IRIT

A Solid modeling Program

EMail: gershon@gr.utah.edu

This manual is for IRIT version 3.0.

Contents

1 Introduction

IRIT is a small solid modeler developed for educational purposes. Although small, it is now powerful enough to create quite complex scenes. I wrote it mainly so I can learn all the little small and not so small problems in developing such a system.

IRIT started as a polygonal solid modeler and was originally developed (and mostly still is) 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.

2 Copyrights

BECAUSE *IRIT* AND ITS SUPPORTING TOOLS AS DOCUMENTED IN THIS DOCUMENT ARE LICENSED FREE OF CHARGE, I PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, I GERSHON ELBER PROVIDE *IRIT* PROGRAM AND ITS SUPPORTING TOOLS "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THESE PROGRAMS IS WITH YOU. SHOULD THE *IRIT* PROGRAMS PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL GERSHON ELBER, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR A FAILURE OF THE PROGRAMS TO OPERATE WITH PROGRAMS NOT DISTRIBUTED BY GERSHON ELBER) THE PROGRAMS, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

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.

You are not obligated to me or to anyone else in any way by using *IRIT*. You are encourage 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 any non commerical 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 all directories as specified by the PATH environment variable on MSDOS BC++ port or in the IRIT_PATH environment variable under UNIX hosts or MSDOS DJGPP port. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. Note IRIT_PATH must terminate with '/'. If the variables is not set only the current directory is being searched for **IRIT.CFG** under UNIX systems. Since there exists a configuration file for MSDOS BC++ called IRIT-DOS.CFG, for MSDOS DJGPP called IRIT_DJG.CFG, and for UNIX called IRIT-UNX.CFG, make sure you copy the right one for your system, into **IRIT.CFG**.

In addition, if 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 ALIASs 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.

Under MSDOS the solid modeler can be executed in text mode (see the .cfg and the -t flag below) on virtually any MSDOS compatible system. The BC++ port uses Borlands BGI interface which makes it possible to use almost any device that has a BGI interface from Hercules to a Super VGA. See the configuration file for the supported device. The DJGPP port can be used on any system that has graphic driver for it in this compiler. Under BC++ the coprocessor will be used if detected but floating pointing emulation will take place otherwise. Under DJGPP the provided emulator (emu387) must be used if no 387 exists. If a mouse or a joystick exists they can be use by enabling their respective flags in the configuration file. The mouse sensitivity can be controlled via the configuration file MouseSensitivity flag. In addition whether a mouse or a joystick exists or not, the numeric keypad can be used to move the cursor as well. Shifted keys will move the cursors 10 pixels at a time instead of one. Since MSDOS does not support windowing system, an interface library called intr_lib is used and which provides the windowing and interfacing capabilities expected from a windowing system. Four windows are created under BC++, three under DJGPP:

View	The window where geometry is displayed.
Transformation	The window holding the transformation interaction.
	This window pops up with INTERACT command, by default.
Input	The window input is keyed in/sourced from a file.
Status	Status window, mainly for core left display (only BC++).

These windows can be placed everywhere in the screen by specifing their position and size in the configuration file. Border width and color of the window can be specified in the configuration file as well as other window attributes such as smooth scrolling and window headers. Furthermore, these windows can be moved, resized, poped, or pushed during a session via a pop up window triggered by the left mouse button (EXECUTE - see INTERACT) or F1 keystroke. Note however that while the mouse cursor is on the Input window scroll bar or in the Trans window, EXECUTE has a different meaning and the pop up menu will not be displayed. The following entries can be found in the pop up menu, and their meaning.

Redraw all	Redraw all windows. No window configuration is modified.
Move	Move a window. Window to move is picked by left mouse button.
Resize	Resize a window. Window to resize is picked as above.
Pop	Pop up a window. Window to pop up is picked as above.
Push	Push up a window. Window to push down is picked as above.
Zoom	Zoom a window to full screen. Window is picked as above.
	Zoom on a zoomed window restore the window original size.
Reset	Reset all windows to their default position and size.
Headers	Toggle the display of windows header - names.

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

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

If poly3d is used under SGI gl library, you can set the prefered windows location in the poly3d.cfg. No color control is provided at this time.

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

When developing new model under MSDOS, it is extremely convenient to switch between the solid modeler and an editor of your choice to edit the '.irt' file. The solid modeler provides a way to fork to an editor of your choice to edit the current '.irt' program you develop. Set EditPrgm (in irit.cfg) to the editor of your choice, which should be small for fast access. Specify the full path to that editor. Use the edit command in *IRIT* to call it. Only one argument is given to the editor once forking to it - the '.irt' file name as given to the edit command (see EDIT command). You can alias the command to make it even faster as it is done in the current iritinit.irt - *IRIT* initialization file. Under UNIX one can use the gnu emacs irit.el *IRIT* mode provided in this package to run *IRIT*.

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 INTERACT and VIEW 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 directly invoke instead of waiting to input from stdin.

4 First Usage

Once executed, the program opens four windows under MSDOS BC++ (three under MSDOS DJGPP - no status window): view, status, trans and input windows, and opens a view window under UNIX.

Commands are entered through the input window, which is in the same window as the shell executed *IRIT* under UNIX. Objects are viewed in the view window, and the status window is used as an auxiliary window (MSDOS only).

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 (must be capitalized as this is an alias loaded via the iritinit.irt initialization file - see the ALIAS command for more).

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

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

Be careful. Most operators are overloaded. This means that you can multiply two scalars (numbers) or two vectors or even two matrices with the same multiplication symbol (∗). To get its on line type 'help("∗");'

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

Under MSDOS, input from keyboard has full line editing capability:

1. Up/Down arrow : retrieve old command for 10 last commands.
2. Left/Right arrows : to move left and right along the line.
3. Home/End : to move to beginning/end of line.
4. Delete : to delete the character the cursor is on.
5. Back space : to delete one char backward.
6. Insert : toggles insert/overwrite mode. Note cursor shape is modified.
7. Esc : clear the entire line.
8. CR : to accept line and quit.

