlpt( E2, 0.1,  0.0 ),
                           ctlpt( E2, 0.2,  2.0 ),
                           ctlpt( E2, 0.3,  0.0 ),
                           ctlpt( E2, 0.4,  2.0 ),
                           ctlpt( E2, 0.5,  0.0 ),
                           ctlpt( E2, 0.6,  2.0 ),
                           ctlpt( E2, 0.7,  0.0 ),
                           ctlpt( E2, 0.8,  2.0 ),
                           ctlpt( E2, 0.85, 1.0 ) ),
                     list( KV_OPEN ) );
Axis = circle( vector( 0, 0, 0 ), 1 );
Frame = circle( vector( 0, 0, 0 ), 1 )
        * rotx( 90 ) * trans( vector( 1.5, 0.0, 0.0 ) );
```

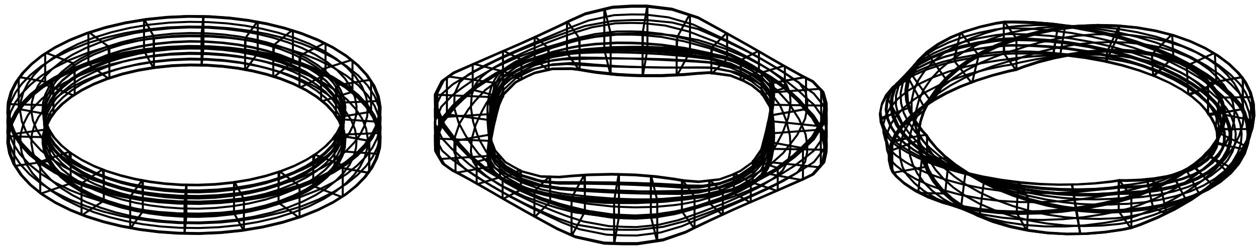


Figure 49: Three examples of the use of SWPSCLSRF (Srf1, Srf2, Srf3 from left to right in swpsclsrf documentation).

```
Srf1 = SWPSCLSRF( Cross, Axis, scaleCrv, off, 0 );
Srf2 = SWPSCLSRF( Cross, Axis, scaleCrv, off, 2 );
Srf3 = SWPSCLSRF( Cross, Axis, 1.0, Frame, 0 );
```

constructs a rounded rectangle cross-section and sweeps it along a circle, while scaling and orienting in several ways. The axis curve **Axis** is automatically refined in **Srf2** to better approximate the requested scaling.

See also SWEEPSRF for sweep with no scale. See Figure 49.

10.2.106 SYMBPROD

```
CurveType SYMBPROD( CurveType Crv1, CurveType Crv2 )
```

or

```
SurfaceType SYMBPROD( SurfaceType Srf1, SurfaceType Srf2 )
```

Computes the symbolic product of the given two curves or surfaces as a curve or surface. The product is computed coordinate-wise.

Example:

```
ProdSrf = SYMBPROD( Srf1, Srf2 )
```

10.2.107 SYMBDPROD

```
CurveType SYMBDPROD( CurveType Crv1, CurveType Crv2 )
```

or

```
SurfaceType SYMBDPROD( SurfaceType Srf1, SurfaceType Srf2 )
```

Computes the symbolic dot (inner) product of the given two curves or surfaces as a *scalar* curve or surface.

Example:

```
DiffCrv = symbdiff( Crv1, Crv2 )
DistSqrCrv = SYMBDPROD( DiffCrv, DiffCrv )
```

Computes a scalar curve that at parameter t is equal to the distance square between Crv1 at t and Crv2.

10.2.108 SYMBCPROD

```
CurveType SYMBCPROD( CurveType Crv1, CurveType Crv2 )
```

or

```
SurfaceType SYMBCPROD( SurfaceType Srf1, SurfaceType Srf2 )
```

Computes the symbolic cross product of the given two curves or surfaces as a curve or surface.
Example:

```
NrmlSrf = SYMBCPROD( sderive( Srf, ROW ), sderive( Srf, COL ) )
```

computes a normal surface as the cross product of the surface two partial derivatives (see SNRMLSRF).

10.2.109 SYMBSUM

```
CurveType SYMBSUM( CurveType Crv1, CurveType Crv2 )
```

or

```
SurfaceType SYMBSUM( SurfaceType Srf1, SurfaceType Srf2 )
```

Computes the symbolic sum of the given two curves or surfaces as a curve or surface. The sum is computed coordinate-wise.
Example:

```
SumCrv = SYMBSUM( Crv1, Crv2 )
```

10.2.110 SYMBDIFF

```
CurveType SYMBDIFF( CurveType Crv1, CurveType Crv2 )
```

or

```
SurfaceType SYMBDIFF( SurfaceType Srf1, SurfaceType Srf2 )
```

Computes the symbolic difference of the given two curves or surfaces as a curve or surface. The difference is computed coordinate-wise.
Example:

```
DiffCrv = SYMBDIFF( Crv1, Crv2 )
DistSqrCrv = symbdprod( DiffCrv, DiffCrv )
```

10.2.111 TBEZIER

```
TrivarType TBEZIER( ListType CtlMesh )
```

Creates a Bezier trivariate using the provided control mesh. **CtlMesh** is a list of planes, each of which is a list of rows, each of which is a list of control points. All control points must be of type (E1-E5, P1-P5), or regular PointType defining the trivariate's control mesh. Surface's point type will be of a space which is the union of the spaces of all points.

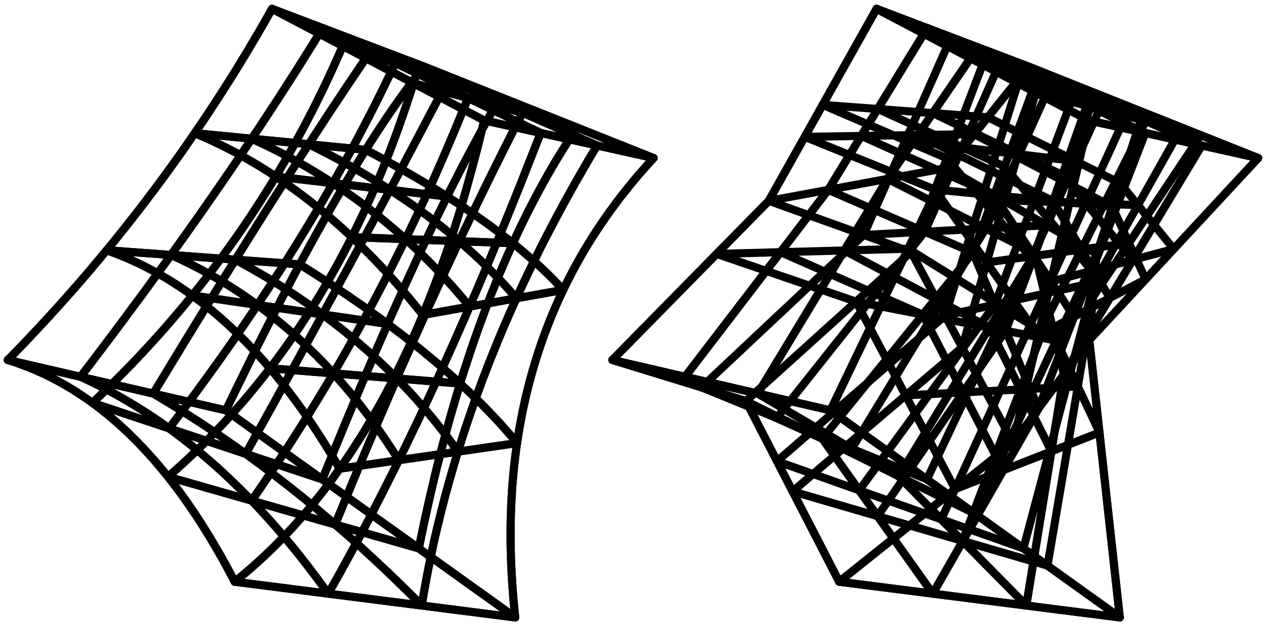


Figure 50: A trivariate Bezier of degree 2 by 3 by 3 (left) and a trilinear Bspline (right). Both share the same control mesh.

The created trivariate is the piecewise polynomial (or rational) function,

$$T(u, v, w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{l} P_{ijk} B_i(u) B_j(v) B_k(w) \tag{15}$$

where P_{ijk} are the control points **CtlMesh**. and l, m and n are the degrees of the trivariate, which are one less than the number of points in the appropriate direction.

Example:

```
TV = TBEZIER( list( list( list( ctlpt( E3, 0.1, 0.1, 0.0 ),
                                ctlpt( E3, 0.2, 0.5, 1.1 ),
                                ctlpt( E3, 0.3, 0.1, 2.2 ) ),
                          list( ctlpt( E3, 0.4, 1.3, 0.5 ),
                                ctlpt( E3, 0.5, 1.7, 1.7 ),
                                ctlpt( E3, 0.6, 1.3, 2.9 ) ),
                          list( ctlpt( E3, 0.7, 2.4, 0.5 ),
                                ctlpt( E3, 0.8, 2.6, 1.4 ),
                                ctlpt( E3, 0.9, 2.8, 2.3 ) ) ),
                    list( list( ctlpt( E3, 1.1, 0.1, 0.5 ),
                                ctlpt( E3, 1.3, 0.2, 1.7 ),
                                ctlpt( E3, 1.5, 0.3, 2.9 ) ),
                          list( ctlpt( E3, 1.7, 1.2, 0.0 ),
                                ctlpt( E3, 1.9, 1.4, 1.2 ),
                                ctlpt( E3, 1.2, 1.6, 2.4 ) ),
                          list( ctlpt( E3, 1.4, 2.3, 0.9 ),
                                ctlpt( E3, 1.6, 2.5, 1.7 ),
                                ctlpt( E3, 1.8, 2.7, 2.5 ) ) ) ) );
```

creats a trivariate Bezier which is linear in the first direction, and quadratic in the second and third. See Figure 50.

10.2.112 TBSPLINE

```
TrivarType TBSPLINE( NumericType UOrder,
                     NumericType VOrder,
                     NumericType WOrder,
                     ListType CtlMesh,
                     ListType KnotVectors )
```

Creates a Bspline trivariate with the provided **UOrder**, **VOrder** and **WOrder** orders, the control mesh **CtlMesh**, and the three knot vectors in **KnotVectors**. **CtlMesh** is a list of planes, each of which is a list of rows, each of which is a list of control points. All control points must be of point type (E1-E5, P1-P5), or regular PointType defining the trivariate's control mesh. Trivariate's point type will be of a space which is the union of the spaces of all points. **KnotVectors** is a list of three knot vectors. Each knot vector is a list of NumericType knots of length **#CtlPtList** plus the **Order**. If, however, the length of the knot vector is equal to **#CtlPtList + Order + Order - 1** the curve is assumed *periodic.* The knot vector may also be a list of a single constant KV_OPEN or KV_FLOAT or KV_PERIODIC, in which a uniform knot vector with the appropriate length and with open, floating or periodic end condition will be constructed automatically.

The created surface is the piecewise polynomial (or rational) surface,

$$T(u,v,w) = \sum_{i=0}^{m} \sum_{j=0}^{n} \sum_{k=0}^{l} P_{ijk} B_{i,\chi}(u) B_{j,\xi}(v) B_{k,\phi}(w) \tag{16}$$

where P_{ijk} are the control points **CtlMesh**, and l, m and n are the degrees of the surface, which are one less than **UOrder**, **VOrder** and **WOrder** and χ, ξ and ϕ are the three knot vectors of the trivariate.

Example:

```
TV = TBSPLINE( 2, 2, 2,
               list( list( list( ctlpt( E3, 0.1, 0.1, 0.0 ),
                                 ctlpt( E3, 0.2, 0.5, 1.1 ),
                                 ctlpt( E3, 0.3, 0.1, 2.2 ) ),
                           list( ctlpt( E3, 0.4, 1.3, 0.5 ),
                                 ctlpt( E3, 0.5, 1.7, 1.7 ),
                                 ctlpt( E3, 0.6, 1.3, 2.9 ) ),
                           list( ctlpt( E3, 0.7, 2.4, 0.5 ),
                                 ctlpt( E3, 0.8, 2.6, 1.4 ),
                                 ctlpt( E3, 0.9, 2.8, 2.3 ) ) ),
                     list( list( ctlpt( E3, 1.1, 0.1, 0.5 ),
                                 ctlpt( E3, 1.3, 0.2, 1.7 ),
                                 ctlpt( E3, 1.5, 0.3, 2.9 ) ),
                           list( ctlpt( E3, 1.7, 1.2, 0.0 ),
                                 ctlpt( E3, 1.9, 1.4, 1.2 ),
                                 ctlpt( E3, 1.2, 1.6, 2.4 ) ),
                           list( ctlpt( E3, 1.4, 2.3, 0.9 ),
                                 ctlpt( E3, 1.6, 2.5, 1.7 ),
                                 ctlpt( E3, 1.8, 2.7, 2.5 ) ) ) ),
               list( list( KV_OPEN ),
                     list( KV_OPEN ),
                     list( KV_OPEN ) ) );
```

constructs a trilinear Bspline trivariate with open end conditions. See Figure 50.

10.2.113 TDERIVE

```
TrivarType TDERIVE( TrivarType TV, NumericType Dir )
```

Returns a vector field trivariate representing the differentiated trivariate in the given direction (ROW, COL, or DEPTH). Evaluation of the returned trivariate at a given parameter value will return a vector representing the partial derivative of **TV** in **Dir** at that parameter value.

```
TV = tbezier( list( list( list( ctlpt( E1, 0.1 ),
                                ctlpt( E1, 0.2 ) ),
                          list( ctlpt( E1, 0.3 ),
                                ctlpt( E1, 0.4 ) ) ),
                    list( list( ctlpt( E1, 2.4 ),
                                ctlpt( E1, 2.2 ) ),
                          list( ctlpt( E1, 2.3 ),
                                ctlpt( E1, 2.1 ) ) ) ) );

DuTV = TDERIVE( TV, ROW );
DvTV = TDERIVE( TV, COL );
DwTV = TDERIVE( TV, DEPTH );
```

computes the gradiate of a scalar trivariate field, by computing its partials with respect to u, v, and w.

10.2.114 TDIVIDE

```
TrivarType TDIVIDE( TrivarType TV, ConstantType Direction,
                                                     NumericType Param )
```

Subdivides a trivariate into two at the specified parameter value **Param** in the specified **Direction** (ROW, COL, or DEPTH). **TV** can be either a Bspline trivairate in which **Param** must be conatined in the parametric domain of the trivariate, or a Bezier trivariate in which **Param** must be in the range of zero to one.

It returns a list of the two sub-trivariates. The individual trivariates may be extracted from the list using the **NTH** command.

Example:

```
TvDiv = TDIVIDE( Tv2, depth, 0.3 );
Tv2a = nth( TvDiv, 1 ) * tx( -2.2 );
Tv2b = nth( TvDiv, 2 ) * tx( 2.0 );
```

subdivides **Tv2** at the parameter value of 0.3 in the DEPTH direction, See Figure 51.

10.2.115 TEVAL

```
CtlPtType TEVAL( TrivarType TV,
                 NumericType UParam,
                 NumericType VParam,
                 NumericType WParam )
```

Evaluates the provided trivariate **TV** at the given **UParam**, **VParam** and **WParam** values. **UParam**, **VParam**, **WParam** must be contained in the surface parametric domain if **TV** is a Bspline surface, or between zero and one if **TV** is a Bezier trivariate. The returned control point has the same type as the control points of **TV**.

Example:

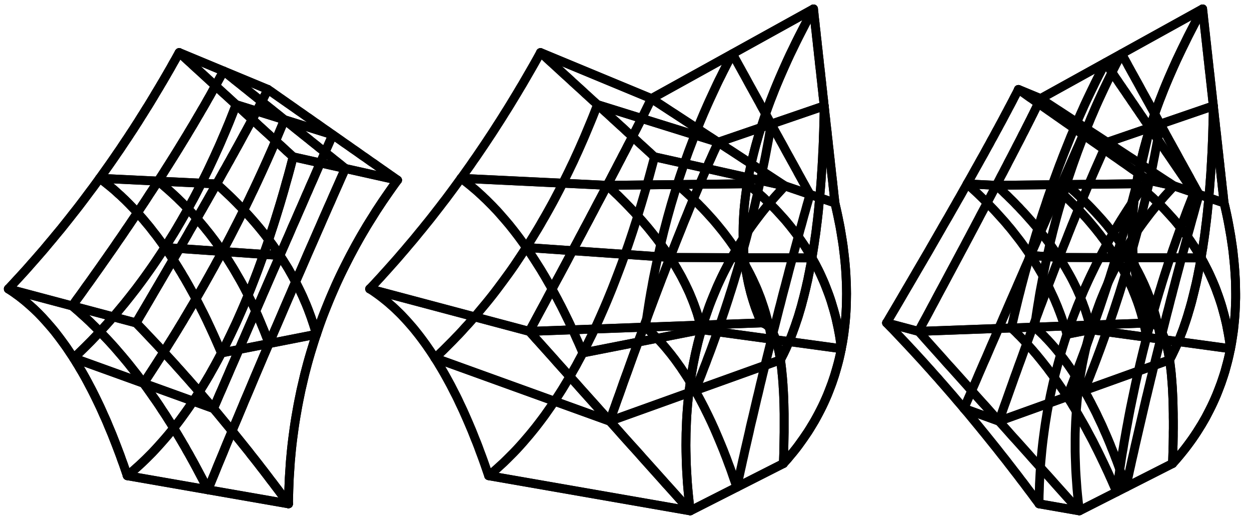


Figure 51: A trivariate can be subdivided into two distinct regions using TDIVIDE.

```
CPt = TEVAL( TV1, 0.25, 0.22, 0.7 );
```

evaluates **TV** at the parameter values of (0.25, 0.22, 0.7).

10.2.116 TEXTGEOM

```
AnyType TEXTGEOM( StringType Str, VectorType Spacing, NumericType Scaling )
```

Creates a displayable geometry that represents the text in **Str**, with **Spacing** space between individual characters. Each character is scaled by **Scaling** where scaling of one generates a close to a unit size character.

Example:

```
a = TEXTGEOM("Text", vector( 0.12, 0, 0 ), 0.1 );
b = TEXTGEOM("IRIT", vector( 0, -0.12, 0 ), 0.1 );
```

Creates an horizontal **Text** and a vertical top to bottom **IRIT**, both as geometrical objects.

10.2.117 TFROMSRFS

```
TrivarType TFROMSRFS( ListType SrfList, NumericType OtherOrder )
```

Constructs a trivariate by substituting the surfaces in **SrfList** as planes in a control mesh of a trivariate. Surfaces in **SrfList** are made compatible by promoting Bezier surfaces to Bsplines if necessary, and raising degree and refining as required before substituting the control meshes of the surfaces as planes in the mesh of the trivariate. The other, third, direction order is set by **OtherOrder**, which cannot be larger than the number of surfaces.

The trivariate interpolates the first and last surfaces only.

Example:

```
s1 = sbezier( list( list( ctlpt( E3, -0.5, -0.5, 0 ),
                          ctlpt( E3, -0.5,  0.5, 0 ) ),
                    list( ctlpt( E3,  0.5, -0.5, 0 ),
                          ctlpt( E3,  0.5,  0.5, 0 ) ) ) ) * sc( 0.3 );
Srfs = list( s1 * sc( 2.0 ),
```

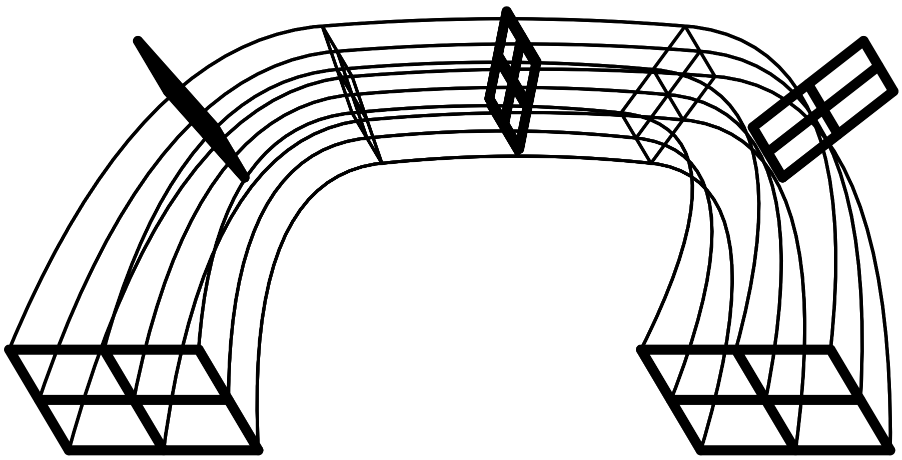


Figure 52: A trivariate (thin lines) is constructed via five planar surfaces (thick lines) using the TFROMSRFS constructor...

```
              s1 * sx( 1.4 ) * ry( 45 ) * tz( 1.0 ),
              s1 * ry( 90 ) * trans( vector( 1.0, 0.0, 1.1 ) ),
              s1 * sx( 1.4 ) * ry( 135 ) * trans( vector( 2.0, 0.0, 1.0 ) ),
              s1 * sc( 2.0 ) * ry( 180 ) * trans( vector( 2.0, 0.0, 0.0 ) ) );
color( Srfs, red );

ts = tfromsrfs( Srfs, 3 );
color( ts, green );
view( list( Srfs, ts ), on );
```

Constructs a trivariate from five planar surfaces and display both the trivariate and the five planar surfaces, in different colors. See Figure 52.

10.2.118 TINTERP

```
TrivarType TINTERP( TrivarType TV );
```

Given a trivariate data structure, computes a new trivariate in the same function space (i.e. same knot sequences and orders) that interpolates the given triavriate, **TV**, at the node parameter values.

Example:

```
tv = tbspline( 3, 3, 2,
               list( list( list( ctlpt( E3, 0.1, 0.1, 0.0 ),
                                 ctlpt( E3, 0.2, 0.5, 1.1 ),
                                 ctlpt( E3, 0.3, 0.1, 2.2 ) ),
                           list( ctlpt( E3, 0.4, 1.3, 0.5 ),
                                 ctlpt( E3, 0.5, 1.7, 1.7 ),
                                 ctlpt( E3, 0.6, 1.3, 2.9 ) ),
                           list( ctlpt( E3, 0.7, 2.4, 0.5 ),
                                 ctlpt( E3, 0.8, 2.6, 1.4 ),
                                 ctlpt( E3, 0.9, 2.8, 2.3 ) ) ),
                     list( list( ctlpt( E3, 1.1, 0.1, 0.5 ),
                                 ctlpt( E3, 1.3, 0.2, 1.7 ),
                                 ctlpt( E3, 1.5, 0.3, 2.9 ) ),
```

```
                                list( ctlpt( E3, 1.7, 1.2, 0.0 ),
                                      ctlpt( E3, 1.9, 1.4, 1.2 ),
                                      ctlpt( E3, 1.2, 1.6, 2.4 ) ),
                                list( ctlpt( E3, 1.4, 2.3, 0.9 ),
                                      ctlpt( E3, 1.6, 2.5, 1.7 ),
                                      ctlpt( E3, 1.8, 2.7, 2.5 ) ) ),
                          list( list( ctlpt( E3, 2.8, 0.1, 0.4 ),
                                      ctlpt( E3, 2.6, 0.7, 1.3 ),
                                      ctlpt( E3, 2.4, 0.2, 2.2 ) ),
                                list( ctlpt( E3, 2.2, 1.1, 0.4 ),
                                      ctlpt( E3, 2.9, 1.2, 1.5 ),
                                      ctlpt( E3, 2.7, 1.3, 2.6 ) ),
                                list( ctlpt( E3, 2.5, 2.9, 0.7 ),
                                      ctlpt( E3, 2.3, 2.8, 1.7 ),
                                      ctlpt( E3, 2.1, 2.7, 2.7 ) ) ) ),
                    list( list( KV_OPEN ),
                          list( KV_OPEN ),
                          list( KV_OPEN ) ) );
tvi = TINTERP( tv );
```

creates a quadratic by quaratic by linear trivairatiate **tvi** that interpolates the control points of **tv** at the node parameter values.

10.2.119 TORUS

```
PolygonType TORUS( VectorType Center, VectorType Normal,
                   NumericType MRadius, NumericType mRadius )
```

Creates a TORUS geometric object, defined by **Center** as the center of the TORUS, **Normal** as the normal to the main plane of the TORUS, **MRadius** and **mRadius** as the major and minor radii of the TORUS. See RESOLUTION for the accuracy of the TORUS approximation as a polygonal model.

Example:

```
T = TORUS( vector( 0.0, 0.0, 0.0), vector( 0.0, 0.0, 1.0), 0.5, 0.2 );
```

constructs a torus with major plane as the XY plane, major radius of 0.5, and minor radius of 0.2. See Figure 53.

10.2.120 TREFINE

```
TrivarType TREFINE( TrivarType TV, ConstantType Direction,
                    NumericType Replace, ListType KnotList )
```

Provides the ability to **Replace** a knot vector of **TV** or refine it in the specified direction **Direction** (ROW, COL, or DEPTH). **KnotList** is a list of knots to refine **TV** at. All knots should be contained in the parametric domain of **TV** in **Direction**. If the knot vector is replaced, the length of **KnotList** should be identical to the length of the original knot vector of **TV** in **Direction**. If **TV** is a Bezier trivariate, it is automatically promoted to be a Bspline trivariate.

Example:

```
TV = TREFINE( TREFINE( TREFINE( TV,
                                ROW, FALSE, list( 0.333, 0.667 ) ),
                       COL, FALSE, list( 0.333, 0.667 ) ),
              DEPTH, FALSE, list( 0.333, 0.667 ) );
```

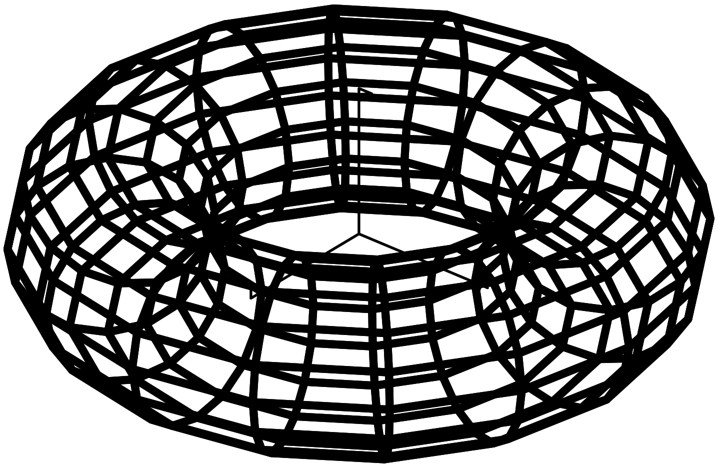


Figure 53: A torus primitive can be constructed using a TORUS constructor...

refines **TV** in all directions by adding two more knots at 0.333 and 0.667.

10.2.121 TREGION

```
TrivarType TREGION( TrivarType Srf, ConstantType Direction,
                    NumericType MinParam, NumericType MaxParam )
```

Extracts a region of **TV** between **MinParam** and **MaxParam** in the specified **Direction**. Both **MinParam** and **MaxParam** should be contained in the parametric domain of **TV** in **Direction**.

Example:

```
Tv1 = tbezier( list( list( list( ctlpt( E3, 0.1, 0.0, 0.8 ),
                                 ctlpt( E3, 0.2, 0.1, 2.4 ) ),
                           list( ctlpt( E3, 0.3, 2.2, 0.2 ),
                                 ctlpt( E3, 0.4, 2.3, 2.0 ) ) ),
                     list( list( ctlpt( E3, 2.4, 0.8, 0.1 ),
                                 ctlpt( E3, 2.2, 0.7, 2.3 ) ),
                           list( ctlpt( E3, 2.3, 2.6, 0.5 ),
                                 ctlpt( E3, 2.1, 2.5, 2.7) ) ) ) );

Tv1r1 = TREGION( Tv1, row, 0.1, 0.2 );
Tv1r2 = TREGION( Tv1, row, 0.4, 0.6 );
Tv1r3 = TREGION( Tv1, row, 0.99, 1.0 );
```

extracts three regions of **Tv1** along the ROW direction. See Figure 54.

10.2.122 TRIMSRF

```
TrimSrfType TRIMSRF( SurfaceType Srf,
                     CurveType TrimCrv,
                     NumericType HasUpperLevel )
```

or

```
TrimSrfType TRIMSRF( SurfaceType Srf,
                     ListType TrimCrvs,
                     NumericType HasUpperLevel )
```

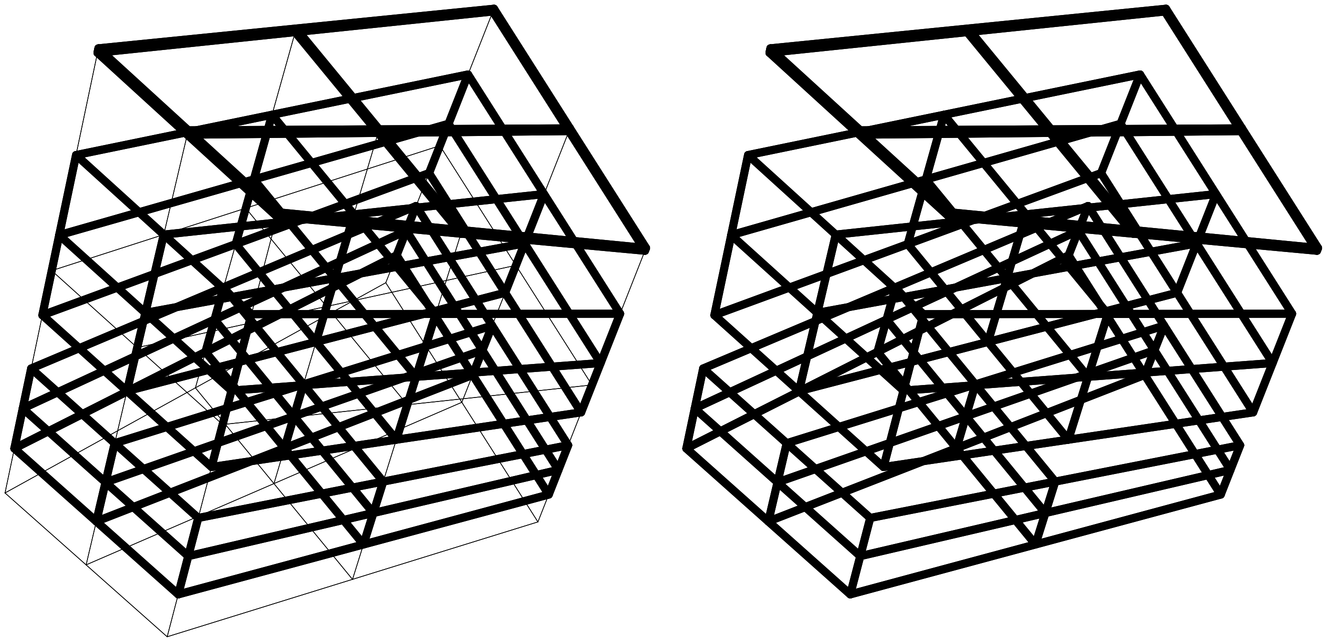


Figure 54: A region can be extracted from a freeform trivariate using TREGION.

Creates a trimmed surface from the provided surface **Srf** and the trimming curve **TrimCrv** or curves **TrimCrvs**. If **HasUpperLevel** is FALSE, an additional trimming curve is automatically being added that contains the entire parametric domain of **Srf**. No validity test is performed on the trimming curves which are assumed two dimensional curves contained in the parametric domain of **Srf**.

Example:

```
spts = list( list( ctlpt( E3, 0.1, 0.0, 1.0 ),
                   ctlpt( E3, 0.3, 1.0, 0.0 ),
                   ctlpt( E3, 0.0, 2.0, 1.0 ) ),
             list( ctlpt( E3, 1.1, 0.0, 0.0 ),
                   ctlpt( E3, 1.3, 1.5, 2.0 ),
                   ctlpt( E3, 1.0, 2.1, 0.0 ) ),
             list( ctlpt( E3, 2.1, 0.0, 2.0 ),
                   ctlpt( E3, 2.3, 1.0, 0.0 ),
                   ctlpt( E3, 2.0, 2.0, 2.0 ) ),
             list( ctlpt( E3, 3.1, 0.0, 0.0 ),
                   ctlpt( E3, 3.3, 1.5, 2.0 ),
                   ctlpt( E3, 3.0, 2.1, 0.0 ) ),
             list( ctlpt( E3, 4.1, 0.0, 1.0 ),
                   ctlpt( E3, 4.3, 1.0, 0.0 ),
                   ctlpt( E3, 4.0, 2.0, 1.0 ) ) );
sb = sbspline( 3, 3, spts, list( list( KV_OPEN ), list( KV_OPEN ) ) );

TCrv1 = cbspline( 2,
                  list( ctlpt( E2, 0.3, 0.3 ),
                        ctlpt( E2, 0.7, 0.3 ),
                        ctlpt( E2, 0.7, 0.7 ),
                        ctlpt( E2, 0.3, 0.7 ),
                        ctlpt( E2, 0.3, 0.3 ) ),
                  list( KV_OPEN ) );
```

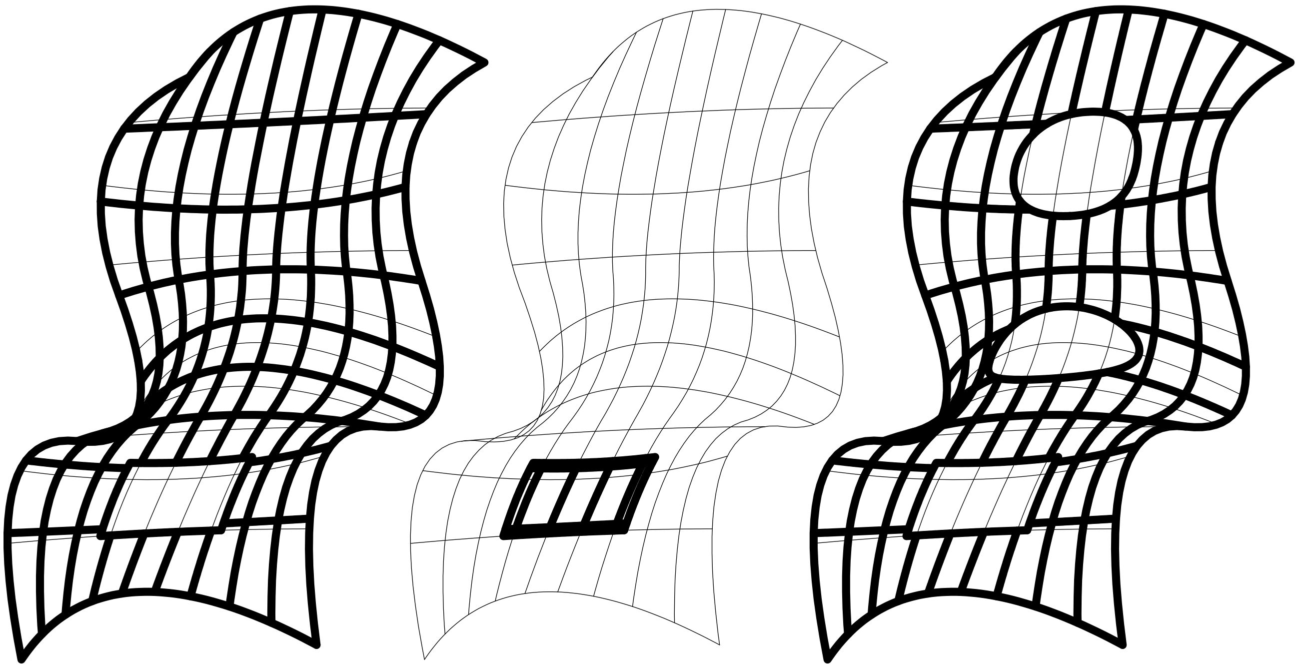


Figure 55: Three trimmed surfaces created from the same Bspline surface. In thin lines is the original surface while in thick lines are the trimmed surfaces.