5 Data Types

These are the Data Types recognized the system. 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 (points/vectors).
CtlPtType	Curve or Surface Control Point.
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.
GeometriceType	One of Polygon/lineType, CurveType, SurfaceType.
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. Implementation is currently limited to 250 (50 MSDOS) objects in a list.

6 Commands summary

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

+	CBEZIER	CRAISE	GPOLYLINE	ROTY	SREFINE
−	CBSPLINE	CREFINE	HELP	ROTZ	SREGION
*	CDIVIDE	CREGION	IF	RULEDSRF	STANGENT
/	CEDITPT	CROSSEC	INCLUDE	SAVE	SURFREV
^	CEVAL	CSURFACE	INTERACT	SBEZIER	SWEEPSRF
ABS	CHDIR	CTANGENT	LIST	SBSPLINE	SYSTEM
ACOS	CIRCLE	CTLPT	LN	SCALE	TAN
ALIAS	CIRCPOLY	CYLIN	LOAD	SDIVIDE	TIME
ARC	CLOSED	DIR	LOG	SEDITPT	TORUS
AREA	CMESH	EDIT	LOGFILE	SEVAL	TRANS
ASIN	COLOR	EXIT	MERGEPOLY	SFROMCRVS	VARLIST
ATAN	COMMENT	EXP	NORMAL	SIN	VECTOR
ATAN2	CON2	EXTRUDE	NTH	SNOC	VIEW
ATTRIB	CONE	FOR	OFFSET	SNORMAL	VOLUME
BEEP	CONVEX	FREE	PAUSE	SPHERE	
BOOLSUM	COS	GBOX	POLY	SQRT	
BOX	CPOLY	GPOLYGON	ROTX	SRAISE	

7 Functions and Variables

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

These are the functions returning a **NumericType**:

ABS	ASIN	COS	LN	SQRT
ACOS	ATAN	CPOLY	LOG	TAN
AREA	ATAN2	EXP	SIN	VOLUME

These are the functions returning a **GeometricType**:

ARC	CIRCLE	CREFINE	GBOX	SBSPLINE	SREFINE
BOOLSUM	CIRCPOLY	CREGION	GPOLYGON	SDIVIDE	SREGION
BOX	CMESH	CROSSEC	GPOLYLINE	SEDITPT	STANGENT
CBEZIER	CON2	CSURFACE	MERGEPOLY	SEVAL	SURFREV
CBSPLINE	CONE	CTANGENT	OFFSET	SFROMCRVS	SWEEPSRF
CDIVIDE	CONVEX	CTLPT	POLY	SNORMAL	TORUS
CEDITPT	CPOLY	CYLIN	RULEDSRF	SPHERE	
CEVAL	CRAISE	EXTRUDE	SBEZIER	SRAISE	

These are the functions to linearly transform an object:

ROTX	ROTY	ROTZ	SCALE	TRANS

These are the miscellaneous functions:

ALIAS	COLOR	FOR	INTERACT	NTH	TIME
ATTRIB	COMMENT	FREE	LIST	PAUSE	VARLIST
BEEP	DIR	HELP	LOAD	SAVE	VECTOR
CHDIR	EDIT	IF	LOGFILE	SNOC	VIEW
CLOSED	EXIT	INCLUDE	NORMAL	SYSTEM	

These are the variables predefined in the system:

AXES	DUMPLVL	FLAT4PLY	INTERNAL	RESOLUTION
DRAWCTLPT	ECHOSRC	INTERCRV	MACHINE	VIEW_MAT

These are the constants predefined in the system:

APOLLO	E2	KV_FLOAT	ON	ROW	WHITE
BLACK	E3	KV_OPEN	P2	SGI	YELLOW
BLUE	FALSE	MAGENTA	P3	SUN	
COL	GREEN	MSDOS	PI	TRUE	
CYAN	HP	OFF	RED	UNIX	

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 and 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 is a regular numetic addition, or one can write 'PolyObj1 + PolyObj2' in which the plus sign is now the Boolean operation of union between two geometric objects. The exact way each operator is overloaded is defined below.

In this environment the representation of reals, integers, and even Boolean data is identical. Data is automatically promoted as necessary. The constants TRUE and FALSE are defined as 1.0 and 0.0 respectively, for example.

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 )
        )
    );
```

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

The parser is almost totally case insensitive (with one exception - see the ALIAS command) so Obj, obj and OBJ will refer to the same object while MergePoly, MERGEPOLY, and margePoly 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 10 characters. By assigning to an old object, the old object will be automatically deleted and if necessary its type will be changing on the fly. For example:

[1] In fact the C language do 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 of 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 its dot product.

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

9 Operator overloading

This section lists the way the basic operators +, −, ∗, /, and ^ are overloaded. In other words, what action is taken by each of these operators depending upon its arguments.

9.1 Overloading +

The + operator is overloaded above the following domains:

```
NumericType + NumericType -> NumericType
VectorType  + VectorType  -> VectorType
MatrixType  + MatrixType  -> MatrixType
PolygonType + PolygonType -> PolygonType  (Boolean UNION operation)
CurveType   + CurveType   -> CurveType    (Curve curve chaining)
CurveType   + CtlPtType   -> CurveType    (Curve control point chaining)
CtlPtType   + CtlPtType   -> CurveType    (Control points chaining)
ListType    + ListType    -> ListType     (Append lists operator)
```

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 object are also disjoint - this system only tests for no intersection.

9.2 Overloading −

The − operator is overloaded above the following domains:

As a Diadic operator:

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

As a Monadic operator:

```
- NumericType -> NumericType
- VectorType  -> VectorType    (Scaling all vector by -1)
- MatrixType  -> MatrixType    (Scaling all 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 opeartion (Monadic 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  * VectorType    -> NumericType    (Inner product)
MatrixType  * NumericType   -> MatrixType     (Matrix Scaling)
MatrixType  * VectorType    -> VectorType     (Vector transform)
MatrixType  * MatrixType    -> MatrixType     (Matrix multiplication)
MatrixType  * GeometricType -> GeometricType  (Object transform)
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.