```
TCrv2 = circle( vector( 0.5, 0.5, 0.0 ), 0.25 );
TCrv3 = cbspline( 3,
                  list( ctlpt( E2, 0.3, 0.3 ),
                        ctlpt( E2, 0.7, 0.3 ),
                        ctlpt( E2, 0.7, 0.7 ),
                        ctlpt( E2, 0.3, 0.7 ) ),
                  list( KV_PERIODIC ) );

TSrf1 = TRIMSRF( sb, TCrv1, false );
TSrf2 = TRIMSRF( sb, TCrv1, true );
TSrf3 = TRIMSRF( sb, list( TCrv1, TcRv2 * ty( 1 ), TCrv3 * ty( 2 ) ),
                 false );
```

constructs three trimmed surfaces. **Tsrf1** contains the outer boundary and excludes what is inside **TCrv1**, **TSrf2** contains only the domain inside **TCrv1**. **TCrv3** has three holes corresponds to the three trimming curves. See Figure 55.

10.3 Object transformation functions

All the routines in this section construct a 4 by 4 homogeneous transformation matrix representing the required transform. These matrices may be concatenated to achieve more complex transforms using the matrix multiplication operator $*$. For example, the expression

```
m = trans( vector( -1, 0, 0 ) ) * rotx( 45 ) * trans( vector( 1, 0, 0 ) );
```

constructs a transform to rotate an object around the $X = 1$ line, 45 degrees. A matrix representing the inverse transformation can be computed as:

```
InvM = m ^ -1
```

See also overloading of the - operator.

10.3.1 HOMOMAT

```
MatrixType HOMOMAT( ListType MatData )
```

Creates an arbitrary homogeneous transformation matrix by manually providing its 16 coefficients. Example:

```
for ( a = 1, 1, 720 / step,
      view_mat = save_mat *
                 HOMOMAT( list( list( 1, 0, 0, 0 ),
                                list( 0, 1, 0, 0 ),
                                list( 0, 0, 1, -a * step / 500 ),
                                list( 0, 0, 0, 1 ) ) ):
      view( list( view_mat, axes ), on )
);
```

looping and viewing through a sequence of perspective transforms, created using the HOMOMAT constructor.

10.3.2 ROTVEC

```
MatrixType ROTVEC( VectorType Vec, NumericType Angle )
```

Creates a rotation around the vector **Vec** matrix with **Angle** degrees.

10.3.3 ROTX

```
MatrixType ROTX( NumericType Angle )
```

Creates a rotation around the X transformation matrix with **Angle** degrees.

10.3.4 ROTY

```
MatrixType ROTY( NumericType Angle )
```

Creates a rotation around te Y transformation matrix with **Angle** degrees.

10.3.5 ROTZ

```
MatrixType ROTZ( NumericType Angle )
```

Creates a rotation around the Z transformation matrix with **Angle** degrees.

10.3.6 ROTZ2V

```
MatrixType ROTZ2V( VectorType Dir )
```

Creates a rotation matrix that takes Z axis into **Dir**. Length of **Dir** is ignored.

10.3.7 ROTZ2V2

```
MatrixType ROTZ2V2( VectorType Dir, VectorType Dir2 )
```

Creates a rotation matrix that takes Z axis into **Dir**, while the X axis is aligned with **Dir2**. The lengths of **Dir** and **Dir2** are ignored.

10.3.8 SCALE

```
MatrixType SCALE( VectorType ScaleFactors )
```

Creates a scaling by the **ScaleFactors** transformation matrix.

10.3.9 TRANS

```
MatrixType TRANS( VectorType TransFactors )
```

Creates a translation by the **TransFactors** transformation matrix.

10.4 General purpose functions

10.4.1 ATTRIB

```
ATTRIB( AnyType Object, StringType Name, AnyType Value )
```

Provides a mechanism to add an attribute of any type to an **Object**, with name **Name** and value **Value**. This ATTRIB function is tuned and optimized toward numeric values or strings as **Value** although any other object type can be saved as attribue.

These attributes may be used to pass information to other programs about this object, and are saved with the objects in data files. For example,

```
ATTRIB(Glass, "rgb", "255,0,0");
ATTRIB(Glass, "refract", "1.4");
     .
     .
     .
RMATTR(Glass, "rgb");   # Removes "rgb" attribute.
```

sets the RGB color and refraction index of the **Glass** object and later removes the RGB attribute.

Attribute names are case insensitive. Spaces are allowed in the **Value** string, as well as the double quote itself, although the latter must be escaped:

```
ATTRIB(Glass, "text", "Say \"this is me\"");
```

See also RMATTR for removal of attributes as well as AWIDTH, ADWIDTH, and COLOR.

10.4.2 ADWIDTH

```
ADWIDTH( GeometricType Object, NumericType DWidth )
```

Sets the width of the object. This display width is used in pixels in display devices for width of line drawing, if supported by the display device. See also ATTRIB, COLOR, and AWIDTH.

This function is equivament to using,
ATTRIB(**Object**, "dwidth", **Width**);

10.4.3 AWIDTH

```
AWIDTH( GeometricType Object, NumericType Width )
```

Sets the width of the object to one of those specified below. This width is used in real object side dimensions in tools such as scan converters and rendering tools for rendering lines and curves, as well as postscript. See also ATTRIB, COLOR, and ADWIDTH.

This function is equivament to using,
ATTRIB(**Object**, "width", **Width**);

10.4.4 CHDIR

```
CHDIR( StringType NewDir )
```

Sets the current working directory to be **NewDir**.

10.4.5 CLNTCLOSE

```
CLNTCLOSE( NumericType Handler, NumericType Kill )
```

Closes a communication channel to a client. **Handler** contains the index of the communication channel opened via CLNTEXEC. If **Kill**, the client is send an exit request for it to die. Otherwise, the communication is closed and the client is running stand alone. See also CLNTREAD, CLNTWRITE, and CLNTEXEC.

Example:

```
h2 = clntexec( "nuldrvs -s-" );
      .
      .
      .

CLNTCLOSE( h2,TRUE );
```

closes the connection to the nuldrvs client, opened via CLNTEXEC.

10.4.6 CLNTWRITE

```
CLNTWRITE( NumericType Handler, AnyType Object )
```

Writes one object **Object** to a communication channel of a client. **Handler** contains the index of the communication channel opened via CLNTEXEC. See also CLNTREAD, CLNTCLOSE, and CLNTEXEC.

Example:

```
h2 = clntexec( "nuldrvs -s-" );
      .
      .

CLNTWRITE( h2, Model );
      .
      .

clntclose( h2,TRUE );
```

writes the object named Model to client through communication channel h2.

10.4.7 COLOR

```
COLOR( GeometricType Object, NumericType Color )
```

Sets the color of the object to one of those specified below. Note that an object has a default color (see IRIT.CFG file) according to its origin - loaded with the LOAD command, PRIMITIVE, or BOOLEAN operation result. The system internally supports colors (although you may have a B&W

system) and the colors recognized are: **BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, and WHITE**.

See the ATTRIB command for more fine control of colors using the RGB attribute. See also AWIDTH and AWIDTH.

This function is equivament to using,

ATTRIB(**Object**, "color", **Width**);

10.4.8 COMMENT

```
COMMENT
```

Two types of comments are allowed:

1. One-line comment: starts anywhere in a line at the '#' character, up to the end of the line.

2. Block comment: starts at the COMMENT keyword followed by a unique character (anything but white space), up to the second occurrence of that character. This is a fast way to comment out large blocks.

Example:

```
COMMENT $
  This is a comment
$
```

10.4.9 ERROR

```
ERROR( StringType Message);
```

Breaks the execution and returns to IRIT main loop, after printing **Message** to the screen. May be useful in user defined function to break execution in cases of fatal errors.

10.4.10 EXIT

```
EXIT();
```

Exits from the solid modeler. NO warning is given!

10.4.11 FOR

```
FOR( NumericType Start, NumericType Increment, NumericType End, AnyType Body )
```

Executes the **Body** (see below), while the FOR loop conditions hold. **Start, Increment, End** are evaluated first, and the loop is executed while <= **End** if **Increment** > **0**, or while >= **End** if **Increment** < **0**. If **Start** is of the form "Variable = Expression", then that variable is updated on each iteration, and can be used within the body. The body may consist of any number of regular commands, separated by COLONs, including nesting FOR loops to an arbitrary level.

Example:

```
step = 10;
rotstepx = rotx(step);
FOR ( a = 1, 1, 360 / step,
    view_mat = rotstepx * view_mat:
    view( list( view_mat, axes ), ON )
);
```

Displays axes with a view direction that is rotated 10 degrees at a time around the X axis.

10.4.12 HELP

```
HELP( StringType Subject )
```

Provides help on the specified Subject.
Example:

```
HELP("");
```

will list all *IRIT* help subjects.

10.4.13 FREE

```
FREE( GeometricType Object )
```

Because of the usually huge size of geometric objects, this procedure may be used to free them. Reassigning a value (even of different type) to a variable automatically releases the old variable's allocated space as well.

10.4.14 FUNCTION

```
FuncName = FUNCTION(Prm1, Prm2, ... , PrmN):LclVal1:LclVar2: ... :LclVarM:
    FuncBody;
```

Defines a function named FuncName with N parameters and M local variables ($N, M >= 0$). Here is a (simple) example of a function with no local variables and a single parameter that computes the square of a number:

```
sqr = FUNCTION(x):
   return = x * x;
```

Functions can be defined with optional parameters and optional local variables. A function's body may contain an arbitrary set of expressions including for/while loops, (user) function calls, or even recursive function calls, all separated by colons. The returned value of the function is the value of an automatically defined local variable named return. The return variable is a regular local variable within the scope of the function and can be used as any other variable.

If a variable's name is found in neither the local variable list nor the parameter list, it is searched in the global variable list (outside the scope of the function). Binding of names of variables is static as in the C programming language.

Because binding of variables is performed in execution time, there is a somewhat less restrictive type checking of parameters of functions that are invoked within a user's defined function.

A function can invoke itself, i.e., it can be recursive. However, since a function should be defined when it is called, a dummy function should be defined before the recursive one is defined:

```
factorial = function(x):return = x; # Dummy function.
factorial = function(x):
    if (x <= 1, return = 1, return = x * factorial(x - 1));
```

Overloading is valid inside a function as it is outside. For example, for

```
add = FUNCTION(x, y):
   return = x + y;
```

the following function calls are all valid:

```
add(1, 2);
add(vector(1,2,3), point(1,2,3));
add(box(vector(-3, -2, -1), 6, 4, 2), box(vector(-4, -3, -2), 2, 2, 4));
```

Finally, here is a more interesting example that computes an approximation of the length of a curve, using the sqr function defined above:

```
distptpt = FUNCTION(pt1, pt2):
    return = sqrt(sqr(coord(pt1, 1) - coord(pt2, 1)) +
                  sqr(coord(pt1, 2) - coord(pt2, 2)) +
                  sqr(coord(pt1, 3) - coord(pt2, 3)));

crvlength = FUNCTION(crv, n):pd:t:t1:t2:dt:pt1:pt2:i:
    return = 0.0:
    pd = pdomain(crv):
    t1 = nth(pd, 1):
    t2 = nth(pd, 2):
    dt = (t2 - t1) / n:
    pt1 = coerce(ceval(crv, t1), e3):
    for (i = 1, 1, n,
         pt2 = coerce(ceval(crv, t1 + dt * i), e3):
         return = return + distptpt(pt1, pt2):
         pt1 = pt2);
```

Try, for example:

```
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 30) / 2;
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 100) / 2;
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 300) / 2;
```

See PROCEDURE and IRITSTATE's "DebugFunc" for more.

10.4.15 IF

```
IF( NumericType Cond, AnyType TrueBody { , AnyType FalseBody } )
```

Executes **TrueBody** (group of regular commands, separated by COLONs - see FOR loop) if the **Cond** holds, i.e., it is a numeric value other than zero, or optionally, if it exists, executes **FalseBody** if the **Cond** does not hold, i.e., it evaluates to a numeric value equal to zero.

Examples:

```
IF ( machine == IBMOS2, resolution = 5, resolution = 10 );
IF ( a > b, max = a, max = b );
```

sets the resolution to be 10, unless running on an IBMOS2 system, in which case the RESOLUTION variable will be set to 5 in the first statement, and set max to the maximum of a and b in the second statement.

10.4.16 INCLUDE

```
INCLUDE( StringType FileName )
```

Executes the script file **FileName**. Nesting of include file is allowed up to 10 levels deep. If an error occurs, all open files in all nested files are closed and data are waited for at the top level (standard input).

A script file can contain any command the solid modeler supports.

Example:

```
INCLUDE( "general.irt" );
```

includes the file "general.irt".

10.4.17 IRITSTATE

```
IRITSTATE( StringType State, AnyType Data )
```

Sets a state variable in the *IRIT* solid modeller. Current supported state variables are,

State Name	Data Type	Comments
InterpProd	ConstantType	TRUE for Bspline sym. products via interpolation FALSE for Bspline sym. products via bezier
DebugMalloc	StringType	If "Reset", memory allocation is cleared/reset. No "Free unallocated pointer" test after "Reset". If "Print", all allocated blocks are printed. Otherwise, used as "address, n": ptr address to search for with abort() called after n mallocs.
DebugFunc	NumericType	> 0 user func. debug information. > 2 print params on entry, ret. val. on exit. > 4 global var. list
FloatFrmt	StringType	Specifies a new printf floating point format.
InterCrv	NumericType	If TRUE Boolean operations creates only intersection curves. If FALSE, full Boolean operation results.
Coplanar	NumericType	If TRUE, Coplanar polygons are handled by Boolean operations.
PolySort	NumericType	Axis of Polygon Intersection sweep in Boolean operations: 0 for X axis, 1 for Y axis, 2 for Z axis.
EchoSource	NumericType	If TRUE, irit scripts are echoed to stdout.
DumpLevel	NumericType	Controls the way variables/expressions ar dumped. Only object names/types if >= 0, Scalars and vectors are dumped if >= 1, Curves and Surfaces are dumped if DumpLvl >= 2, Polygons/lines are dumped if DumpLvl >= 3, and List objects are traversed recursively if DumpLvl >= 4.
TrimCrvs	NumericType	Number of samples the higher order trimmed curves are sampled, in piecewise linear approximation. If zero, computed symbolically as composition.
UVBoolean	NumericType	If TRUE, Boolean between surfaces returns UV instead of Euclidean curves.

Example:

```
IRITSTATE( "DebugFunc", 3 );
IRITSTATE( "FloatFrmt", "%8.5lg" );
```

To print parameters of user defined functions on entry, and return value on exit. Also selects a floating point printf format of "

10.4.18 INTERACT

```
INTERACT( GeometryTreeType Object )
```

A user-defined function (see iritinit.irt) that does the following, in order:

1. Clear the display device.
2. Display the given **Object**.
3. Pause for a keystroke.

This user-defined function in version 4.0 of *IRIT* is an emulation of the INTERACT function that used to exist in previous versions.

Example:

```
INTERACT( list( view_mat, Axes, Obj ) );
```

displays and interacts with the object **Obj** and the predefined object **Axes**. VIEW_MAT will be used to set the starting transformation.

See VIEW and VIEWOBJ for more.

10.4.19 LIST

```
ListType LIST( AnyType Elem1, AnyType Elem2, ... )
```

Constructs an object as a list of several other objects. Only a reference is made to the Elements, so modifying Elem1 after being included in the list will affect Elem1 in that list next time list is used!

Each inclusion of an object in a list increases its internal **used** reference. The object is freed iff in **used** reference is zero. As a result, attempt to delete a variable (using FREE) which is referenced in a list removes the variable, but the object itself is freed only when the list is freed.

10.4.20 LOAD

```
AnyType LOAD( StringType FileName )
```

Loads an object from the given **FileName**. The object may be any object defined in the system, including lists, in which the structure is recovered and reconstructed as well (internal objects are inserted into the global system object list if they have names). If no file type is provided, ".dat" is assumed.

This command can also be used to load binary files. Ascii regular data files are usually loaded in much more time then binary files due the the parsing required. Binary data files can be loaded directly like ascii files in *IRIT*, but can only be inspected through **IRIT** tools such as dat2irit. A binary data file must have a ".bdt" (Binary DaTa) type in its name.

Under unix, compressed files can be loaded if the given file name has a postfix of ".Z". The unix system's "zcat" will be invoked via a pipe for that purpose.

10.4.21 LOGFILE

```
LOGFILE( NumericType Set )
```

or

```
LOGFILE( StringType FileName )
```

If **Set** is non zero (see TRUE/FALSE and ON/OFF), then everything printed in the input window, will go to the log file specified in the IRIT.CFG configuration file. This file will be created the first time logfile is turned ON. If a string **FileName** is provided, it will be used as a log file name from now on. It also closes the current log file. A "LOGFILE(on);" must be issued after a log file name change.

Example:

```
LOGFILE( "Data1" );
LOGFILE( on );
printf( "Resolution = %lf\\n", list( resolution ) );
LOGFILE( off );
```

to print the current resolution level into file Data1.

10.4.22 MSLEEP

```
MSLEEP( NumericType MilliSeconds )
```

Causes the solid modeller to sleep for the prescribed time in milliseconds.

Example:

```
for ( i = 1, 1, sizeof( crvs ),
    c = nth( crvs, i ):
    color( c, yellow ):
    msleep(20):
    viewobj( c )
);
```

Displays an animation sequence and sleeps for 20 milliseconds between iterations.

10.4.23 NTH

```
AnyType NTH( ListType ListObject, NumericType Index )
```

Returns the **Index** (base count 1) element of the list **ListObject**.

Example:

```
Lst = list( a, list( b, c ), d );
Lst2 = NTH( Lst, 2 );
```

and now **Lst2** is equal to 'list(b, c)'.

10.4.24 PAUSE

```
PAUSE( NumericType Flush )
```

Waits for a keystroke. Nice to have if a temporary stop in a middle of an included file (see INCLUDE) is required. If **Flush** is TRUE, then the input is first flushed to guarantee that the actual stop will occur.

10.4.25 PRINTF

```
PRINTF( StringType CtrlStr, ListType Data )
```

A formatted printing routine, following the concepts of the C programming language's *printf* routine. **CtrlStr** is a string object for which the following special '%' commands are supported:

%d, %i, %o, %x, %X	Prints the numeric object as an octal or hexadecimal integer.
%e, %f, %g,	Prints the numeric object in several formats of
%E, %F	floating point numbers.
%s	Prints the string object as a string.
%pe, %pf, %pg	Prints the three coordinates of the point object.
%ve, %vf, %vg	Prints the three coordinates of the vector object.
%Pe, %Pf, %Pg,	Prints the four coordinates of the plane object.
%De, %Df, %Dg,	Prints the given object in IRIT's data file format.

All the '%' commands can include any modifier that is valid in the C programming language printf routine, including l (long), prefix character(s), size, etc. The point, vector, plane, and object commands can also be modified in a similar way, to set the format of the numeric data printed.

Also supported are the newline and tab using the backslash escape character:

```
PRINTF("\\tThis is the char \"\\%\"\\n", nil());
```

Backslashes should be escaped themselves as can be seen in the above example. Here are few more examples:

```
PRINTF("this is a string \"%s\" and this is an integer %8d.\\n",
      list("STRING", 1987));
PRINTF("this is a vector [%8.5lvf]\\n", list(vector(1,2,3)));
IritState("DumpLevel", 9);
PRINTF("this is a object %8.6lDf...\\n", list(axes));
PRINTF("this is a object %10.8lDg...\\n", list(axes));
```

This implementation of PRINTF is somewhat different than the C programming language's version, because the backslash *always* escapes the next character during the processing stage of IRIT's parser. That is, the string

```
'\\tThis is the char \"\\%\"\\n'
```

is actually parsed by the IRIT's parser into

```
'\tThis is the char "\%"\n'
```

because this is the way the IRIT parser processes strings. The latter string is the one that PRINTF actually see.

10.4.26 PROCEDURE

```
ProcName = PROCEDURE(Prm1, Prm2, ... , PrmN):LclVal1:LclVar2: ... :LclVarM:
    ProcBody;
```

A procedure is a function that does not return a value, and therefore the return variable (see FUNCTION) should not be used. A procedure is identical to a function in every other way. See FUNCTION for more.

10.4.27 RMATTR