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
MatrixType  ^ NumericType -> MatrixType   (Matrix to the power)
PolygonType ^ PolygonType -> PolygonType  (Boolean MERGE operation)
```

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

9.6 Assignments

Assignments are allowed as side effects, any place in any expressions: 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 same expression are a bad practice.

Any assignment which is not at top level, MUST be within parenthesis.

10 Function's Description

The function description below defines their calling sequence in ANSI C notation. Listed are all the functions the system knows about 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 )
```

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

That means that the area of a polygonal approximation to a sphere will be returned, not the sphere area.

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 angle circle.

10.1.7 COS

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

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

10.1.8 CPOLY

`NumericType CPOLY( PolygonType Object )`

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

10.1.9 EXP

`NumericType EXP( NumericType Operand )`

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

10.1.10 LN

`NumericType LN( NumericType Operand )`

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

10.1.11 LOG

`NumericType LOG( NumericType Operand )`

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

10.1.12 SIN

`NumericType SIN( NumericType Operand )`

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

10.1.13 SQRT

`NumericType SQRT( NumericType Operand )`

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

10.1.14 TAN

`NumericType TAN( NumericType Operand )`

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

10.1.15 VOLUME

`NumericType VOLUME( PolygonType Object )`

Return the volume of the given **Object** (in object units). Returned is the real volume of the polygonal object - not the object it might approximate.

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

10.2 GeometricType returning functions

10.2.1 ARC

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

An arc constructor between two end points **StartPos** and **EndPos** centered at **Center**. Arc will always be less than 180 degrees so the shortest arc path from **StartPos** to **EndPos** is selected. The case where **StartPos**, **Center**, and **EndPos** are colinear 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 X and Y axes at coordinate 1.

10.2.2 BOOLSUM

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

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 four curves end points 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 parameteric direction and **Crv3** and **Crv4** are the two in the other.

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 );
```

10.2.3 BOX

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

Creates a main planes parallel BOX polygonal object, defined by **Point** as base position, and **Dx, Dy, Dz** as BOX dimensions. Note 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.

10.2.4 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 the same type (E2, E3, P2, or P3). Example:

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

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

10.2.5 CBSPLINE

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

Creates a Bspline curve out of the provided control point list and the knot vector, with the specified order. **CtlPtList** is a list of control points, all of the same type (E2, E3, P2, or P3) defining the curve control polygon. The length of the **KnotVector** must be equal to the number of control points in **CtlPtList** plus **Order**. Example:

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

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

The knot vector list may be specified as **list(KV_OPEN)** or as **list(KV_FLOAT)** in which a uniform open or floating knot vector with the appropriate length is automatically constructed.

10.2.6 CDIVIDE

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

Subdivides a curve into two at the specified parameter value. **Curve** can be either a Bspline curve in which **Param** must be within Curve parametric domain or a Bezier curve in which **Param** must be in the 0 to 1 range.

Returned is a list of the two sub-curves. The individual curves may be extracted from the list using the NTH command. For example:

```
CrvLst = CDIVIDE( Crv, 0.5 );
Crv1 = nth( CrvLst, 1 );
Crv2 = nth( CrvLst, 2 );
```

subdivides the curve **Crv** at the parameter value 0.5.

10.2.7 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 **Curve** points. 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**.

10.2.8 CEVAL

```
CtlPtType CEVAL( CurveType Curve, NumericType Param )
```

Evaluates the provided **Curve** at the given **Param** value. **Param** should be with the curve domain if **Curve** is a Bspline curve, or between 0 and 1 if **Curve** is a Bezier curve. Returned control point has the same type as the **Curve** control points. Example:

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

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

10.2.9 CIRCLE

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

Construct a circle at the specified **Center** with the specified **Radius**. Returned circle is a Bspline curve of four piecewise Bezier 90 degree arcs. Circle is always parallel to the XY plane. Use the linear transformation routines to place the circle in the appropriate location.

10.2.10 CIRCPOLY

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

As in this solid modeler, open objects are allowed (although any two objects intersection should form a closed loop), a plane (open!) object also exists. The function defines a circular polygon with the given **Normal**, and includes the **Trans** point. It is implemented as a FINITE circle with resolution (see RESOLUTION) edges, radius **Radius** and center **Trans** on that plane. It is the user responsibility to make sure that any Boolean operation on it (or any other open object) will create closed loops as intersecting curves and/or open loops terminated on its boundary (no termination in the middle of it is allowed).

Alternative to the function are manual construction of the required plane as a single polygon using POLY, or construction of a flat ruled surface using RULEDSRF.

10.2.11 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 surface **Srf** control mesh.

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

Note this curve is *not* necessarily in the surface **Srf**. Example:

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

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

10.2.12 CON2

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

Create a truncated CONE geometric object, defined by **Center** as CONE main base center, CONE axes **Direction**, and CONE base radii **Radius1/2**.

Note **Direction** magnitude also sets the CONE height. **Direction** can be any 3D vector.

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

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

constructs a truncated cone based at the XY parallel plane Z = -1, top at plane Z = 3, and with radii of 2 and 1 respectively.

10.2.13 CONE

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

Create a CONE geometric object, defined by **Center** as CONE base center, CONE axes **Direction**, and CONE base radius **Radius**. Note **Direction** magnitude also sets the CONE height. **Direction** can be any 3D vector.

See RESOLUTION for accuracy of CONE approximation as polygonal model. Example:

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

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

See also CON2.

10.2.14 CONVEX

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

Coerce non convex polygons in **Object**, into convex one. 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 automatically coerced non convex input polygons to convex ones, using this same routine.

However some external tools (like irit2ray, poly3d-r and poly3d-h) requires convex polygons. This function may be used on the objects to provide that, just before they are being saved into data files. Example:

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

decompose non convex polygons into convex ones so that data file could be used by external tools requiring convex polygons.

10.2.15 CPOLY

```
NumericType CPOLY( PolygonType Polys )
```

Returns number of polygons in provided object. Example:

```
DumpLvl = 1
NumPolys = CPOLY( Obj );
NumPolys;
```

prints the number of polygons in object **Obj**.

10.2.16 CRAISE

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

Raise **Curve** to the **NewOrder** Order specified. Currently implemented for Bezier curves of any order and linear Bspline curve only. Example:

```
Crv = ctlpt( E3, 0.0, 0.0, 0.0 ) +
      ctlpt( E3, 0.0, 0.0, 1.0 ) +
      ctlpt( E3, 1.0, 0.0, 1.0 );
Crv2 = CRAISE( Crv, 4 );
```

raises the 90 degrees corner linear Bspline curve **Crv** to be a cubic.

10.2.17 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 and all should be within **Curve** parametric domain. If knot vector is to be replaced, the length of **KnotList** should be identical to the length of **Curve** knot vector. 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** at the three knots 0.25, 0.5, and 0.75.

10.2.18 CREGION

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

Extracts a subdomain of **Curve** between **MinParam** and **MaxParam**. Both **MinParam** and **MaxParam** should be within **Curve** parametric domain. Example:

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

extracts the subdomain of **Crv** from the parameter value 0.3 to the parameter value 0.6.

10.2.19 CROSSEC

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

Unfortunately, this feature is NOT implemented...

10.2.20 CSURFACE

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

Extract an iso parametric curve out of **Srf** in the specified **Direction** (ROW or COL) at the specified parameter value **Param**. **Param** must be in **Srf** parameter range in **Direction** direction. The returned curve is *in* **Srf**. See also CMESH. Example:

```
Crv = CSURFACE( Srf, COL, 0.15 );
```

extract an iso parametric curve in the COLumn direction at parameter value 0.15 from surface **Srf**.

10.2.21 CTANGENT

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

Computes the tangent vector to **Curve** at the parameter value **Param**. Example:

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

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

10.2.22 CTLPT

```
CPt = CTLPT( ConstantType E2, NumericType X, NumericType Y )
```

or

```
CPt = CTLPT( ConstantType E3, NumericType X, NumericType Y, NumericType Z )
```

or

```
CPt = CTLPT( ConstantType P2, NumericType W, NumericType X, NumericType Y )
```

or

```
CPt = CTLPT( ConstantType P3, NumericType W,
                              NumericType X, NumericType Y, NumericType Z )
```

Construct a single control point to be used in curves and surfaces construction. Four types of points may be constructed as follows:

E2	A two dimensional point with X and Y.
E3	A three dimensional point with X, Y, and Z.
P2	A two dimensional rational point at X/W, and Y/W.
P3	A three dimensional rational point at X/W, Y/W, and Z/W.

Example:

```
CPt = CTLPT( E3, 0.0, 0.0, 0.0 );
```

constructs an **E3** points at the origin.

10.2.23 CYLIN

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

Create a CYLINder geometric object, defined by **Center** as CYLIN base center, CYLIN axes **Direction**, and CYLIN base radius **Radius**. Note Direction magnitude also sets the CYLIN height. **Direction** can be any 3D vector. See RESOLUTION for accuracy of CYLIN approximation as polygonal model. Example:

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

constructs a cylinder along the X axis from the origin to X = 10.

10.2.24 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 constrcted. 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).

No limitation exists on the polygon (can be non-convex), but **Dir** cannot be coplanar with the polygon plane. The curve need not be planar. 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 a half of it in reverse and merging the two sub-curves. **Cross** is then used as the cross section for the extrusion operation.

10.2.25 GBOX

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

Create a parallelpiped - 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 special case of GBOX where **Dx** = vector(Dx, 0, 0), **Dy** = vector(0, Dy, 0), **Dz** = vector(0, 0, Dz).

Note **Dx**, **Dy**, **Dz** must be non-coplanar in order to create a feasible object. 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));
```

10.2.26 GPOLYGON

```
PolygonType GPOLYGON( GeometryTreeType Object )
```

Approximate all Surface(s) in **Object** as polygons using the RESOLUTION and FLAY4PLY variables. The larger RESOLUTION is the more polygons and finer the result approximation will be. Each Bezier patch will have roughly $RESOLUTION^2$ polygons.

FLAT4PLY is a Boolean flag controlling the conversion of an (almost) flat patch into four (TRUE) or two (FALSE) polygons. Normals are computed to polygons vertices using surface normal, so Guaroud or Phong rendering can be performed. Returned is a single polygon object, no matter how complex **Object** Hierarchy is. Example:

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

Converts to polygons the three surfaces **Srf1**, **Srf2**, and **Srf3**.

10.2.27 GPOLYLINE

```
PolylineType GPOLYLINE( GeometryTreeType Object )
```

Converts all Surface(s) and Curves(s) in **Object** into polylines using the RESOLUTION variable. The larger RESOLUTION is the finer the result approximation will be. Returned is a single polyline object, no matter how complex **Object** Hierarchy is. Example:

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

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

10.2.28 MERGPOLY

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

Merge a set of polygonal objects in **PolyList** list to a single polygonal object. All elements in **ObjectList** must be of PolygonType type. No test is made on the validity of the data. This function performs the same operation as the ^ operator would, but may be more convenient to use under some conditions. 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 ) );

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

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 ) );

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

10.2.29 OFFSET

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

or

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

Offsets **Crv** or **Srf**, by translating all control points in the curve or surface normal, by the **OffsetDistance** amount. Returned curve or surface only approximates the real offset. One may improve the offset accuracy using refinement. Negative **OffsetDistance** denotes offset in the reversed direction of the normal.

Example:

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

offsets **Crv** by the amount of -0.1 in the reversed normal direction.

10.2.30 POLY

```
PolygonType POLY( ListType ObjectList )
```

Create a single polygon (and therefore open) object, defined by the vertices which respectively defined by the objects in **ObjectList** (see LIST). All elements in **ObjectList** must be of VertorType type. No validity test is made and it is the user responsibility. see CIRCPOLY for conditions applied to open objects. 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 ) );
```

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

10.2.31 RULEDSRF

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

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

Example:

```
Circ = circle( vector( 0.0, 0.0, 0.0 ), 0.25 );
Cyl = RULEDSRF( circ, circ * trans( vector( 0.0, 0.0, 1.0 ) ) );
```

Constructs a cylinder of radius 0.25 along the Z axis from 0 to 1.

10.2.32 SBEZIER

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

Creates a Bezier surface out of the provided control mesh. **CtlMesh** is a list of control points rows, each is a list of control points. All control points must be of the same type (E2, E3, P2, or P3).

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 ) ),
                        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 ) ) ) );
```

10.2.33 SBSPLINE

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

Creates a Bspline surface out of the provided **UOrder** and **VOrder** orders, the control mesh **CtlMesh** and the two knot vectors **KnotVectors**. **CtlMesh** is a list of control points rows, each is a list of control points. All control points must be of the same type (E2, E3, P2, or P3). **KnotVectors** is a list of two knot vectors, each is a list of NumericType knots or a list of a single constant KV_OPEN or KV_FLOAT in which a uniform knot vector with open or floating end condition will automatically be constructed. 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 Bspline surface with its first knot vector being uniform with open end condition.

10.2.34 SDIVIDE

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

Subdivides a surface into two at the specified parameter value **Param** in the specified **Direction** (ROW or COL). **Srf** can be either a Bspline curve in which **Param** must be within surface parametric domain or a Bezier curve in which **Param** must be in the 0 to 1 range.

Returned is a list of the two sub-surfaces. The individual surfaces may be extracted from the list using the **NTH** command. 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.

10.2.35 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 **Srf** control mesh by substituting **CtlPt** instead. CtlPt must have the same point type as **Srf** points. 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**.

10.2.36 SEVAL

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

Evaluates the provided surface **Srf** at the given **UParam** and **VParam** values. Both **UParam** and **VParam** should be within the surface parametric domain if **Srf** is a Bspline surface, or between 0 and 1 if **Srf** is a Bezier surface. Returned control point has the same type as **Srf** control points. Example:

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

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

10.2.37 SFROMCRVS

```
SurfaceType SFROMCRVS( ListType CrvList )
```

Construct a surface by substituting the curves in **CrvList** as rows in a surface control mesh. Curves in **CrvList** are made compatible by promoting Bezier curves to Bsplines if necessary and raising degree and refining as required before substituting their control polygons as rows in the mesh.

The other direction order is the same as the first direction order or if not enough curves are provided equal to the number of curves in **CrvList**.

The surface will interpolate 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 * scale( vector( 0.0, 0.0, 2.0 ) )
            * trans( vector( 0.1, 0.1, 0.1 ) );
Srf = SFROMCRVS( list( Crv1, Crv2, Crv3 ) );
```

10.2.38 SNORMAL

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

Computes the normal vector to **Srf** at the parameter values **UParam** and **VParam**. Example:

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

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

10.2.39 SPHERE

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

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

10.2.40 SRAISE

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

Raise **Srf** to the specified **NewOrder** Order in the specified direction **Direction**. Currently implemented for Bezier surfaces of any order and linear Bspline surfaces only. 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 );
```

construct a bilinear flat ruled surface and raise its both directions to be quadratic.

10.2.41 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 within **Srf** parametric domain in **Direction** direction. If knot vector is to be replaced, the length of **KnotList** should be identical to the length of **Srf** knot vector in direction **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

10.2.42 SREGION

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

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

```
   SubSrf = SREGION( Srf, COL, 0.3, 0.6 );
```

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

10.2.43 STANGENT

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

Computes the tangent vector to **Srf** at the parameter values **UParam** and **VParam** in the direction **Direction**. 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).

10.2.44 SURFREV

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

or

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

Create a surface of revolution by rotating the first polygon/curve of the given **Object**, around the Z axes. No limitation exists on the polygon (can be non-convex), aside from the requirement for it to be non coplanar with a plane of the form Z = Const. No limitation exists for the curve. 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 use it to construct a surface of revolution by rotating it around the Z axis.

10.2.45 SWEEPSRF

```
SurfaceType SWEEPSRF( CurveType CrossSection, CurveType Axis,
                                                        NumericType Scale )
```

or

```
SurfaceType SWEEPSRF( CurveType CrossSection, CurveType Axis,
                                                        CurveType ScaleCrv )
```

Construct a generalized cylinder surface. This function sweeps a specified cross section **Cross-Section** along the provided **Axis**. The cross section may be constantly scaled (first form above), or scaled along the Axis parametric domain (second form).

No refinement is performed on any of the curves so scaling and axis following result is only approximated. Refinement at the proper location should improve the output accuracy. **ScaleCrv** parametric domain do not have to match the **Axis** parametric domain and their domains are 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( Arc1, FALSE, list( 0.25, 0.5, 0.75 ) );
   ScaleCrv = cbezier( list( ctlpt( E2, 0.0, 0.01 ),
                             ctlpt( E2, 1.0, 0.5 ),
                             ctlpt( E2, 2.0, 0.01 ) ) );
   Srf = SWEEPSRF( Cross, Axis, ScaleCrv );
```

constructs a rounded rectangle cross section and sweep it along an arc while scaling it so its end points shrink. Note the axis curve **Axis** is manually refined to better approximate the scaling required.

10.2.46 TORUS

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

Create a TORUS polygonal object, defined by **Center** as TORUS center, **Normal** as main TORUS plane normal, **MRadius** as major radius, and **mRadius** as minor.

See RESOLUTION for accuracy of TORUS approximation as 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.

10.3 Object transformation functions

All the routines in this section constructs a 4 by 4 homogeneouos matrix representing the required transform. These matrices may be concatenated to achieve a 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 transform can be computed as:

```
InvM = m ^ -1
```

See also overloading the - operator.

10.3.1 ROTX

```
MatrixType ROTX( NumericType Angle )
```

Creates a rotation transformation matrix (around X) with **Angle** degrees.

10.3.2 ROTY

```
MatrixType ROTY( NumericType Angle )
```

Creates a rotation transformation matrix (around Y) with **Angle** degrees.

10.3.3 ROTZ

```
MatrixType ROTZ( NumericType Angle )
```

Creates a rotation transformation matrix (around Z) with **Angle** degrees.

10.3.4 SCALE

```
MatrixType SCALE( VectorType ScaleFactors )
```

Creates a scaling transformation matrix of **ScaleFactors** scaling factors.

10.3.5 TRANS