```
RMATTR( AnyType Object, StringType Name )
```

Removes attribute named **Name** from object **Object**. This function will have no affect on **Object** if **Object** have no attribute named **Name**.
See also ATTRIB.

10.4.28 SAVE

```
SAVE( StringType FileName, AnyType Object )
```

Saves the provided **Object** in the specified file name **FileName**. No extension type is needed (ignored if specified), and ".dat" is supplied by default. **Object** can be any object type, including list, in which structure is saved recursively. See also LOAD. If a display device is actively running at the time SAVE is invoked, its transformation matrix will be saved with the same name but with extension type of ".mat" instead of ".dat".

This command can also be used to save binary files. Ascii regular data files are usually loaded in much more time then binary files due the the parsing required. Binary data files can be loaded directly like ascii files in *IRIT*, but must be inspected through **IRIT** tools such as dat2irit. A binary data file must have a ".bdt" (Binary DaTa) type in its name.

Under unix, files will be saved compressed if the given file name has a postfix of ".Z". The unix system's "compress" will be invoked via a pipe for that purpose.

Example:

```
SAVE( "Obj1.bdt.Z", Obj1 );
```

Saves Obj1 in the file Obj1.bdt.Z as compressed binary file.

10.4.29 SNOC

```
SNOC( AnyType Object, ListType ListObject )
```

Similar to the lisp cons operator but puts the new **Object** in the *end* of the list **ListObject** instead of the beginning, in place.

Example:

```
Lst = list( axes );
SNOC( Srf, Lst );
```

and now **Lst** is equal to the list 'list(axes, Srf)'.

10.4.30 SYSTEM

```
SYSTEM( StringType Command )
```

Executes a system command **Command**. For example,

```
SYSTEM( "ls -l" );
```

10.4.31 TIME

```
TIME( NumericType Reset )
```

Returns the time in seconds from the last time TIME was called with **Reset** TRUE. This time is CPU time if such support is available from the system (times function), and is real time otherwise (time function). The time is automatically reset at the beginning of the execution of this program.

Example:

```
Dummy = TIME( TRUE );
   .
   .
   .
TIME( FALSE );
```

prints the time in seconds between the above two time function calls.

10.4.32 VARLIST

```
VARLIST()
```

List all the currently defined objects in the system.

10.4.33 VECTOR

```
VectorType VECTOR( NumericType X, NumericType Y, NumericType Z )
```

Creates a vector type object, using the three provided NumericType scalars.

10.4.34 VIEW

```
VIEW( GeometricTreeType Object, NumericType ClearWindow )
```

Displays the (geometric) object(s) as given in **Object**.

If **ClearWindow** is non-zero (see TRUE/FALSE and ON/OFF) the window is first cleared (before drawing the objects).

Example:

```
VIEW( Axes, FALSE );
```

displays the predefined object **Axes** in the viewing window on top of what is drawn already.

In version 4.0, this function is emulated (see iritinit.irt) using the VIEWOBJ function. In order to use the current viewing matrix, VIEW_MAT should be provided as an additional parameter. For example,

```
VIEW( list( view_mat, Obj ), TRUE );
```

However, since VIEW is a user defined function, the following will not use VIEW_MAT as one would expect:

```
VIEW( view_mat, TRUE );
```

because VIEW_MAT will be renamed inside the VIEW user defined function to a local (to the user defined function) variable.

In iritinit.irt one can find several other useful VIEW related functions:

VIEWCLEAR	Clears all data displayed on the display device.
VIEWREMOVE	Removes the object specified by name from display.
VIEWDISC	Disconnects from display device (which is still running) while allowing IRIT to connect to a new device.
VIEWEXIT	Forces the display device to exit.
VIEWSAVE	Request sdisplay device to save transformation matrix.
BEEP	An emulation of the BEEP command of versions prior to 4.0.
VIEWSTATE	Allows to change the state of the display device.

For the above VIEW related functions, only VIEWREMOVE, VIEWSAVE, and VIEWSTATE require a parameter, which is the file name and view state respectively. The view state can be one of several commands. See the display device section for more.

Examples:

```
VIEWCLEAR();
VIEW( axes, off );
VIEWSTATE( "LngrVecs" );
VIEWSTATE( "DrawSolid" );
VIEWSAVE( "matrix1" );
VIEWREMOVE( "axes" );
VIEWDISC();
```

10.4.35 VIEWOBJ

```
VIEWOBJ( GeometricTreeType Object )
```

Displays the (geometric) object(s) as given in **Object**. **Object** may be any GeometricType or a list of other GeometricTypes nested to an arbitrary level.

Unlike *IRIT* versions prior to 4.0, VIEW_MAT is not explicitly used as the transformation matrix. In order to display with a VIEW_MAT view, VIEW_MAT should be listed as an argument (in that exact name) to VIEWOBJ. Same is true for the perspective matrix PRSP_MAT.

Example:

```
VIEWOBJ( list( view_mat, Axes ) );
```

displays the predefined object **Axes** in the viewing window using the viewing matrix VIEW_MAT.

10.4.36 WHILE

```
WHILE( NumericType Cond, AnyType Body )
```

Executes the **Body** (see below), while the WHILE loop conditions **Cond** is evaluated into a non zero value. **Cond** is being evaluated before each iteration.

The body may consist of any number of regular commands, separated by COLONs, including nesting loops to an arbitrary level.

Example:

```
deg = 0;
rotstepx = rotx( 10 );
WHILE ( deg < 360,
    deg = deg + 10:
    view_mat = rotstepx * view_mat:
    view( list( view_mat, axes ), ON )
);
```

Displays axes with a view direction that is rotated 10 degrees at a time around the X axis.

10.5 System variables

System variables are predefined objects in the system. Any time *IRIT* is executed, these variable are automatically defined and set to values which are sometimes machine dependent. These are *regular* objects in any other sense, including the ability to delete or overwrite them. One can modify, delete, or introduce other objects using the IRITINIT.IRT file.

10.5.1 AXES

Predefined polyline object (PolylineType) that describes the XYZ axes.

10.5.2 DRAWCTLPT

Predefined Boolean variable (NumericType) that controls whether curves' control polygons and surfaces' control meshes are drawn (TRUE) or not (FALSE). Default is FALSE.

10.5.3 FLAT4PLY

Predefined Boolean object (NumericType) that controls the way almost flat surface patches are converted to polygons: four polygons (TRUE) or only two polygons (FALSE). Default value is FALSE.

10.5.4 MACHINE

Predefined numeric object (NumericType) holding the machine type as one of the following constants: MSDOS, SGI, HP, SUN, APOLLO, UNIX, IBMOS2, IBMNT, and AMIGA.

10.5.5 POLY_APPROX_OPT

A variable controlling the algorithm to convert surfaces to polygons. This two digit number controls the method that is used to subdivide a surface into polygons. The first digit (units) can be one of:

0	Uniform sampling in a fixed grid.
1	An alternated U and V subdivision direction. Once U is subdivided and then V is subdivided.
2	A min max subdivision direction. In other words, the direction that minimizes the maximal error is selected.

The second digit (tenths) can be one of:

0	A fixed sized regular grid. The side of the grid is set via the RESOLUTION variable.
1	This mode is *not* for general use.
2	Maximal distance between the surface and its polygonal approximation is bounded by bilinear surface fit. Maximal distance allowed is set via POLY_APPROX_TOL. *Recommended* choice for optimal polygonization.
3	This mode is *not* for general use.

10.5.6 POLY_APPROX_UV

A Boolean predefined variable. If TRUE, UV values of surface polygonal approximation are placed on attribute lists of vertices.

10.5.7 POLY_APPROX_TOL

A numeric predefined tolerance control on the distance between the surface and its polygonal approximation in POLY_APPROX_OPT settings.

10.5.8 PRSP_MAT

Predefined matrix object (MatrixType) to hold the perspective matrix used/set by VIEW and/or INTERACT commands. See also VIEW_MAT.

10.5.9 RESOLUTION

Predefined numeric object (NumericType) that sets the accuracy of the polygonal primitive geometric objects and the approximation of curves and surfaces. Holds the number of divisions a circle is divided into (with minimum value of 4). If, for example, RESOLUTION is set to 6, then a generated CONE will effectively be a six-sided pyramid. Also controls the fineness of freeform curves and surfaces when they are approximated as piecewise linear polylines, and the fineness of freeform surfaces when they are approximated as polygons.

10.5.10 VIEW_MAT

Predefined matrix object (MatrixType) to hold the viewing matrix used/set by VIEW and/or INTERACT commands. See also PRSP_MAT.

10.6 System constants

The following constants are used by the various functions of the system to signal certain conditions. Internally, they are represented numerically, although, in general, their exact value is unimportant and may be changed in future versions. In the rare circumstance that you need to know their values, simply type the constant as an expression.

Example:

```
MAGENTA;
```

10.6.1 AMIGA

A constant designating an AMIGA system, in the MACHINE variable.

10.6.2 APOLLO

A constant designating an APOLLO system, in the MACHINE variable.

10.6.3 BLACK

A constant defining a BLACK color.

10.6.4 BLUE

A constant defining a BLUE color.

10.6.5 COL

A constant defining the COLumn or U direction of a surface or a trivariate mesh.

10.6.6 CTLPT_TYPE

A constant defining an object of type control point.

10.6.7 CURVE_TYPE

A constant defining an object of type curve.

10.6.8 CYAN

A constant defining a CYAN color.

10.6.9 DEPTH

A constant defining the DEPTH direction of a trivariate mesh. See TBEZIER, TBSPLINE.

10.6.10 E1

A constant defining an E1 (X only coordinate) control point type.

10.6.11 E2

A constant defining an E2 (X and Y coordinates) control point type.

10.6.12 E3

A constant defining an E3 (X, Y and Z coordinates) control point type.

10.6.13 E4

A constant defining an E4 control point type.

10.6.14 E5

A constant defining an E5 control point type.

10.6.15 FALSE

A zero constant. May be used as Boolean operand.

10.6.16 GREEN

A constant defining a GREEN color.

10.6.17 HP

A constant designating an HP system, in the MACHINE variable.

10.6.18 IBMOS2

A constant designating an IBM system running under OS2, in the MACHINE variable.

10.6.19 IBMNT

A constant designating an IBM system running under Windows NT, in the MACHINE variable.

10.6.20 KV_FLOAT

A constant defining a floating end condition uniformly spaced knot vector.

10.6.21 KV_OPEN

A constant defining an open end condition uniformly spaced knot vector.

10.6.22 KV_PERIODIC

A constant defining a periodic end condition with uniformly spaced knot vector.

10.6.23 LIST_TYPE

A constant defining an object of type list.

10.6.24 MAGENTA

A constant defining a MAGENTA color.

10.6.25 MATRIX_TYPE

A constant defining an object of type matrix.

10.6.26 MSDOS

A constant designating an MSDOS system, in the MACHINE variable.

10.6.27 NUMERIC_TYPE

A constant defining an object of type numeric.

10.6.28 OFF

Synonym of FALSE.

10.6.29 ON

Synonym for TRUE.

10.6.30 P1

A constant defining a P1 (W and WX coordinates, in that order) rational control point type.

10.6.31 P2

A constant defining a P2 (W, WX, and WY coordinates, in that order) rational control point type.

10.6.32 P3

A constant defining a P3 (W, WX, WY, and WZ coordinates, in that order) rational control point type.

10.6.33 P4

A constant defining a P4 rational control point type.

10.6.34 P5

A constant defining a P5 rational control point type.

10.6.35 PARAM_CENTRIP

A constant defining a centripetal length parametrization.

10.6.36 PARAM_CHORD

A constant defining a chord length parametrization.

10.6.37 PARAM_UNIFORM

A constant defining an uniform parametrization.

10.6.38 PI

The constant of 3.141592...

10.6.39 PLANE_TYPE

A constant defining an object of type plane.

10.6.40 POINT_TYPE

A constant defining an object of type point.

10.6.41 POLY_TYPE

A constant defining an object of type poly.

10.6.42 RED

A constant defining a RED color.

10.6.43 ROW

A constant defining the ROW or V direction of a surface or a trivariate mesh.

10.6.44 SGI

A constant designating an SGI system, in the MACHINE variable.

10.6.45 STRING_TYPE

A constant defining an object of type string.

10.6.46 SURFACE_TYPE

A constant defining an object of type surface.

10.6.47 SUN

A constant designating a SUN system, in the MACHINE variable.

10.6.48 TRIMSRF_TYPE

A constant defining an object of type trimmed surface.

10.6.49 TRIVAR_TYPE

A constant defining an object of type trivariate function.

10.6.50 TRUE

A non zero constant. May be used as Boolean operand.

10.6.51 UNDEF_TYPE

A constant defining an object of no type (yet).

10.6.52 UNIX

A constant designating a generic UNIX system, in the MACHINE variable.

10.6.53 VECTOR_TYPE

A constant defining an object of type vector.

10.6.54 WHITE

A constant defining a WHITE color.

10.6.55 YELLOW

A constant defining a YELLOW color.

11 Animation

The animation tool adds the capability of animating objects using forward kinematics, exploiting animation curves. Each object has different attributes, that prescribe its motion, scale, and visibility as a function of time. Every attribute has a name, which designates it's role. For instance an attribute animation curve named MOV_X describes a translation motion along the X axis.

11.1 How to create animation curves in IRIT

Let OBJ be an object in IRIT to animate.

Animation curves are either scalar (E1/P1) curves or three dimensional (E3/P3) curves with one of the following names:

MOV_X, MOV_Y, MOV_Z	Translation along one axis
MOV_XYZ	Arbitrary translation along all three axes
ROT_X, ROT_Y, ROT_Z	Rotating around a single axis (degrees)
SCL_X, SCL_Y, SCL_Z	Scale along a single axis
SCL	Global scale
VISIBLE	Visibility

The visibility curve is a scalar curve that enables the display of the object if the visibility curve is positive at time t and disables the display (hide) the object if the visibility curve is negative at time t.

The animation curves are all attached as an attribute named "animation" to the object OBJ.

Example:

```
mov_x = cbezier( ctlpt( E1, 0.0 ),
                 ctlpt( E1, 1.0 ) );
scl   = cbezier( ctlpt( E1, 1.0 ),
                 ctlpt( E1, 0.1 ) );
rot_y = cbezier( ctlpt( E1, 0.0 ),
                 ctlpt( E1, 0.0 ) );
                 ctlpt( E1, 360.0 ) );
attrib(OBJ, "animation", list( mov_x, scl, rot_y ) );
```

To animate OBJ between time zero and one (Bezier curves are always between zero and one), by moving it a unit size in the X direction, scaling it to angular speed from zero to 360 degrees.

OBJ can now be save into a file or displayed via one of the regular viewing commands in IRIT (i.e. VIEWOBJ).

Animation is not always between zero and one. To that end one can apply the CREPARAM function to modify the parametric domain of the animation curve. The convention is that if the time is below the starting value of the parametric domain, the starting value of the curve is used. Similarly if the time is beyond the end of the parameter domain of the animation curve, the end value of the animation curve is used.

Example:

```
CREPARAM( mov_x, 3.0, 5.0 );
```

to set the time of the motion in the x axis to be from $t = 3$ to $t = 5$. for $t < 3$, mov_x(3) is used, and for $t > 5$, mov_x(5) is employed.

the animation curves are regular objects in the IRIT system. Hence, only one object named mov_x or scl can exist at one time. If you create a new object named mov_x, the old one is overwritten! To preserve old animation curves you can detach the old ones by executing 'free(mov_x)' that removes the object named mov_x from IRIT's object list but not from its previous used locations within other list objects, if any. For example:

```
mov_x = cbezier( ctlpt( E1, 0.0 ),
                 ctlpt( E1, 1.0 ) );
attrib(obj1, "animation", list( mov_x ) );
free(mov_x);

mov_x = cbezier( ctlpt( E1, 2.0 ),
                 ctlpt( E1, 3.0 ) );
attrib(obj2, "animation", list( mov_x ) );
free(mov_x);
```

11.2 A more complete animation example

```
a = box( vector( 0, 0, 0 ), 1, 1, 1 );
b = box( vector( 0, 0, 0 ), 1, 1, 1 );
c = box( vector( 0, 0, 0 ), 1, 1, 1 );
d = sphere( vector( 0, 0, 0), 0.7 );
```

```
pt0   =  ctlpt( e1,  0.0 );
pt1   =  ctlpt( e1,  1.0 );
pt2   =  ctlpt( e1,  2.0 );
pt6   =  ctlpt( e1,  6.0 );
pt360 =  ctlpt( e1,  360.0 );

pt10 = ctlpt( e1, -4.0 );
pt11 = ctlpt( e1,  1.0 );
pt12 = ctlpt( e1,  4.0 );
pt13 = ctlpt( e1, -1.0 );

visible = creparam( cbezier( list( pt10,  pt11 ) ), 0.0, 5.0 );
mov_x   = creparam( cbezier( list( pt0, pt6, pt2 ) ), 0.0, 1.2 );
mov_y   = mov_x;
mov_z   = mov_x;
rot_x   = creparam( cbspline( 2,
                              list( pt0, pt360, pt0 ),
                              list( KV_OPEN ) ),
                    1.2, 2.5 );
rot_y   = rot_x;
rot_z   = rot_x;
scl     = creparam( cbezier( list( pt1, pt2, pt1, pt2, pt1 ) ),
                    2.5, 4.0 );
scl_x   = scl;
scl_y   = scl;
scl_z   = scl;
mov_xyz = creparam( circle( vector( 0, 0, 0 ), 2.0 ), 4.0, 5.0 );

attrib( d, "animation", list( mov_xyz, visible ) );
free( visible );

visible = creparam( cbezier( list( pt12,  pt13 ) ), 0.0, 5.0 );

attrib( a, "animation", list( rot_x, mov_x, scl, scl_x, visible ) );
attrib( b, "animation", list( rot_y, mov_y, scl, scl_y, visible ) );
attrib( c, "animation", list( rot_z, mov_z, scl, scl_z, visible ) );

color( a, red );
color( b, green );
color( c, blue );
color( d, cyan );

demo = list( a, b, c, d );

interact( demo );
viewanim( 0, 5, 0.01 );
```

In this example, we create four objects, three cubes and one sphere. Animation curves to translate the three cubes along the three axes for the time period of t = 0 to t = 1.2 are created. Rotation curves to rotate the three cubes along the three axes are then created for time period of t = 1.2 to t = 2.5. Finally, for the time period of t = 2.5 to t = 4.0. the cubes are (not only) unifomly scaled. For the time period of t = 4 to t = 5, the cubes become invisible and the sphere, that becomes visible, is rotated along a circle of radius 2.

12 Display devices

The following display device drivers are available,

Device Name	Invocation	Environment
xgldrvs	xgldrvs -s-	SGI 4D GL regular driver.
xogldrvs	xogldrvs -s-	SGI 4D Open GL/Motif driver.
xgladap	xgladap -s-	SGI 4D GL adaptive isocurve experimental driver.
x11drvs	x11drvs -s-	X11 driver.
xmtdrvs	xmtdrvs -s-	X11 Motif driver.
xglmdrvs	xglmdrvs -s-	SGI 4D GL and X11/Motif driver.
wntdrvs	wntdrvs -s-	IBM PC Windows NT driver.
wntgdrvs	wntgdrvs -s-	IBM PC Windows NT Open GL driver.
os2drvs	os2drvs -s-	IBM PC OS2 2.x driver.
amidrvs	amidrvs -s-	AmigaDOS 2.04+ driver.
nuldrvs	nuldrvs -s- [-d] [-D]	A device to print the object stream to stdout.

All display devices are clients communicating with the server (*IRIT*) using IPC (inter process communication). On Unix and Window NT sockets are used. A Windows NT client can talk to a server (*IRIT*) on a unix host if hooked to the same netwrok. On OS2 pipes are used, and both the client and server must run on the same machine. On AmigaDOS exec messages are used, and both the client and server must run on the same machine.

While all display devices support object(s) transformations via a transformation control window, many of the display devices allow one to click and drag on the viewing window to rotate (Left Button) and to translate (Right Button). This mode exploits the two degrees of freedom of the mouse to provide intuitive dual axis rotation and translation.

The server (*IRIT*) will automatically start a client display device if the IRIT_DISPLAY environment variable is set to the name and options of the display device to run. For example:

```
setenv IRIT_DISPLAY xgldrvs -s-
```

The display device must be in a directory that is in the **path** environment variable. Most display devices require the '-s-' flags to run in a non-standalone mode, or a client-server mode. Most drivers can also be used to display data in a standalone mode (i.e., no server). For example:

```
xgldrvs -s solid1.dat irit.mat
```

Effectively, all the display devices are also data display programs (poly3d, which was the display program prior to version 4.0, is retired in 4.0). Therefore some functionality is not always as expected. For example, the quit button will always force the display device to quit, even if poped up from *IRIT*, but will not cause *IRIT* to quit as might logically expected. In fact, the next time *IRIT* will try to communicate with the display device, it will find the broken connection and will start up a new display device.

Most display devices recognize attributes found on objects. The following attributes are usually recognized (depending on the device capability.):

- **COLOR**: Selects the drawn color of the object to be one of the 8/16 predefined colors in the *IRIT* system: white, red, green, blue, yellow, cyan, magenta, black. **RGB**: Overwrites (if supported) the COLOR attribute (if given) and sets the color of the object to the exact prescribed RGB set. **DWIDTH**: Sets the width in pixels of the drawn object, when drawn as a wireframe.

All display devices recognize all the command line flags and all the configuration options in a configuration file, as described below. The display devices will make attemps to honor the requests, to the best of their ability. For example, only xgldrvs can render shaded models, and so only it will honor a **DrawSolid** configuration options.

12.1 Command Line Options

```
???drvs [-s] [-u] [-n] [-N] [-i] [-c] [-C] [-m] [-a] [-g x1,x2,y1,y2]
        [-G x1,x2,y1,y2] [-I #IsoLines] [-F PolygonOpti FineNess]
        [-f PolylineOpti SampPerCrv] [-l LineWidth] [-r] [-A Shader] [-B]
        [-2] [-d] [-D] [-L NormalLen] [-4] [-b BackGround] [-S LgtSrcPos]
        [-Z ZMin ZMax] [-M] [-P] [-x ExecAnim] [-z] DFiles
```

- **-s**: Runs the driver in a Standalone mode. Otherwise, the driver will attempt to communicate with the *IRIT* server.
- **-u**: Forces a Unit matrix. That is, input data are *not* transformed at all.
- **-n**: Draws normals of vertices.
- **-N**: Draws normals of polygons.
- **-i**: Draws internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-c**: Sets depth cueing on. Drawings that are closer to the viewer will be drawn in more intense color.
- **-C**: Cache the piecewise linear geometry so curves and surface can be redisplay faster. Purging it will free memory, on the other hand.
- **-m**: Provides some more information on the data file(s) parsed.
- **-a**: Sets the support of antialiased lines.
- **-g x1,x2,y1,y2**: Prescribes the position and location of the transformation window by prescribing the domain of the window in screen space pixels.
- **-G x1,x2,y1,y2**: Prescribes the position and location of the viewing window by prescribing the domain of the window in screen space pixels.
- **-I #IsoLines**: Specifies number of isolines per surface, per direction. A specification of zero isolines is possible only on the command line and it denotes the obvious.
- **-F PolyOpti FineNess**: Controls the method used to approximate surfaces into polygons. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.

- **-l LineWidth**: Sets the linewidth, in pixels. Default is one pixel wide.
- **-r**: Rendered mode. Draws object as solid.
- **-A Shader**: Shader can be one of 1 (Flat), 2 (Gouraud), or 3 (Phong).
- **-B**: Back face culling of polygons.
- **-2**: Double buffering. Prevents screen flicker on the expense of possibly less colors.
- **-d**: Debug objects. Prints to stderr all objects read from communcation port with the server *IRIT*.
- **-D**: Debug input. Prints to stderr all characters read from communcation port with the server *IRIT*. Lowest level of communication.
- **-L NormalLen**: Sets the length of the drawn normals in thousandths of a unit.
- **-4**: Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only.
- **-b BackGround**: Sets the background color as three RGB integers in the range of 0 to 255.
- **-S LgtSrcPos**: Sets the lighting via the light source position.
- **-Z ZMin ZMax**: Sets the near and far Z clipping planes.
- **-M**: Draw control mesh/polygon of curves and surfaces, as well.
- **-x ExecAnim**: Command to execute as a subprocess every iteration of display of an animation sequence. This command can for example save the display into an image file, saving the animation sequence. One parameter is passed, which is an running index starting from one.
- **-P**: Draws curves and surfaces (surfaces are drawn using a set of isocurves, see -I, or polygons, see -f).
- **-z**: Prints version number and current defaults.

12.2 Configuration Options

The configuration file is read *before* the command line options are processed. Therefore, all options in this section can be overriden by the appropriate command line option, if any.

- **TransPrefPos:** Preferred location (Xmin, YMin, Xmax, Ymax) of the transformation window.
- **ViewPrefPos:** Preferred location (Xmin, YMin, Xmax, Ymax) of the viewing window.
- **BackGround:** Background color. Same as '-b'.
- **Internal:** Draws internal edges. Same as '-i'.
- **LightSrcPos:** Sets the location of the (first) light source as a rational four coefficient location. W of zero sets the light source at infinity.
- **ExecAnim:** Executes animation. Same as '-x'.
- **DrawVNormal:** Draws normals of vertices. Same as '-n'.
- **DrawPNormal:** Draws normals of polygons. Same as '-n'.

- **MoreVerbose:** Provides some more information on the data file(s) parsed. Same as '-m'.
- **UnitMatrix:** Forces a Unit matrix. That is, input data are *not* transformed at all. Same as '-u'.
- **DrawSolid:** Requests a shaded surface rendering, as opposed to isocurve surface representation.
- **BFaceCull:** Requests the removal of back facing polygons, for better visibility.
- **DoubleBuffer:** Requests drawing using a double buffer, if any.
- **DebugObjects:** Debug objects. Prints to stderr all objects read from the communcation port with the server *IRIT*. Same as '-d'.
- **DebugEchoInput:** Debug input. Prints to stderr all characters read from the communcation port with the server *IRIT*. Lowest level of communications.
- **DepthCue:** Set depth cueing on. Drawings that are closer to the viewer will be drawn in more intense color. Same as '-c'.
- **CacheGeom:** Normally piecewise linear approximation of freefroms is cached. By setting this option to FALSE, no such auxiliary data is being saved, reducing the memory overhead. Same as '-C'.
- **FourPerFlat:** Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only. Same as '-4'.
- **AntiAlias:** Request the drawing of anti aliased lines.
- **DrawSurfaceMesh:** Draws control mesh/polygon of curves and surfaces, as well. Same as '-M'.
- **DrawSurfacePoly:** Draws curves and surfaces (surfaces are drawn using a set of isocurves, see -I, or polygons, see -f). Same as '-P'.
- **StandAlone:** Runs the driver in a Stand alone mode. Otherwise, the driver will attempt to communicate with the *IRIT* server. Same as '-s'.
- **NumOfIsolines:** Specifies number of isolines per surface, per direction. Same as '-I'.
- **SamplesPerCurve:** Specifies the samples per (iso)curve. See '-f'.
- **LineWidth:** Sets the linewidth, in pixels. Default is one pixel wide. Same as '-l'
- **AdapIsoDir:** Selects the direction of the adaptive isoline rendering.
- **PolygonOpti:** Controls the method used to subdivide a surface into polygons that approximate it. Same as '-F'.
- **PolylineOpti:** Controls the method used to subdivide a curve into polylines that approximate it. Same as '-f'.
- **ShadingModel:** One of 1 (Flat), 2 (Gouraud), or 3 (Phong). Same as '-A'.
- **TransMode:** Selects between object space transformations and screen space transformation.
- **ViewMode:** Selects between perspective and orthographic views.
- **NormalLength:** Sets the length of the drawn normals in thousandths of a unit. Same as '-L'.

- **ZClipMin:** Sets the minimal clipping plane in Z. Same as '-Z'.
- **ZClipMax:** Sets the maximal clipping plane in Z. Same as '-Z'.
- **FineNess:** Controls the fineness of the surface to polygon subdivision. See '-F'.

12.3 Interactive mode setup

Commands that affect the status of the display device can also be sent via the communication port with the *IRIT* server. The following commands are recognized as string objects with object name of "COMMAND_":

BEEP	Makes some sound.
CLEAR	Clears the display area. All objects are deleted.
DCLEAR	Delayed clear. Same as CLEAR but delayed until next object is sent from the server. Useful for animation.
DISCONNECT	Closes connection with the server, but does not quit.
EXIT	Closes connection with the server and quits.
GETOBJ NAME	Requests the object named NAME that is returned in the output channel to the server.
MSAVE NAME	Saves the transformation matrix file by the name NAME.
REMOVE NAME	Request the removal of object named NAME from display.
ANIMATE TMin TMax Dt	Animates current scene from TMin to TMax in Dt steps.
STATE COMMAND	Changes the state of the display device. See below.

The following commands are valid for the STATE COMMAND above,

MoreSense:	More sensitive mouse control.
LessSense:	Less sensitive mouse control.
ScrnObject:	Toggle screen/object transformation mode.
PerspOrtho:	Toggles perspective/orthographic trans. mode.
DepthCue:	Toggles depth cueing drawing.
DrawSolid:	Toggles isocurve/shaded solid drawing.
ShadingMdl:	Toggles shading model for solid solid drawing.
BFaceCull:	Cull back facing polygons.
DblBuffer:	Toggles single/double buffer mode.
AntiAlias:	Toggles anti aliased lines.
DrawIntrnl:	Toggles drawing of internal lines.
DrawVNrml:	Toggles drawing of normals of vertices.
DrawPNrml:	Toggles drawing of normals of polygons.
DSrfMesh:	Toggles drawing of control meshes/polygons.
DSrfPoly:	Toggles drawing of curves/surfaces.
4PerFlat:	Toggles 2/4 polygons per flat surface regions.
MoreIso:	Doubles the number of isolines in a surface.
LessIso:	Halves the number of isolines in a surface.
FinrAprx:	Doubles the number of samples per curve.
CrsrAprx:	Halves the number of samples per curve.
LngrVecs:	Doubles the length of displayed normal vectors.
ShrtrVecs:	Halves the length of displayed normal vectors.
WiderLns:	Doubles the width of the drawn lines.
NarrwLns:	Halves the width of the drawn lines.
WiderPts:	Doubles the width of the cross of drawn points.
NarrwPts:	Halves the width of the cross of drawn points.
FinrAdapIso:	Doubles the number of adaptive isocurves.
CrsrAdapIso:	Halves the number of adaptive isocurves.
FinerRld:	Doubles number of ruled surfaces in adaptive isocurves.
CrsrRld:	Halves number of ruled surfaces in adaptive isocurves.
RuledSrfApx:	Toggles ruled surface approx. in adaptive isocurves.
AdapIsoDir:	Toggles the row/col direction of adaptive isocurves.
Front:	Selects a front view.
Side:	Selects a side view.
Top:	Selects a top view.
Isometry:	Selects an isometric view.
Clear:	Clears the viewing area.

Obviously not all state options are valid for all drivers. The *IRIT* server defines in iritinit.irt several user-defined functions that exercise some of the above state commands, such as VIEWSTATE and VIEWSAVE.

In addition to state modification via communication with the *IRIT* server, modes can be interactively modified on most of the display devices using a pop-up menu that is activated using the *right button in the transformation window*. This pop up menu is somewhat different in different drivers, but its entries closely follow the entries of the above state command table.

12.4 Animation Mode

All the display drivers are now able to animate objects with animation curves' attributes on them. For more on the way animation curves can be created see the Animation Section of this manual (Section 11).

Once a scene with animation curves' attributes is being loaded into a display device, one can enter "animation" mode using the "Animation" button available in all display devices. The user is then prompt (either graphically or in a textual based interface) for the starting time, termination time and step size of the animation. The parameter space of the animation curve is serving as the time domain. The default staring and terminating times are set as the minimal and maximal parametric domain values of all animation curves. An object at time t below the minimal parametric value will be placed at the starting value of the animation curve. Similarly, an object at time t above the maximal parametric value will be placed at the termination value of the animation curve. The user can also set a bouncing back and forth mode, the number of repetitions, and if desired, request the saving of all the different scenes in the animation as seperate files so a high quality animation can be created.

12.5 Specific Comments

- The x11drvs supports the following X Defaults (searched at /.Xdefaults):

```
#ifndef COLOR
irit*MaxColors:                    1
irit*Trans*BackGround:             Black
irit*Trans*BorderColor:            White
irit*Trans*TextColor:              White
irit*Trans*SubWin*BackGround:      Black
irit*Trans*SubWin*BorderColor:     White
irit*Trans*CursorColor:            White
irit*View*BackGround:              Black
irit*View*BorderColor:             White
irit*View*CursorColor:             White
#else
irit*MaxColors:                    15
irit*Trans*BackGround:             NavyBlue
irit*Trans*BorderColor:            Red
irit*Trans*TextColor:              Yellow
irit*Trans*SubWin*BackGround:      DarkGreen
irit*Trans*SubWin*BorderColor:     Magenta
irit*Trans*CursorColor:            Green
irit*View*BackGround:              NavyBlue
irit*View*BorderColor:             Red
irit*View*CursorColor:             Red
#endif
irit*Trans*BorderWidth:            3
irit*Trans*Geometry:               =150x500+510+0
irit*View*BorderWidth:             3
irit*View*Geometry:                =500x500+0+0
```

- The Motif-based display drivers contain three types of gadgets which can be operated in the following manner. Scales: can be dragged or clicked outside button for single (mouse's middle button) or continuous (mouse's left button) action. Pushbuttons: activated by clicking the mouse's left button. The control panel: allowes rotation, translation of the objects in three axes, determine perspective ratio, viewing object from top, side, front or isometrically, determining scale factor and clipping settings, and operate the matrix stack.

  The environment window toggles between screen or object transformation, depth cue on or off, orthographic or perspective projection, wireframe or solid display, single or double buffering, showing or hiding normals, including or excluding the surface;s mesh and curve;s control polygon, surface drawing using isolines or polygons, and four or two polygons per flat patch. Same display devices allow for the inclusion or exclusion of internal edges, and enable or disable of antialiased lines. Scales in the X11/Motif based devices set normals length, lines width, control sensitivity, the number of islolines and samples, etc.

- The locations of windows as set via [-g] and [-G] and/or via the configuration file overwrites in x11drvs the Geometry X11 defaults. To use the Geometry X11 default use '-G " "' and '-g " "' or set the string to empty size in the configuration file.
- In os2drvs, only -G is used to specify the dimensions of the parent window that holds both the viewing and the transformation window.
- In os2drvs, the following key strokes are available as short cuts:

Key	Function
^x	Quit
^s	Save
^f	Front View
^d	Side View
^t	Top View
^i	Isometric VIew
^p	Perspetive/Orthographic
^n	View Internal Edges
^v	View Vertices' Normals
^g	View Polygons' Normals
^b	Backface Culling
^c	Depth Cue
^m	View Control Mesh/Poly

13 Utilities - General Usage

The *IRIT* solid modeler is accompanied by quite a few utilities. They can be subdivided into two major groups. The first includes auxiliary tools such as illustrt and poly3d-h. The second includes filters such as irit2ray and irit2ps.

All these tools operate on input files, and most of the time produce data files. In all utilities that read files, the dash ('-') can be used to read stdin.

Example:

```
poly3d-h solid1.dat | irit2ps - > solid1.ps
```

All the utilities have command line options. If an option is set by a '-x' then '-x-' resets the option. The command line options overwrite the settings in config files, and the reset option is useful for cases where the option is set by default, in the configuration file.

All utilities can read a sequence of data files. However, the *last* transformation matrices found (VIEW_MAT and PRSP_MAT) are actually used.

Example:

```
poly3d-h solid1.dat | x11drvs solid1.dat - solid1.mat
```

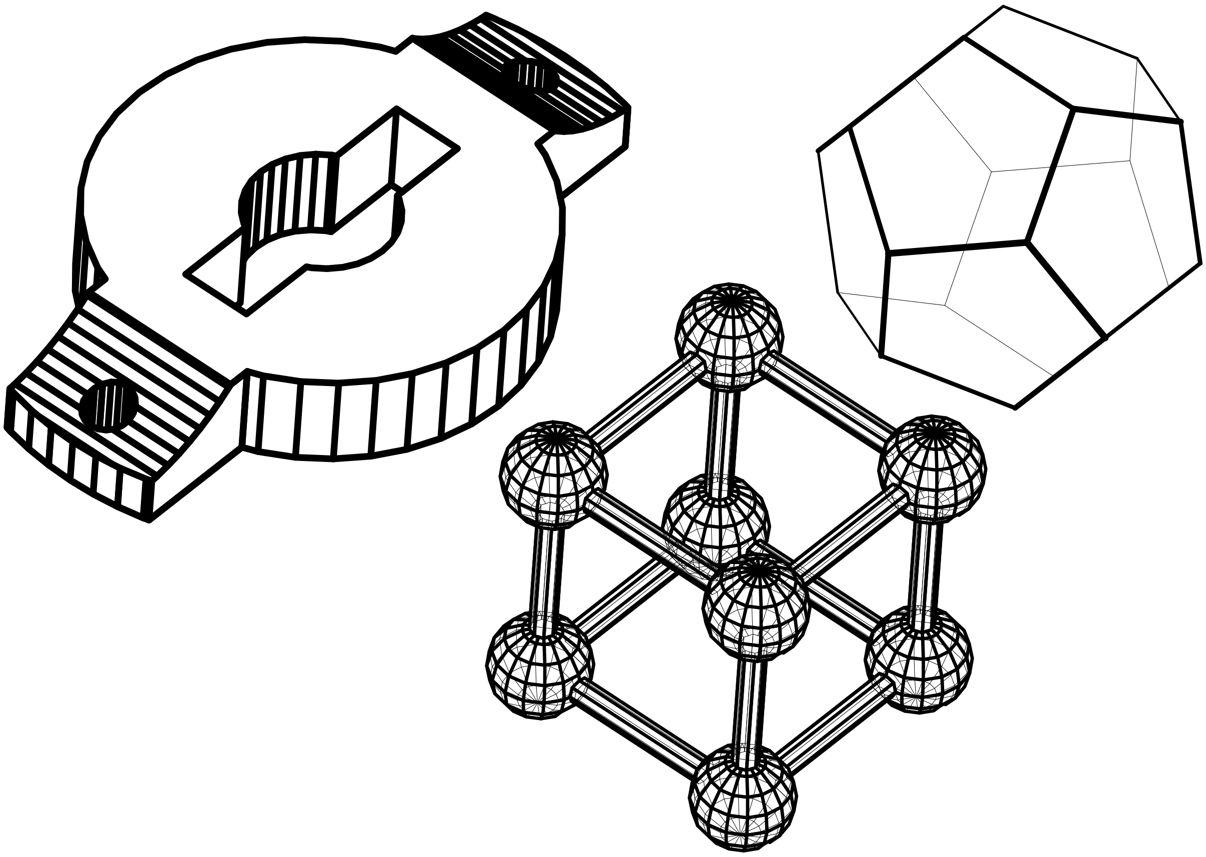


Figure 56: Some examples of the use of hidden line removal tool poly3d-h to remove hidden lines.

x11drvs will display the original solid1.dat file with its hidden version, as computed by poly3d-h, all with the solid1.mat, ignoring all other matrices in the data stream.

Under unix, compressed files with a postfix ".Z" will be *automatically* uncompressed on read and write. The following is legal under unix,

```
poly3d-h solid1.dat.Z | x11drvs solid1.dat.Z - solid1.mat
```

where solid1.dat.Z was saved from within IRIT using the command

```
save( "solid1.dat.Z", solid1 );
```

or similar. The unix system's "compress" and "zcat" are used for the purpose of (un)compressing the data via pipes. See also SAVE and LOAD.

14 Poly3d-h - Hidden Line Removing Program

14.1 Introduction

poly3d-h is a program to remove hidden lines from a given polygonal model. Freeform objects are preprocessed into polygons with controlled fineness. See Figure 56 for some output examples of using this tool.

The program performs 4 passes over the input:

1. Preprocesses and maps all polygons in a scene, and sorts them.

2. Generates edges out of the polygonal model and sorts them (preprocessing for the scan line algorithm) into buckets.

3. Intersects edges, and splits edges with non-homogeneous visibility (the scan line algorithm).

4. Applies a visibility test of each edge.

This program can handle CONVEX polygons only. From *IRIT* one can ensure that a model consists of convex polygons only using the CONVEX command:

```
CnvxObj = convex( Obj );
```

just before saving it into a file. Surfaces are always decomposed into triangles.

poly3d-h output is in the form of polylines. It is a regular *IRIT* data file that can be viewed using any of the display devices, for example.

14.2 Command Line Options

```
poly3d-h [-b] [-m] [-i] [-e #Edges] [-H] [-4] [-W Width]
       [-F PolyOpti FineNess] [-q] [-o OutName] [-c] [-z] DFiles > OutFile
```

- **-b**: BackFacing - if an object is closed (such as most models created by *IRIT*), back facing polygons can be deleted, therefore speeding up the process by at least a factor of two.
- **-m**: More - provides some more information on the data file(s) parsed.
- **-i**: Internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-e n**: Number of edges to use from each given polygon (default all). Handy as '-e 1 -4' for freeform data.
- **-H**: Dumps both visible lines and hidden lines as separated objects. Hidden lines will be dumped using a different (dimmer) color and (a narrower) line width.
- **-4**: Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only.
- **-W Width**: Selects a default width for visible lines in inches.
- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-q**: Quiet mode. No printing aside from fatal errors. Disables -m.
- **-o OutName**: Name of output file. Default is stdout.
- **-z**: Prints version number and current defaults.
- **-c**: Clips data to screen (default). If disabled ('-c-'), data outside the view screen ([-1, 1] in x and y) are also processed.

Some of the options may be turned on in poly3d-h.cfg. They can be then turned off in the command line as '-?-'.

14.3 Configuration

The program can be configured using a configuration file named poly3d-h.cfg. This is a plain ASCII file you can edit directly and set the parameters according to the comments there. 'poly3d-h -z' will display the current configuration as read from the configuration file.

The configuration file is searched in the directory specified by the IRIT_PATH environment variable. For example, 'setenv IRIT_PATH /u/gershon/irit/bin/'. If the IRIT_PATH variable is not set, the current directory is searched.

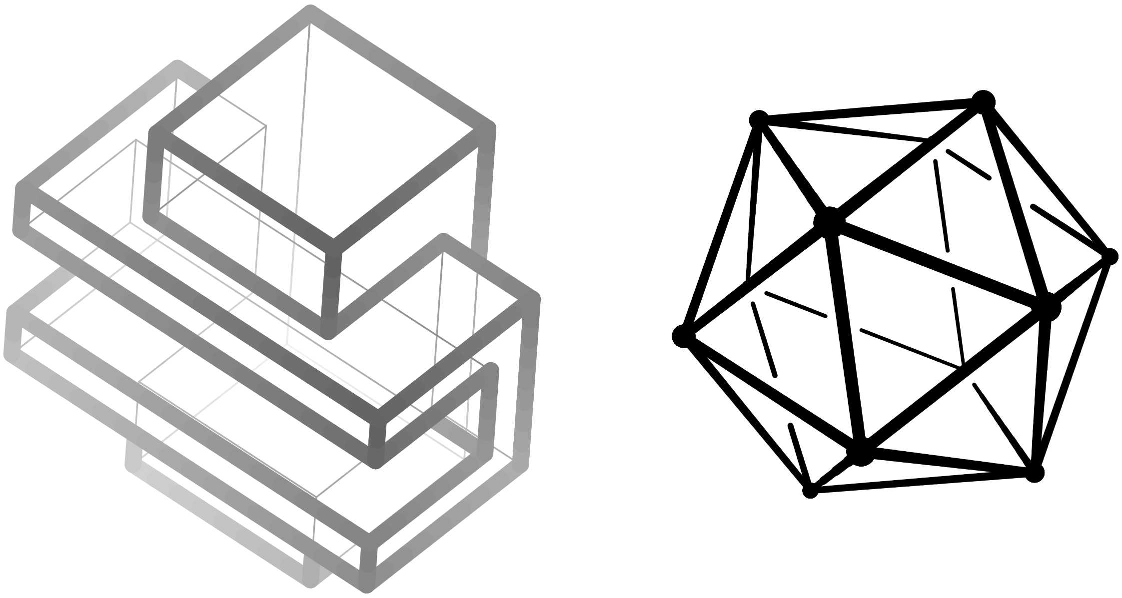


Figure 57: Some examples of the use of the illustration tool illustrt.

14.4 Usage

As this program is not interactive, usage is quite simple, and the only control available is using the command line options.

The images in Figure 56 were created using the following commands:

```
poly3d-h -W 0.01 -H -q molecule.dat view1.dat | irit2ps - > molecule.ps
poly3d-h -W 0.02 -q solid2h.dat view2.dat | irit2ps - > solid2h.ps
poly3d-h -W 0.02 -H -q dodechdr.dat view3.dat |
                                  irit2ps -d -0.59 0.59 - > dodechdr.ps
```

If a certain surface should be polygonized into a finer/caurser set of polygons than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. Further, "u_resolution" and "v_resolution" might be similarly used to set relative resolution for the u or v direction only.

15 Poly3d-r - A Simple Data Rendering Program

Retired. Sources can be found in the contrib directory, but this program is no longer supported. See irender program instead.

16 Illustrt - Simple line illustration filter

16.1 Introduction

illustrt is a filter that processes IRIT data files and dumps out modified IRIT data files. illustrt can be used to make simple nice illustrations of data. The features of illustrt include depth sorting, hidden line clipping at intersection points, and vertex enhancements. illustrt is designed to closely interact with irit2ps, although it is not neceessary to use irit2ps on illustrt output.

See Figure 57 for some output examples of using this tool.

16.2 Command Line Options

```
illustrt [-I #IsoLines] [-S #SampPerCrv] [-s] [-M] [-P] [-p]
         [-l MaxLnLen] [-a] [-t TrimInter] [-o OutName] [-Z InterSameZ]
         [-m] [-z] DFiles
```

- **-I #IsoLines**: Specifies number of isolines per surface, per direction.
- **-S #SampPerCrv**: Specifies the samples per (iso)curve.
- **-s**: sorts the data in Z depth order that emulates hidden line removal once the data are drawn.
- **-M**: Dumps the control mesh/polygon as well.
- **-P**: Dumps the curve/surface as isocurves.
- **-p**: Dumps vertices of polygons/lines as points.
- **-l MaxLnLen**: breaks long lines into shorter ones with maximal length of MaxLnLen. This option is necessary to achieve good depth depending line width in the '-d' option of irit2ps.
- **-a**: takes into account the angle between the two (poly)lines that intersect when computing how much to trim. See also -t.
- **-t TrimInter**: Each time two (poly)line segments intersect in the projection plane, the (poly)line that is farther away from the viewer is clipped TrimInter amount from both sides. See also -a.
- **-o OutName**: Name of output file. Default is stdout.
- **-Z InterSameZ**: The maximal Z depth difference of intersection curves to be be considered invalid.
- **-m**: More talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

16.3 Usage

illustrt is a simple line illustration tool. It process geometry such as polylines and surfaces and dumps geometry with attributes that will make nice line illustrations. illustrt is geared mainly toward its use with irit2ps to create postscript illustrations. Here is a simple example:

```
illustrt -s -l 0.1 solid1.dat | irit2ps -W 0.05 -d 0.2 0.6 -u - > solid.ps
```

make sure all segments piped into irit2ps are shorter than 0.1 and sort them in order to make sure hidden surface removal is correctly applied. Irit2ps is invoked with depth cueing activated, and a default width of 0.05.

illustrt dumps out regular *IRIT* data files, so output can be handled like any other data set. illustrt does the following processing to the input data set:

- Converts surfaces to isocurves ('-I' flag) and isocurves and curves to polylines ('-S' flag), and converts polygons to polylines. Polygonal objects are considered closed and even though each edge is shared by two polygons, only a single one is generated.
- Finds the intersection location in the projection plane of all segments in the input data set and trims away the far segment at both sides of the intersection point by an amount controlled by the '-t' and '-a' flags.

- Breaks polylines and long lines into short segments, as specified via the '-l' flag, so that width depth cueing can be applied more accurately (see irit2ps's '-d' flag) as well as the Z sorting.
- Generates vertices of polygons in the input data set as points in output data controlled via the '-p' flag. set.
- Applies a Z sort to the output data, if '-s', so drawing in order of the data will produce a properly hidden surface removal drawing.

Here is a more complex example. Make sure tubular is properly set via "attrib(solid1, "tubular", 0.7);" and invoke:

```
illustrt -s -p -l 0.1 -t 0.05 solid1.dat |
    irit2ps -W 0.05 -d 0.2 0.6 -p h 0.05 -u - > solid.ps
```

makes sure all segments piped into irit2ps are shorter than 0.1, generates points for the vertices, sorts the data in order to make sure hidden surface removal is correctly applied, and trims the far edge by 0.05 at an intersection point. Irit2ps is invoked with depth cueing activated and a default width of 0.05, points are drawn as hollowed circles of default size 0.05, and lines are drawn tubular.

Objects in the input stream that have an attribute by the name of "IllustrtNoProcess" are passed to the output unmodified. Objects in the input stream that have an attribute by the name of "SpeedWave" will have a linear segments added that emulate fast motion with the following attributes,

`"Randomness,DirX,DirY,DirZ,Len,Dist,LenRandom,DistRandom,Width".`

Objects in the input stream that have an attribute by the name of "HeatWave" will have a spiral curves added that emulate a heat wave in the +Z axis with the following attributes,

`"Randomness,Len,Dist,LenRandom,DistRandom,Width".`

Examples:

```
attrib(Axis, "IllustrtNoProcess", "");
attrib(Obj, "SpeedWave", "0.0005,1,0,0,5,3,3,2,0.05");
attrib(Obj, "HeatWave", "0.015,0.1,0.03,0.06,0.03,0.002");
```

17 Irender - Simple Scan Line Renderer

17.1 Introduction

irender is a program to render *IRIT* scenes into images. It is a software based Z buffer that is able to create images in few formats. Several of its features includes parametric and volumetric texture mapping, shadow computations, transparency and antialiasing.

Freeform objects are preprocessed into polygons with controlled fineness. See Figure 58 for some output examples of using this tool.

17.2 Command Line Options

```
 irender [-z] [-v] [-s XSize YSize] [-a Ambient] [-b R G B] [-B]
         [-F PolyOpti FineNess] [-f PolyOpti SampPerCrv]
         [-M Flat/Gouraud/Phong] [-P WMin [WMax]] [-S] [-T]
         [-A FilterName] [-Z] [-n] [-i rle/ppm] files
```

- **-z**: Prints version number and current defaults.

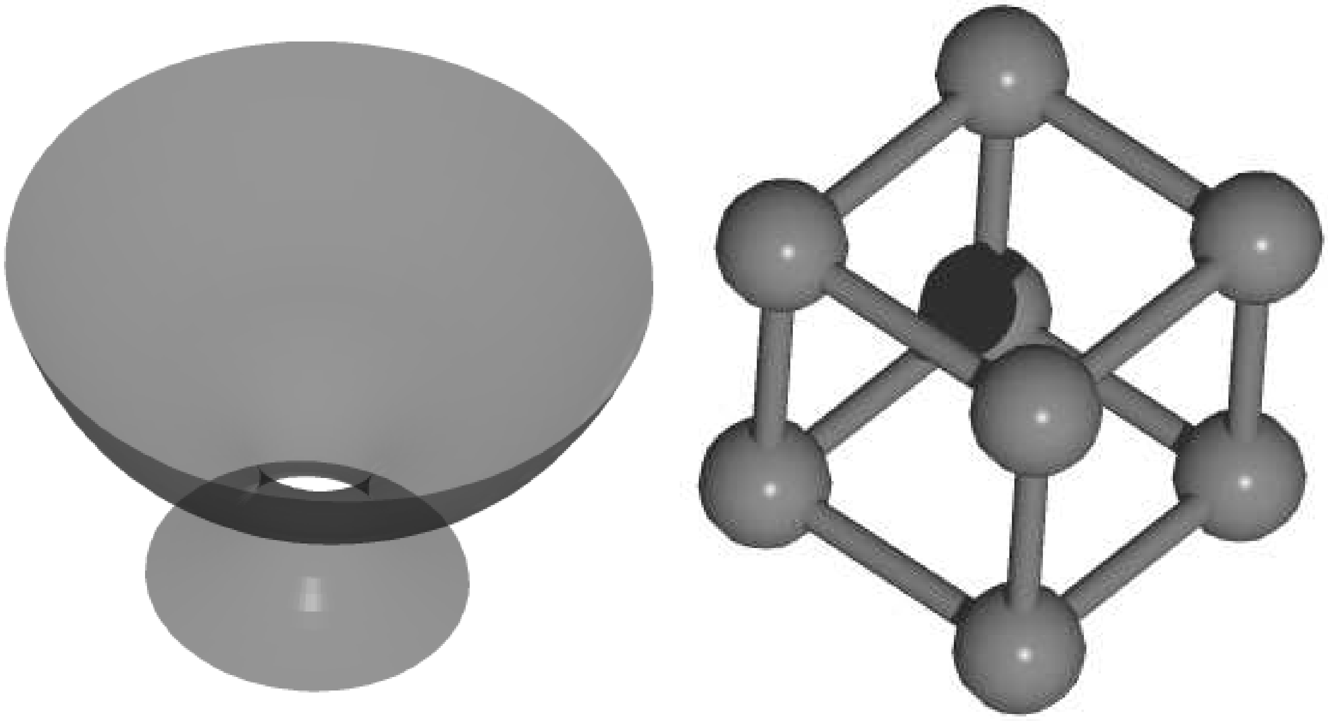


Figure 58: Some examples of the use of irender scan convertion tool to render to render images off *IRIT* scenes. Shadows can be seen in the molecule image while the glass is rendered transparent.

- **-v**: Verbose mode. Prints informative messages as it progresses.
- **-s XSize YSize**: Sets the size of the output image, in pixels. Default to 512x512.
- **-a Ambient**: Sets the ambient lighting fraction. Between zero (no ambient lighting) and one. Default to 0.2.
- **-b R G B**: Sets the background color. Each of thre R,G,B colors is an integer value between zero and 255. Default to black.
- **-B** : Apply back face culling. Somewhat faster, but only correct for closed objects. Default is no back face culling.
- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. Default is 0 and 20.0 (no optimal sampling with fineness of 20.0 (real number)).
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed. Default is 0 64 (no optimal sampling with 64 samples).
- **-M Flat/Gouraud/Phong**: Selects the shader to be used. Default to Phong if has normals of vertices, Flat if no normals are found.
- **-P WMin [WMax]**: Width of rendered polyline, in pixels. If only WMin is specified, all polylines are set to have WMin width. Otherwise, if WMax is prescribed as well, polylines' width is set to be proportional to their depth with WMax is the width of closest polyline and WMin the farest polyline.
- **-S**: Enable shadow computation. No shadows will be rendered without -S.

- **-T**: Enable transparency computation. No transparent object will be processed without -T.
- **-A FilterName**: Selects an antialiasing filter. FilterName can be one of 'none', 'box', 'triangle', 'quadratic', 'cubic', 'catrom', 'mitchell', 'gaussian', 'sinc, and 'bessel'. Default is 'none'.
- **-Z**: Output will be in the form of Z depth instead of a color image. Output will be 32 bits depth instead of RGBA.
- **-n**: Reverses the normals of vertices and planes, globally.
- **-i rle/ppm**: Selects output image type. Currently the Utah Raster Toolkit's (URT) rle format is being supported as well as the PPM format.

  Some of the options may be turned on in irender.cfg. They can be then turned off in the command line as '-?-'.

17.3 Configuration

The program can be configured using a configuration file named irender.cfg. This is a plain ASCII file you can edit directly and set the parameters according to the comments there. 'irender -z' will display the current configuration as read from the configuration file.

The configuration file is searched in the directory specified by the IRIT_PATH environment variable. For example, 'setenv IRIT_PATH /u/gershon/irit/bin/'. If the IRIT_PATH variable is not set, the current directory is searched.

17.4 Usage

As this program is not interactive, usage is quite simple, and the only control available is using the command line options.

The images in Figure 58 were created using the following commands:

```
irender -s 350 350 -b 255 255 255 -S -A sync -i rle lightsrc.dat
                                 molecule.dat view_mat.dat > molecule.rle
irender -s 700 700 -F 0 64 -M Flat -b 255 255 255 -T -A sync -i rle
                                 glass.dat view_mat.dat > glass.rle
```

17.5 Advanced Usage

One can specify several attributes that affect the way the scene is rendered. The attributes can be generated within *IRIT*. See also the ATTRIB *IRIT* command.

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise, a color as set via the *IRIT* COLOR command is used.

If a certain surface should be finer/causer than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. Further, "u_resolution" and "v_resolution" might be similarly used to set relative resolution for the u or v direction only.

Example:

```
attrib( Ball, "rgb", "255,0,0" );
color( Sphere, white );
```

The cosine exponent of the phong shader can be set for a specific object via the SRF_COSINE attribute.

Example:

```
attrib( Ball, "srf_cosine", 16 );
```

An object can be drawn transparent instead of opaque, if it has a "transp" attribute. A transparent value of one denotes a completely transparent object, while a value of zero means a completely opaque object. Transparent object will be rendered as such if and only if the '-T' command line option is set.
Example:

```
attrib( final, "transp", 0.5 );
```

Several types of texture mapping are supported. Parametric texture may be attached to a parametric surface where the prescribed image is mapped onto the rectangular parametric domain of the surface.
Example:

```
attrib( Srf1, "ptexture", "checker.ppm" );
```

The program will automatically detected a ppm file from an rle according to the file's name.
A second type of texture mapping can be applied to all geometric objects. Herein, a procedural texture mapping is employed and currently only "wood" and "marble" are supported. A second parameter that must be provided for procedural textures is the scaling factor of the texture, which can be either one parameter of uniform scaling or a vector of three coefficients for scaling in x, y, and z. The two parameters are seperated by a comma.
Example:

```
attrib( Obj1, "texture", "marble, 2" );
attrib( Obj2, "texture", "wood, 1 0.5 2.5" );
```

which sets **Obj1** to have a marble procedural texture with a uniform scaling factor of 2 and a wood texture for **Obj2** with scaling factors of (1, 0.5, 2.5) in x, y, and z.
In addition, a scalar surface spanning the same parameteric domain as an original surface may be used as texture mapping function. Herein, the scalar function texture is evaluated at each UV parameter value and is mapped through a color scale to yield the output color. This type of texture is useful for stress maps or analysis maps on top of freeform surfaces. Several related attributes are supported: "stexture_scale" which prescribes the color scale image (only its first column is employed), and "stexture_bound" that sets the domain that will be clipped to the min max values. Funally, "stexture_func" can hold the functions "sqrt" or "abs" to be applied to the evaluated surface value.
Example:

```
attrib( Srf, "stexture", scrvtr( Srf, P1, off ) );
attrib( Srf, "stexture_scale", "color_scale.ppm" );
attrib( Srf, "stexture_func", "sqrt" );
attrib( Srf, "stexture_bound", "0.0 100.0" );
```

where scrvtr computes a scalar field to **Srf** that represents the sum of the squares of the principle curvatures. The evaluated scalar texture surface's value is piped through a sqrt function. The first column of the image of **color_scale.ppm** is used to set the coloring scale for curvature bounds values between 0.0 and 100.0.
Both "stexture_scale" and "stexture_bound" are optional. The default color scale maps the min/max values from blue to red through green. The default scalar surface texture bound is computed as the extreme values of the "stexture" surface.
While the program has a default for lighting which is two light sources at opposite directions at (1, 1, 1) and (-1, -1, -1), one can overwrite this default. A POINT_TYPE object with LIGHT_SOURCE attribute denotes a light source. If irender detects one or more light sources in the input stream, the default light sources are not created. Two types of light sources may be prescribed, a parallel at infinity

or a point at finite distance light source, distinguished by a TYPE attribute of either POINT_POS or POINT_INFTY. A point light source can be colored, when an RGB attribute will set its color. A point light source will cast shadows if and only if it has SHADOW attribute (one needs to apply the '-S' command line option as well for rendering shadows). Finally, one can construct two mirrored light sources at opposite directions if TWOLIGHT attribute is added to the light source object.

Example:

```
Light1 = point( 0, 0, 10 );
attrib( Light1, "light_source", on );
attrib( Light1, "shadow", on );
attrib( Light1, "rgb", "255,0,0" );
attrib( Light1, "type", "point_pos" );

Light2 = point( 1, 1, 1 );
attrib( Light2, "light_source", on );
attrib( Light2, "twolight", on );
attrib( Light2, "type", "point_infty" );
```

constructs two lights sources with **Light1** with red color positioned at (0, 0, 10) and casting shadows, while **Light2** will create two mirrored white parallel lights sources in the direction of (1, 1, 1) and (-1, -1, -1), as is irender's default.

18 Dat2Bin - Data To Binary Data file filter

18.1 Command Line Options

```
dat2bin [-t] [-z] DFiles
```

- **-t**: Dumps data to stdout as text instead of binary. **-z**: Print version number and current defaults.

18.2 Usage

It may be sometimes desired to convert .dat data files into a binary form, for example, for fast loading of files with large geometry. Binary files can be somewhat larger, are unreadable in editors but are much faster to load in. A binary file must have a '.bdt' file type.

Example:

```
dat2bin b58polys.dat > b58polys.bdt
dat2bin -t b58polys.bdt | more
```

to convert a text file b58polys.dat into a binary file b58polys.bdt and to view the content of the binary file by converting it back to text. At this time data through pipes must be in text. That is, the following is *illegal*:

```
dat2bin b58polys.dat | xglmdrvs -
```

It should be remembered that the binary format is not documented and it might change in the future. Moreover, it is machine dependent and can very well may be unreadible between different platforms.

19 Dat2Irit - Data To IRIT file filter

Converts '.dat' and '.bdt' data files to '.irt' *IRIT* scripts.

19.1 Command Line Options

```
dat2irit [-z] DFiles
```

- **-z**: Print version number and current defaults.

19.2 Usage

It may be sometimes desired to convert .dat data files into a form that can be fed back to *IRIT* - a '.irt' file. This filter does exactly that.

Example:

```
dat2irit b58.dat > b58-new.irt
```

20 Dxf2Irit - DXF (Autocad) To IRIT filter

Due to lack of real documentation on the DXF format (for surfaces), this filter is not really complete. It only work for polygons, and is provided here only for those desperate enough to try and fix it...

21 Irit2Dxf - IRIT To DXF (Autocad) filter

Due to lack of real documentation on the DXF format (for surfaces), this filter is not really complete. It works only for polygons, and is provided here only for those desperate enough to try and fix it...

22 Irit2Hgl - IRIT To HPGL filter

Converts IRIT geometry into the HL Graphics Language used by HP's plotters.

22.1 Command Line Options

```
irit2hgl [-t XTrans YTrans] [-I #UIso[:#VIso[:#WIso]]]
     [-f PolyOpti SampPerCrv] [-F PolyOpti FineNess] [-M] [-G] [-T]
     [-i] [-o OutName] [-z] DFiles
```

- **-t XTrans YTrans**: X and Y translation. of the image. Default is (0, 0).
- **-I #UIso[:#VIso]**: Specifies the number of isolines per surface, per direction. If #VIso is not specified, #UIso is used for #VIso as well.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.
- **-F PolygonOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4. This enforces the dump of freefrom geometry as polygons.
- **-M**: Dumps the control mesh/polygon as well.
- **-G**: Dumps the freeform geometry.
- **-T**: Talkative mode. Prints processing information.

- **-i**: Internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-o OutName**: Name of output file. By default the name of the first data file from *DFiles* list is used. See below on the output files.
- **-z**: Prints version number and current defaults.

22.2 Usage

Irit2Hgl converts freeform surfaces and polygons into polylines in a format that can be used by HPGL. Example:

```
irit2Hgl -M -f 0 16 saddle.dat > saddle.hgl
```

However, one can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by the display devices:

```
x11drvs b58.dat
irit2Hgl -M -f 0 16 b58.dat irit.mat > saddle.hgl
```

where irit.mat is the viewing matrix created by x11drvs.

23 Irit2Iv - IRIT To SGI's Inventor filter

IV is the format used by the Inventor modeling/rendering package from SGI.

23.1 Command Line Options

```
irit2iv [-l] [-4] [-P] [-F PolyOpti FineNess] [-f PolyOpti SampPerCrv]
                                                      [-T] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although, most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free form shape (saddle like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.
- **-P**: Polygonize freeform shapes. Default is to leave freeform curves and surfaces as is.
- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

23.2 Usage

Irit2Iv converts freeform surfaces and polygons into polygons and saved in iv Inventor's ascii file format.
Example:

```
irit2iv solid1.dat > solid1.iv
```

Surfaces are converted to polygons with fineness control:

```
irit2iv -F 0 16 - view.mat < saddle.dat > saddle.iv
```

Note the use of '-' for stdin.

24 Irit2Nff - IRIT To NFF filter

24.1 Command Line Options

```
irit2nff [-l] [-4] [-c] [-F PolyOpti FineNess] [-o OutName] [-T] [-g]
                                                              [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although, most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.
- **-c**: Output files should be filtered by cpp. When set, the usually huge geometry file is separated from the main nff file that contains the surface properties and view parameters. By default all data, including the geometry, are saved into a single file with type extension '.nff'. Use of '-c' will pull out all the geometry into a file with the same name but a '.geom' extension, which will be included using the '#include' command. The '.nff' file should, in that case, be preprocessed using cpp before being piped into the nff renderer.
- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-o OutName**: Name of output file. By default the name of the first data file from the *DFiles* list is used. See below on the output files.
- **-g**: Generates the geometry file only. See below.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

24.2 Usage

Irit2Nff converts freeform surfaces into polygons in a format that can be used by an NFF renderer. Usually, one file is created with '.nff' type extension. Since the number of polygons can be extremely large, a '-c' option is provided, which separates the geometry from the surface properties and view specification, but requires preprocessing by cpp. The geometry is isolated in a file with extension '.geom' and included (via '#include') in the main '.nff' file. The latter holds the surface properties for all the geometry as well as the viewing specification. This allows for the changing of shading or the viewing properties while editing small ('.nff') files.

If '-g' is specified, only the '.geom' file is created, preserving the current '.nff' file. The '-g' flag can be specified only with '-c'.

In practice, it may be useful to create a low resolution approximation of the model, change viewing/shading parameters in the '.nff' file until a good view and/or surface quality is found, and then run Irit2Nff once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2nff -c -l -F 0 8 b58.dat
```

creates b58.nff and b58.geom with low resolution (FineNess of 5).

Once done with parameter setting, a fine approximation of the model can be created with:

```
irit2nff -c -l -g -F 0 64 b58.dat
```

which will only recreate b58.geom (because of the -g option).

One can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by a display device:

```
xgldrvs b58.dat
irit2nff -l -F 0 32 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by xgldrvs.

24.3 Advanced Usage

One can specify surface qualities for individual surfaces of a model. Several such attributes are supported by Irit2Nff and can be set within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain surface should be finer/caurser than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. Further, "u_resolution" and "v_resolution" might be similarly used to set relative resolution for the u or v direction only.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The rectangle can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the center of the patch. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces by setting a "twoperflat" or a "fourperflat" attribute.

NFF specific properties are controlled via the following attributes: "kd", "ks", "shine", "trans", "index". Refer to the NFF manual for detail.

Example:

```
attrib( srf1, "kd", 0.3 );
attrib( srf1, "shine", 30 );
```

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise, a color, as set via the *IRIT* COLOR command, is used if set.

Example:

```
attrib( tankBody, "rgb", "244,164,96" );
```

25 Irit2Plg - IRIT To PLG (REND386) filter

PLG is the format used by the rend386 real time renderer for the IBM PC.

25.1 Command Line Options

```
irit2plg [-l] [-4] [-F PolyOpti FineNess] [-T] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although, most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free form shape (saddle like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.
- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

25.2 Usage

Irit2Plg converts freeform surfaces and polygons into polygons in a format that can be used by the REND386 renderer.

Example:

```
irit2plg solid1.dat > solid1.plg
```

Surfaces are converted to polygons with fineness control:

```
irit2plg -F 0 16 - view.mat < saddle.dat > saddle.plg
```

Note the use of '-' for stdin.

26 Irit2Ps - IRIT To PS filter

26.1 Command Line Options

```
irit2ps [-l] [-4] [-s Size] [-I #UIso[:#VIso[:#WIso]]] [-F PolyOpti FineNess]
        [-f PolyOpti SampPerCrv] [-M] [-G] [-P] [-W LineWidth]
        [-w WidenLen WidenWidth] [-b R G B] [-B X1 Y1 X2 Y2] [-c] [-C]
        [-T] [-i] [-o OutName] [-d [Zmin Zmax]] [-D [Zmin Zmax]]
        [-p PtType PtSize] [-u] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.

- **-s Size**: Controls the size of the postscript output in inches. Default is to fill the entire screen.
- **-I #UIso[:#VIso[:#WIso]]**: Specifies the number of isolines per surface/trivariate, per direction. If #VIso or #WIso is not specified, #UIso is used for #VIso etc.
- **-F PolygonOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.
- **-M**: Dumps the control mesh/polygon as well.
- **-G**: Dumps the curve/surface (as freeform geometry). Default. See -I, -C, -f for control on polyline approximation.
- **-P**: Dumps the curve/surface (as polygons). See -F, -l, -4 for control on polygonal approximation.
- **-W #LineWidth**: Sets the line drawing width in inches. Default is as thin as possible. This option will overwrite only those objects that do *not* have a "width" attribute. See also -d. If LineWidth is negative its absolute value is used to scale the current width of the object if has one, or the default width otherwise.
- **-w WidenLen WidenWidth**: If end points of polylines should be made wider, and if so to what width.
- **-b R G B**: Sets a colored background. RGB are three integers prescribing the Red, Green, and Blue coefficients. if no -c (i.e. a gray level drawing) this color is converted to a gray level using RGB to T.V. Y(IQ) channel conversion.
- **-B X1 Y1 X2 Y2**: Clips the drawing area outsize the bounding box from (X1, Y1) to (X2, Y2).
- **-c**: Creates a *color* postscript file.
- **-C**: Curve mode. Dumps freeform curves and surfaces as cubic Bezier curves. Higher order curves and surfaces and/or rationals are approximated by cubic Bezier curves. This option generates data files that are roughly a third of piecewise linear postscript files (by disabling this feature, -C-), but takes a longer time to compute.
- **-T**: Talkative mode. Prints processing information.
- **-i**: Internal edges (created by *IRIT*) - the default is not to display them, and this option will force displaying them as well.
- **-o OutName**: Name of output file. Default is stdout.
- **-d [Zmin Zmax]**: Sets the ratios between the depth cue and the width of the dumped data. See also -W, -p. Closer lines/points will be drawn wider/larger. Zmin and Zmax are optional. The object's bounding box is otherwise computed and used.
- **-D [Zmin Zmax]**: Same as -d, but depth cue the color or gray scale instead of width. You might need to consider the sorting option of the illustrt tool (-s of illustrt) for proper drawings. Only one of -d and -D can be used.
- **-p PtType PtSize**: Specifies the way points are drawn. PtType can be one of H, F, C for Hollow circle, Full Circle, or Cross. PtSize specifies the size of the point to be drawn, in inches. Vectors will also be drawn as points, but with an additional thin line to the origin. See also -d.

- **-u**: Forces a unit matrix transformation, i.e. no transformation.
- **-z**: Prints version number and current defaults.

26.2 Usage

Irit2Ps converts freeform surfaces and polygons into a postscript file.
Example:

```
irit2ps solid1.dat > solid1.ps
```

Surfaces are converted to polygons with fineness control:

```
irit2ps -f 0 32 -c -W 0.01 saddle.dat > saddle.ps
```

creates a postscript file for the saddle model, in color, and with lines 0.01 inch thick.

26.3 Advanced Usage

One can specify several attributes that affect the way the postscript file is generated. The attributes can be generated within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain object should be thinner or thicker than the rest of the scene, one can set a "width" attribute which specifies the line width in inches of this specific object.
Example:

```
attrib( srf1, "width", 0.02 );
```

will force srf1 to have this width, instead of the default as set via the '-W' flag.

If a (closed) object, a polygon for example, needs to be filled, a "fill" attribute should be set, with a value equal to the gray level desired.
Example:

```
attrib( poly, "fill", 0.5 );
```

will fill poly with %50 gray.

If an object, a polygon for example, needs to be painted in a gray level instead of black, a "gray" attribute should be set, with a value equal to the gray level desired.
Example:

```
attrib( poly, "gray", 0.5 );
```

will draw poly with %50 gray.

Dotted or dashed line effects can be created using a "dash" attribute which is a direct PostScript dash string. A simple form of this string is "[a b]" in which a is the drawing portion (black) in inches, followed by b inches of white space. See the postScript manual for more about the format of this string. Here is an example for a dotted-dash line.

```
attrib( poly, "dash", "[0.006 0.0015 0.001 0.0015] 0" );
```

Surface color is controlled (for color postscript only - see -c) in two levels. If the object has an RGB attribute, it is used. Otherwise, a color as set via the *IRIT* COLOR command is used.
Example:

```
attrib( Ball, "rgb", "255,0,0" );
```

An object can be drawn as "tubes" instead of full lines. The ratio between the inner and the outer radii of the tube is provided as the TUBULAR attribute:

```
attrib( final, "tubular", 0.7 );
```

27 Irit2Ray - IRIT To RAYSHADE filter

27.1 Command Line Options

```
irit2ray [-l] [-4] [-G GridSize] [-F PolyOpti FineNess]
        [-f PolyOpti SampPerCrv] [-o OutName] [-g] [-p Zmin Zmax] [-P]
        [-M] [-T] [-I #UIso[:#VIso[:#WIso]]] [-s ObjSeq#] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.
- **-G GridSize**: Usually objects are grouped as *lists* of polygons. This flags will coerce the usage of the RAYSHADE *grid* structure, with *GridSize* being used as the grid size along the object bounding box's largest dimension.
- **-F PolygonOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.
- **-o OutName**: Name of output file. By default the name of the first data file from the *DFiles* list is used. See below on the output files.
- **-g**: Generates the geometry file only. See below.
- **-p Zmin Zmax**: Sets the ratios between the depth cue and the width of the dumped *polylines*. See also -P. Closer lines will be drawn wider.
- **-P**: Forces dumping polygons as polylines with thickness controlled by -p.
- **-M**: If -P (see -P and -p) then converts the control mesh/polygon to polylines which are represented as a sequence of truncated cones.
- **-T**: Talkative mode. Prints processing information.
- **-I #UIso[:#VIso[:#WIso]]**: Specifies the number of isolines per surface/trivariate, per direction. If #VIso or #WIso is not specified, #UIso is used for #VIso etc.
- **-s ObjSeq#**: Sets object sequence number if no object name. Default 1.
- **-z**: Prints version number and current defaults.