```
MatrixType TRANS( VectorType TransFactors )
```

Creates a translation transformation matrix of **TransFactors** translating amounts.

10.4 General purpose functions

10.4.1 ALIAS

```
ALIAS( StringType Name, StringType Value )
```

Defines a text substitution: each occurrence of **Name** will be replaced by the given **Value**. Unlike the rest of the system, this is CASE SENSITIVE. It is a good practice, therefore, to defines the aliases names to be upper case, and rest of program including alias values in lower case. For example:

```
ALIAS("ED", "edit(\"file.irt\");");
```

defines the alias "ED" to be 'edit("file.irt");'. Note the way the double quotes are being escaped.

Using "ed" instead of "ED" above will cause infinite loop since "ed" will be expanded for ever... The aliases will be expanded until line is too long or 100 expansions occurred in line.

If **Name** is empty string, a list of all defined aliases is printed.

If **Name** is not empty, but **Value** is, that alias is deleted. This is the only case you need to specify the alias **Name** in LOWER case (otherwise it will be expanded...) - the alias **Name** comparison is case insensitive.

10.4.2 ATTRIB

```
ATTRIB( GeometricType Object, StringType Name, StringType Value )
```

Provides a mechanism to add a string attribute to a geometric **Object**, with name **Name** and value **Value**.

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");
```

sets the rendered color of the **Glass** object. This specific attribute provides a finer control on color setting than provided by the color command, for external programs.

10.4.3 BEEP

```
BEEP( NumericType Frequency, NumericType Time )
```

Generates a tone with the given **Frequency** (in Hz), for the given period of **Time** (in milliseconds). This command is system dependent and may work differently or not work at all on some systems.

10.4.4 CHDIR

```
CHDIR( StringType NewPath )
```

Change current working directory to NewPath (if exists). The entry directory is recovered on exit from program.

10.4.5 CLOSED

```
CLOSED( NumericType Set )
```

If **Set** is non zero (see TRUE/FALSE and ON/OFF) then every polygonal object drawn is assumed to be closed. If a polygonal model is closed every edge is basically drawn twice - once for each adjacent polygon. If the object is assumed closed, every such edge will be drawn once only. By default this option is TRUE.

10.4.6 COLOR

```
COLOR( GeometricType Object, NumericType Color )
```

Set the color of the object to one of the specified below. Note that an object has a default color (see IRIT.CFG file) according to his origin - loaded with LOAD command, PRIMITIV, or BOOLEAN operation result. The system internally supports colors (although you may have B&W system) and the colors recognized are: **BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, YELLOW, and WHITE**.

See attrib command for more fine control on colors.

10.4.7 COMMENT

```
COMMENT
```

Two types of comments are allowed:

1. One lines comment: starts anywhere is a line at the '#' char up to the end of the line.

2. Block comment: starts by the COMMENT keyword follows 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. For example:

```
COMMENT $
  This is a comment
$
```

10.4.8 DIR

```
DIR( StringType MatchPattern )
```

Print the files match the MatchPattern in the current working directory. MatchPattern may have wild characters as in regular dos DIR - '*', '?'.

This command is only supported under the MSDOS implementation. Example:

```
DIR( "*.irt" );
```

lists all the '.irt' files in the current directory.

10.4.9 EDIT

```
EDIT( StringType FileName )
```

Invoke the editor (defined in the IRIT.CFG configuration file) as a child process if the solid modeler. Only one parameter is passed to the editor which is the FileName to edit. As the solid modeler is still resident, the child process (the editor) will get only the remained memory - as seen by the core left.

This command is only supported under the MSDOS implementation.

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 )
```

Execute the **Body** (see below), while the FOR loop conditions hold. **Start, Increment, End** are evaluated first and the loop will be 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 arbitrary level. No new variables should be introduced in loops - Use only old variables and/or the iteration variable defined in **Start** (that makes it a feature now...). Example:

```
FOR ( (b = 100), 100, 300,
    FOR ( (a = 100), 100, 2000,
            (
              beep(a, b)
            )
        )
    );
```

exercises the BEEP function in different durations and different frequencies. This will best work under MSDOS systems.

10.4.12 FREE

```
FREE( GeometricType Object )
```

Because of the usually huge size of geometric objects, this procedure may be used to free them. Note however that reassigning a value (even of different type) will automatically release old allocated space as well.

10.4.13 HELP

```
HELP( StringType Subject )
```

Provides help on the specified Subject. Example:

```
HELP("");
```

will list all *IRIT* commands.

10.4.14 IF

```
IF( NumericType Left, StringType Cond, NumericType Right, AnyType Body )
```

Executes **Body** (group of regular commands, separated by COLONs - see FOR) if the condition holds: **Left** and **Right** are evaluated and tested against the specified condition **Cond** which may be: "=", ">", "<", "<>", ">=", "<=". Example:

```
resolution = 10;
IF ( machine, "=", msdos, ( resolution = 5 ) );
```

sets the resolution to be 10 unless running on an MSDOS system in which the resolution variable will be set to 5.

10.4.15 INCLUDE

```
INCLUDE( StringType FileName )
```

Executes the script file **FileName**. Nesting of include file is allowed up to 10 levels deep. If error occurs, all open files in all nested files are closed and data is expected 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".

10.4.16 INTERACT

```
INTERACT( GeometryTreeType Object, NumericType UpdateViewMat )
```

Invoke interactive mode to manipulate (transform) the given (geometric) **Object**. **Object** may be any GeometricType or a list of other GeometryTypes nested to an arbitrary level. **Object** is displayed as a wire frame. ON SGI 4D systems a rendered display is also available.

If **UpdateViewMat** is non zero (see TRUE/FALSE and ON/OFF) then the global viewing matrix VIEW_MAT is updated to the last view from INTERACT.

INTERACT open an interactive menu to rotate/translate/scale an object(s). Each transformation has zero influence in middle of its box, and maximum (and opposite) on the box left and right ends.

Screen transformation transforms according to the screen - X is horizontal, Y vertical, Z into screen. Object transformation transform in object own coordinate system - you probably want to display AXES object with it (see AXES).

Left mouse button (return) is used to EXECUTE transformation. Right mouse button (space bar) is used to ABORT long display in the middle.

On MSDOS, use the numeric keyboard pad (with/without shift) to move if no mouse available.

If **Object** is a GeometricType, that object is being displayed. If however it is an ListType the list is recursively traversed and all geometric objects within the list are displayed. Non geometric object in the list are ignored.