27.2 Usage

Irit2Ray converts freeform surfaces into polygons in a format that can be used by the RAYSHADE ray tracing program. Two files are created, one with a '.geom' extension and one with '.ray'. Since the number of polygons can be extremely large, the geometry is isolated in the '.geom' file and is included (via '#include') in the main '.ray' file. The latter holds the surface properties for all the geometry as well as viewing and RAYSHADE specific commands. This allows for the changing of the shading or the viewing properties while editing small ('.ray') files.

If '-g' is specified, only the '.geom' file is created, preserving the current '.ray' file.

In practice, it may be useful to create a low resolution approximation of the model, change the viewing/shading parameters in the '.ray' file until a good view and/or surface quality is found, and then run Irit2Ray once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2ray -l -F 0 8 b58.dat
```

creates b58.ray and b58.geom with low resolution (FineNess of 5). At such low resolution it can very well may happen that triangles will have normals "over the edge" since a single polygon may approximate a highly curved surface. That will cause RAYSHADE to issue an "Inconsistent triangle normals" warning. This problem will not exist if high fineness is used. One can ray trace this scene using a command similar to:

```
RAYSHADE -p -W 256 256 b58.ray > b58.rle
```

Once done with parameter setting for RAYSHADE, a fine approximation of the model can be created with:

```
irit2ray -l -g -F 0 64 b58.dat
```

which will only recreate b58.geom (because of the -g option).

Interesting effects can be created using the depth cue support and polyline conversion of irit2ray. For example

```
irit2ray -G 5 -P -p -0.0 0.5 solid1.dat
```

will dump solid1 as a set of polylines (represented as truncated cones in RAYSHADE) with varying thickness according to the z depth. Another example is

```
irit2ray -G 5 -P -p -0.1 1.0 saddle.dat
```

which dumps the isolines extracted from the saddle surface with varying thickness.

Each time a data file is saved in *IRIT*, it can be saved with the viewing matrix of the last INTERACT by saving the VIEW_MAT object as well. I.e.:

```
save( "b58", b58 );
```

However one can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by the display devices:

```
os2drvs b58.dat
irit2ray -l -F 0 16 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by os2drvs. The output name, by default, is the last input file name, so you might want to provide an explicit name with the -o flag.

27.3 Advanced Usage

One can specify surface qualities for individual surfaces of a model. Several such attributes are supported by Irit2Ray and can be set within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain surface should be finer/caurser than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. Further, "u_resolution" and "v_resolution" might be similarly used to set relative resolution for the u or v direction only.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The rectangle can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the patch center. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces bu setting "twoperflat" or "fourperflat".

RAYSHADE specific properties are controlled via the following attributes: "specpow", "reflect", "transp", "body", "index", and "texture". The value of this attributes must be strings as it is copied verbatim. Refer to RAYSHADE's manual for their meaning.

Example:

```
attrib( legs, "transp", "0.3" );
attrib( legs, "texture", "wood,2" );
attrib( table, "texture", "marble" );
attrib( table, "reflect", "0.5" );
```

Optional scale can be prescribed to textures. In the above example wooden legs' (that are also transparent...) texture is selected with texture's scaling factor of 2.

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise a color as set via the *IRIT* COLOR command is being used if set.

Example:

```
attrib( tankBody, "rgb", "244,164,96" );
```

28 Irit2Scn - IRIT To SCENE (RTrace) filter

SCENE is the format used by the RTrace ray tracer. This filter was donated by Antonio Costa (acc@asterix.inescn.pt), the author of RTrace.

28.1 Command Line Options

```
irit2scn [-l] [-4] [-F PolyOpti FineNess] [-o OutName] [-g] [-T] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated as a single polygon along their linear direction. Although most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.

- **-4**: Four - Generates four polygons per flat patch.

- **-F PolyOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4.

- **-o OutName**: Name of output file. By default the name of the first data file from *DFiles* list is used. See below on the output files.
- **-g**: Generates the geometry file only. See below.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

28.2 Usage

Irit2Scn converts freeform surfaces and polygons into polygons in a format that can be used by RTrace. Two files are created, one with a '.geom' extension and one with '.scn'. Since the number of polygons can be extremely large, the geometry is isolated in the '.geom' file and is included (via '#include') in the main '.scn' file. The latter holds the surface properties for all the geometry as well as viewing and RTrace specific commands. This allows for the changing of the shading or the viewing properties while editing small ('.scn') files.

If '-g' is specified, only the '.geom' file is created, preserving the current '.scn' file.

In practice, it may be useful to create a low resolution approximation of the model, change the viewing/shading parameters in the '.scn' file until a good view and/or surface quality is found, and then run Irit2Scn once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2scn -l -F 0 8 b58.dat
```

creates b58.scn and b58.geom with low resolution (FineNess of 5).

One can ray trace this scene after converting the scn file to a sff file, using scn2sff provided with the RTrace package.

Once done with the parameter setting of RTrace, a fine approximation of the model can be created with:

```
irit2scn -l -g -F 0 64 b58.dat
```

which will only recreate b58.geom (because of the -g option).

One can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by the display devices:

```
wntdrvs b58.dat
irit2scn -l -F 0 8 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by wntdrvs. The output name, by default, is the last input file name, so you might want to provide an explicit name with the -o flag.

28.3 Advanced Usage

One can specify surface qualities for individual surfaces of a model. Several such attributes are supported by Irit2Scn and can be set within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain surface should be finer/caurser than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. Further, "u_resolution" and "v_resolution" might be similarly used to set relative resolution for the u or v direction only.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The patch can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the patch center. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces bu setting "twoperflat" or "fourperflat".

RTrace specific properties are controlled via the following attributes: "SCNrefraction", "SCNtexture", "SCNsurface. Refer to the RTrace manual for their meaning.

Example:

```
attrib( srf1, "SCNrefraction", 0.3 );
```

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise a color as set via *IRIT* COLOR command is used if set.

Example:

```
attrib( tankBody, "rgb", "244,164,96" );
```

29 Irit2Xfg - IRIT To XFIG filter

29.1 Command Line Options

```
irit2xfg [-s Size] [-t XTrans YTrans] [-I #UIso[:#VIso[:#WIso]]]
     [-f PolyOpti SampPerCrv] [-F PolyOpti FineNess] [-M] [-G] [-T]
     [-i] [-o OutName] [-z] DFiles
```

- **-s Size**: Size in inches of the page. Default is 7 inches.
- **-t XTrans YTrans**: X and Y translation. of the image. Default is (0, 0).
- **-I #UIso[:#VIso]**: Specifies the number of isolines per surface, per direction. If #VIso is not specified, #UIso is used for #VIso as well.
- **-f PolyOpti SampPerCrv**: Controls the method used to approximate curves into polylines. If PolyOpti == 0, equally spaced intervals are used. Otherwise, an adaptive subdivision that optimizes the samples is employed.
- **-F PolygonOpti FineNess**: Optimality of polygonal approximation of surfaces. See the variable POLY_APPROX_OPT for the meaning of FineNess. See also -4. This enforces the dump of freefrom geometry as polygons.
- **-M**: Dumps the control mesh/polygon as well.
- **-G**: Dumps the freeform geometry.
- **-T**: Talkative mode. Prints processing information.
- **-i**: Internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-o OutName**: Name of output file. By default the name of the first data file from *DFiles* list is used. See below on the output files.
- **-z**: Prints version number and current defaults.

29.2 Usage

Irit2Xfg converts freeform surfaces and polygons into polylines in a format that can be used by XFIG.
Example:

```
irit2Xfg -T -f 0 16 saddle.dat > saddle.xfg
```

However, one can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by the display devices:

```
x11drvs b58.dat
irit2Xfg -T -f 0 16 b58.dat irit.mat > saddle.xfg
```

where irit.mat is the viewing matrix created by x11drvs.

30 Data File Format

This section describes the data file format used to exchange data between *IRIT* and its accompanying tools.

```
[OBJECT {ATTRS} OBJNAME
    [NUMBER n]

  | [VECTOR x y z]

  | [CTLPT POINT_TYPE {w} x y {z}]

  | [STRING "a string"]

  | [MATRIX m00 ... m03
            m10 ... m13
            m20 ... m23
            m30 ... m33]

    ;A polyline should be drawn from first point to last. Nothing is drawn
    ;from last to first (in a closed polyline, last point is equal to first).
  | [POLYLINE {ATTRS} #PTS                    ;#PTS = number of points.
        [{ATTRS} x y z]
        [{ATTRS} x y z]
           .
           .
           .
        [{ATTRS} x y z]
    ]

    ;Defines a closed planar region. Last point is NOT equal to first,
    ;and a line from last point to first should be drawn when the boundary
    ;of the polygon is drawn.
  | [POLYGON {ATTRS} #PTS
        [{ATTRS} x y z]
        [{ATTRS} x y z]
```

```
          .
          .
          .
        [{ATTRS} x y z]
    ]

    ;Defines a "cloud" of points.
  | [POINTLIST {ATTRS} #PTS
        [{ATTRS} x y z]
        [{ATTRS} x y z]
          .
          .
          .
        [{ATTRS} x y z]
    ]

    ;Defines a Bezier curve with #PTS control points. If the curve is
    ;rational, the rational component is introduced first.
  | [CURVE BEZIER {ATTRS} #PTS POINT_TYPE
        [{ATTRS} {w} x y z ...]
        [{ATTRS} {w} x y z ...]
          .
          .
          .
        [{ATTRS} {w} x y z ...]
    ]

    ;Defines a Bezier surface with #UPTS * #VPTS control points. If the
    ;surface is rational, the rational component is introduced first.
    ;Points are printed row after row (#UPTS per row), #VPTS rows.
  | [SURFACE BEZIER {ATTRS} #UPTS #VPTS POINT_TYPE
        [{ATTRS} {w} x y z ...]
        [{ATTRS} {w} x y z ...]
          .
          .
          .
        [{ATTRS} {w} x y z ...]
    ]

    ;Defines a Bezier trivariate with #UPTS * #VPTS * #WPTS control
    ;points. If the trivariate is rational, the rational component is
    ;introduced first. Points are printed row after row (#UPTS per row),
    ;#VPTS rows, #WPTS layers (depth).
  | [TRIVAR BEZIER {ATTRS} #UPTS #VPTS #WPTS POINT_TYPE
        [{ATTRS} {w} x y z ...]
        [{ATTRS} {w} x y z ...]
          .
          .
          .
        [{ATTRS} {w} x y z ...]
```

```
  ]

  ;Defines a Bspline curve of order ORDER with #PTS control points. If the
  ;curve is rational, the rational component is introduced first.
  ;Note length of knot vector is equal to #PTS + ORDER.
  ;If curve is periodic KVP prefix the knot vector that has length of
  ;'Length + Order + Order - 1'.
| [CURVE BSPLINE {ATTRS} #PTS ORDER POINT_TYPE
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                    ;Knot vector
      [{ATTRS} {w} x y z ...]
      [{ATTRS} {w} x y z ...]
         .
         .
         .
      [{ATTRS} {w} x y z ...]
  ]

  ;Defines a Bspline surface with #UPTS * #VPTS control points, of order
  ;UORDER by VORDER. If the surface is rational, the rational component
  ;is introduced first.
  ;Points are printed row after row (#UPTS per row), #VPTS rows.
  ;If surface is periodic in some direction KVP prefix the knot vector
  ;that has length of 'Length + Order + Order - 1'.
| [SURFACE BSPLINE {ATTRS} #UPTS #VPTS UORDER VORDER POINT_TYPE
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                ;U Knot vector
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                ;V Knot vector
      [{ATTRS} {w} x y z ...]
      [{ATTRS} {w} x y z ...]
         .
         .
         .
      [{ATTRS} {w} x y z ...]
  ]

  ;Defines a Bspline trivariate with #UPTS * #VPTS * #WPTS control
  ;points. If the trivariate is rational, the rational component is
  ;introduced first. Points are printed row after row (#UPTS per row),
  ;#VPTS rows, #WPTS layers (depth).
  ;If trivariate is periodic in some direction KVP prefix the knot vector
  ;that has length of 'Length + Order + Order - 1'.
| [TRIVAR BSPLINE {ATTRS} #UPTS #VPTS #WPTS UORDER VORDER WORDER POINT_TYPE
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                ;U Knot vector
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                ;V Knot vector
      [KV{P} {ATTRS} kv0 kv1 kv2 ...]                ;W Knot vector
      [{ATTRS} {w} x y z ...]
      [{ATTRS} {w} x y z ...]
         .
         .
         .
      [{ATTRS} {w} x y z ...]
```

```
    ]

    ;Defines a trimmed surface. Encapsulates a surface (can be either a
    ;Bspline or a Bezier surface) and prescribes its trimming curves.
    ;There can be an arbitrary number of trimming curves (either Bezier
    ; or Bspline). Each trimming curve contains an arbitrary number of
    ;trimming curve segments, while each trimming curve segment contains
    ;a parameteric representation optionally followed by a Euclidean
    ;representation of the trimming curve segment.
  | [TRIMSRF
        [SURFACE ...
        ]
        [TRIMCRV
            [TRIMCRVSEG
                [CURVE ...
                ]
            ]
               .
               .
               .
            [TRIMCRVSEG
                [CURVE ...
                ]
            ]
        ]
            .
            .
            .
        [TRIMCRV
            [TRIMCRVSEG
                [CURVE ...
                ]
            ]
               .
               .
               .
            [TRIMCRVSEG
                [CURVE ...
                ]
            ]
        ]
    ]
]

POINT_TYPE -> E1 | E2 | E3 | E4 | E5 | P1 | P2 | P3 | P4 | P5

ATTRS -> [ATTRNAME ATTRVALUE]
       | [ATTRNAME]
       | [ATTRNAME ATTRVALUE] ATTRS
```

Some notes:

* This definition for the text file is designed to minimize the reading time and space. All information can be read without backward or forward referencing.

* An OBJECT must never hold different geometry types or other entities. I.e. CURVEs, SURFACEs, and POLYGONs must all be in different OBJECTs.

* Attributes should be ignored if not needed. The attribute list may have any length and is always terminated by a token that is NOT '`[`'. This simplifies and disambiguates the parsing.

* Comments may appear between '`[OBJECT ...]`' blocks, or immediately after OBJECT OBJNAME, and only there.

A comment body can be anything not containing the '`[`' or the '`]`' tokens (signals start/end of block). Some of the comments in the above definition are *illegal* and appear there only of the sake of clarity.

* It is preferred that geometric attributes such as NORNALs will be saved in the geometry structure level (POLYGON, CURVE or vertices) while graphical and others such as COLORs will be saved in the OBJECT level.

* Objects may be contained in other objects to an arbitrary level.

Here is an example that exercises most of the data format:

```
This is a legal comment in a data file.
[OBJECT DEMO
    [OBJECT REAL_NUM
        And this is also a legal comment.
        [NUMBER 4]
    ]

    [OBJECT A_VECTOR
        [VECTOR 1 2 3]
    ]

    [OBJECT CTL_POINT
        [CTLPT E3 1 2 3]
    ]

    [OBJECT STR_OBJ
        [STRING "string"]
    ]

    [OBJECT UNIT_MAT
        [MATRIX
           1 0 0 0
           0 1 0 0
           0 0 1 0
           0 0 0 1
        ]
    ]

    [OBJECT [COLOR 4] POLY1OBJ
        [POLYGON [PLANE 1 0 0 0.5] 4
            [-0.5 0.5 0.5]
            [-0.5 -0.5 0.5]
            [-0.5 -0.5 -0.5]
```

```
            [-0.5 0.5 -0.5]
        ]
        [POLYGON [PLANE 0 -1 0 0.5] 4
            [0.5 0.5 0.5]
            [-0.5 0.5 0.5]
            [-0.5 0.5 -0.5]
            [0.5 0.5 -0.5]
        ]
    ]

    [OBJECT [COLOR 63] ACURVE
        [CURVE BSPLINE 16 4 E2
            [KV 0 0 0 0 1 1 1 2 3 4 5 6 7 8 9 10 11 11 11 11]
            [0.874 0]
            [0.899333 0.0253333]
            [0.924667 0.0506667]
            [0.95 0.076]
            [0.95 0.76]
            [0.304 1.52]
            [0.304 1.9]
            [0.494 2.09]
            [0.722 2.242]
            [0.722 2.318]
            [0.38 2.508]
            [0.418 2.698]
            [0.57 2.812]
            [0.57 3.42]
            [0.19 3.572]
            [0 3.572]
        ]
    ]

    [OBJECT [COLOR 2] SOMESRF
        [SURFACE BEZIER 3 3 E3
            [0 0 0]
            [0.05 0.2 0.1]
            [0.1 0.05 0.2]

            [0.1 -0.2 0]
            [0.15 0.05 0.1]
            [0.2 -0.1 0.2]

            [0.2 0 0]
            [0.25 0.2 0.1]
            [0.3 0.05 0.2]
        ]
    ]
]
```

31 Bugs and Limitations

Like any program of more than one line, it is far from being perfect. Some limitations, as well as simplifications, are laid out below.

* If the intersection curve of two objects falls exactly on polygon boundaries, for all polygons, the system will scream that the two objects do not intersect at all. Try to move one by EPSILON into the other. I probably should fix this one - it is supposed to be relatively easy.

* Avoid degenerate intersections that result with a point or a line. They will probably cause wrong propagation of the inner and outer part of one object relative to the other. Always extend your object beyond the other object.

* If two objects have no intersection in their boundary, *IRIT* assumes they are disjoint: a union simply combines them, and the other Boolean operators return a NULL object. One should find a FAST way (3D Jordan theorem) to find the relation between the two (A in B, B in A, A disjoint B) and according to that, make a decision.

* Since the boolean sum implementation constructs ruled surfaces with uniform speed, it might return a somewhat incorrect answer, given non-uniform input curves.

* The parser is out of hand and is difficult to maintain. There are several memory leaks there that one should fix.

* The X11 driver has no menu support (any easy way to have menus using Xlib!?).

* IBM R6000 fails to run the drivers in -s- mode.

* Rayshade complains a lot about degenerate polygons on irit2ray output. To alleviate the problem, change the 'equal' macro in common.h in libcommon of rayshade from EPSILON (1e-5) to 1e-7 or even lower.

* On the motif-based drivers (xmtdrvs etc.) clicking the mouse left and right of the scale's button produces stepped transformations. This step size is constant, and is not proportional to the distance between the mouse's position and the position of the button. The reason for the flaw is incorrect callback information returned from the scale in repeattive mode.

* Binary data files are not documented, nor will they be. They might change in the future and are in fact machine dependend. Hence, one platform might fail to read the other's binary data file.