```
INTERACT( list( Axes, Obj ), FALSE );
```

displays and interact with the object **Obj** and the predefined object **Axes**. VIEW_MAT will not be updated once INTERACT is done.

10.4.17 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 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 will remove the variable, but the object itself will be freed only when that list will be freed.

10.4.18 LOAD

```
AnyType LOAD( StringType FileName )
```

Load the object from the given **FileName**. The object may be any object defined in the system, including lists, in which the structure is loaded recursively and reconstructed as well (internal objects are inserted into the global system object list if have names defined). If no file type is provided, ".dat" is assumed.

10.4.19 LOGFILE

```
LOGFILE( NumericType Set )
```

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 IRIT.CFG configuration file. This file will be created the first time logfile is turned ON.

10.4.20 NORMAL

```
NORMAL( NumericType Set, NumericType Size, NumericType Color )
```

If **Set** is non zero (see TRUE/FALSE and ON/OFF) then the normals to the objects are also displayed. Normals should always point INTO the object.

Size sets the length of the normals, and **Color** their color. See COLOR command for legal colors.

10.4.21 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)'.

10.4.22 PAUSE

```
PAUSE( NumericType Flush )
```

Waits for a keystroke. Nice to have if temporary stopping in middle of included file (see INCLUDE) is needed. If **Flush** is TRUE then the input is first flushed to guarantee we will wait.

The implementation of this function is machine dependent and is geared mainly for MSDOS.

10.4.23 SAVE

```
SAVE( StringType FileName, AnyType Object )
```

Saves the provided **Object** into specified file **FileName**. No extension type is needed (ignored if specified), and ".dat" is always used. **Object** can be any object type including list, in which structure is saved recursively. See also LOAD.

10.4.24 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)'.

10.4.25 SYSTEM

```
SYSTEM()
```

Invoke the current command processor (usually COMMAND.COM) as defined by the COMSPEC environment variable. As the solid modeler is still resident, the child process (the command processor) will get only the remained memory - as seen by the core left.

This command is only supported under the MSDOS implementation.

10.4.26 TIME

```
TIME( NumericType Reset )
```

Returns the real time (in seconds) from the last time TIME was called with **Reset** TRUE. Note this is real time and not cpu time so running in a multi tasked system will return values, which not necessarily reflects this program execution time. As mentioned above if **Reset** is non zero the time count is reset. The time is automatically reset to beginning of execution of this program, when the program is first invoked.

Since this is real type, it may be unusable for multitasked systems. Example:

```
Time1 = TIME( TRUE );
    .
    .
    .
Time1 = TIME( FALSE );
DumpLvl = 1;
Time1;
```

prints the time in seconds between the above two time function calls.

10.4.27 VARLIST

```
VARLIST()
```

List all the currently defined objects in the system.

10.4.28 VECTOR

```
VectorType VECTOR( NumericType Operand1, NumericType Operand2,
                                                NumericType Operand3 )
```

Creates a vector type object, out of 3 NumericType scalars.

10.4.29 VIEW

```
VIEW( GeometricTreeType Object, NumericType ClearWindow )
```

Display the (geometric) object(s) as given in **Object**. See INTERACT for more on **Object**.
If **ClearWindow** is non zero (see TRUE/FALSE and ON/OFF) the window is first cleared (before drawing the objects).
The global viewing matrix VIEW_MAT is used as the transformation matrix. Example:

```
VIEW( Axes, FALSE );
```

displays the predefined object **Axes** in the viewing window on top of what was drawn there.

10.5 System variables

System variables are predefined objects in the system. Any time *IRIT* is executed, these variable will exist and be 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 by the use of the IRITINIT.IRT file.

10.5.1 AXES

Predefined polyline object (PolylineType) that holds XYZ axes system. May be viewed.

10.5.2 DRAWCTLPT

Predefined Boolean variable (NumericType) controlling whether curves control polygons and surfaces control meshes will be drawn (TRUE) or not (FALSE).

10.5.3 DUMPLVL

Content of objects assigned to variables may be displayed by executing the command 'objname;' where objname is the name of the object. This variable (NumericType) control the way the data is dumped as follows:

DumpLvl >= 0	Only object names/types are printed.
DumpLvl >= 1	Non geometric object values are dumped.
DumpLvl >= 2	Curves and Surfaces are dumped.
DumpLvl >= 3	Polygons/lines are dumped.
DumpLvl >= 4	List objects are traversed recursively.

10.5.4 ECHOSRC

Predefined Boolean variable (NumericType) controlling echoing of interpreted commands to screen (TRUE) or not (FALSE).

10.5.5 FLAT4PLY

Predefined Boolean object (NumericType) controlling the way almost flat surface patches are converted to polygons: four polygons (TRUE) or only two polygons (FALSE).

10.5.6 INTERCRV

Predefined numeric object (NumericType) that if TRUE the Boolean geometry operators return the intersection curves instead of the result model.

Its default value is FALSE.

10.5.7 INTERNAL

Predefined Boolean object (NumericType) that if not zero enables displaying internal polygon edges (edges created by the convex polygon splitting for example). One usually does not want to see these edges, and its default value is FALSE.

10.5.8 MACHINE

Predefined numeric object (NumericType) holding machine type as one of the following constants: MSDOS, SGI, HP, SUN, APOLLO, UNIX.

10.5.9 RESOLUTION

Predefined numeric object (NumericType) that sets the accuracy of the primitive geometric objects generated. Holds the number of divisions a circle is divided into (with minimum value of 4). If, for example, is set to 6, then a CONE generated, will effectively be 6 sided pyramid.

Also controls the fineness freeform curves and surfaces are approximated as piecewise linear polylines (for display purposes for example), and the fineness freeform surfaces 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.

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 situation you would like to know their values, here is a sequence that will allow you to do so:

```
DumpLvl = 1;
A = BLUE;
A;
```

in other words, assign the constant to a variable and display its content.

10.6.1 APOLLO

A constant designates an APOLLO system in the MACHINE variable.

10.6.2 BLACK

A constant defining a BLACK color.

10.6.3 BLUE

A constant defining a BLUE color.

10.6.4 COL

A constant defining the COLumn direction of a surface mesh.

10.6.5 CYAN

A constant defining a CYAN color.

10.6.6 E2

A constant defining an E2 (X and Y coordinates) control point type.

10.6.7 E3

A constant defining an E3 (X, Y, and Z coordinates) control point type.

10.6.8 FALSE

A zero constant. May be used as Boolean operand.

10.6.9 GREEN

A constant defining a GREEN color.

10.6.10 HP

A constant designates an HP system in the MACHINE variable.

10.6.11 KV_FLOAT

A constant defining a floating end condition uniform knot vector.

10.6.12 KV_OPEN

A constant defining an open end condition uniform knot vector.

10.6.13 MAGENTA

A constant defining a MAGENTA color.

10.6.14 MSDOS

A constant designates an MSDOS system in the MACHINE variable.

10.6.15 OFF

Synonym of FALSE.

10.6.16 ON

Synonym for TRUE.

10.6.17 P2

A constant defining an P2 (X, Y, and W coordinates) rational control point type.

10.6.18 P3

A constant defining an P3 (X, Y, Z, and W coordinates) rational control point type.

10.6.19 PI

The constant of 3.141592...

10.6.20 RED

A constant defining a RED color.

10.6.21 ROW

A constant defining the ROW direction of a surface mesh.

10.6.22 SGI

A constant designates a SGI system in the MACHINE variable.

10.6.23 SUN

A constant designates a SUN system in the MACHINE variable.

10.6.24 TRUE

A non zero constant. May be used as Boolean operand.

10.6.25 UNIX

A constant designates a generic UNIX system in the MACHINE variable.

10.6.26 WHITE

A constant defining a WHITE color.

10.6.27 YELLOW

A constant defining a YELLOW color.

11 poly3d - A Data Display Program

11.1 Introduction

poly3d is a display program for data files created by the *IRIT* solid modeler. Data can be displayed on almost any IBMPC graphic device that has Borland's BGI support, for the MSDOS BC++ port, or using any device driver that is supported by DJGPP, in the MSDOS DJGPP port. Under UNIX systems both MIT's X11 and SGI's GL are supported.

On SGI's, solid smooth shading is available via normal computation support. Displayed images may be saved as postscript files as well as GIF images (if poly3d was built with GIF support).

11.2 Command Line Options

```
poly3d [-c] [-m] [-i] [-e #Edges] [-n] [-N] [-M] [-I n] [-P] [-S n]
                                                [-f FineNess] [-4] [-z] DFiles
```

- **-c**: Closed object - if an object is closed (such as objects created by *IRIT* solid medeller) each edge is actually displayed twice - once for each adjacent polygon. This flag will ensure every edge will be displayed only once.
- **-m**: More - provide 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**: # Edges to use from each given polygon (default all). Very handy to do '-e 2 -4-' or '-e 1 -4' on polygons created from a freeform surface.
- **-n**: Draw vertex normals if data file has them.
- **-N**: Draw polygon normals if data file has them (PLANE definition).
- **-M**: Draw the surfaces control mesh/curves control polygons as well.
- **-I n**: Specify number of isolines per surface.
- **-P**: Generate polygonal approximation for surfaces instead of isolines.
- **-S n**: Specify the log based 2 of number of samples per curve.
- **-f FineNess**: Controls the fineness of the surface to polygon subdivision. This number is log based 2 of roughly the number of subdivisions of the surface in each axes (see cagd_lib for more).
- **-4**: Force four polygons per flat bilinear in surface to polygon conversion. Otherwise two polygons only.
- **-z**: Print version number and current defaults.

Some of the options may be turned on in poly3d.cfg. They can be then turned off in the command line as -?-.

11.3 Configuration

The program can be configured using a configuration file named poly3d.cfg. The appropriate configuration file should be copied into poly3d.cfg: Under MSDOS BC++ its poly3dms.cfg, under MSDOS DJGPP its poly3ddj.cfg, and under UNIX its poly3dun.cfg. This is a plain ascii file you can edit directly and set the parameters according to the comments there. Executing 'poly3d -z' will show the current configuration as read from the configuration file.

MSDOS version only:

The configuration file in MSDOS BC++ system can be in any directory which is in your path - the same place as the executable is probably a good choice. The program supports SuperVGA, VGA/EGA, CGA & HERCULES graphics card, and uses the Turbo C autodetect feature. If this fails you can always coerce it to a specific card - see the poly3d.cfg file. For a SuperVGA you will need to provide your own BGI driver. The program will use 80x87 if it detects one - again uses the Turbo C 80x87 autodetect, or will run (much!) slower without it... The MSDOS DJGPP port can be used with any graphics driver that DJGPP support. If not 80x87 is present, the emulator, emu387, must be used. In both the BC++ and DJGPP ports, a mouse or a joystick may be used if properly selected in poly3d.cfg configuration file. A windowing library, called intr_lib is used (for both BC++ and DJGPP) for the interaction and windows which can be resized/moved/poped/pushed etc. via a default setting as selected by the configuration file and modified via a pop up menu during a session. see Irit solid modeler for more on intr_lib and the interface.

UNIX version only:

The configuration file is being searched in the IRIT_PATH environment variable under UNIX hosts. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. Note IRIT_PATH must terminate with '/'. If the variables is not set only the current directory is being searched. Add the following options to your *.Xdefaults* if you use X11. Most options set are self explanatory. The Trans attributes control the transformation window, while View control the view window. SubWin control the subwindows within the Transformation window:

poly3d*Trans*BackGround:	NavyBlue
poly3d*Trans*BorderColor:	Red
poly3d*Trans*BorderWidth:	3
poly3d*Trans*TextColor:	Yellow
poly3d*Trans*SubWin*BackGround:	DarkGreen
poly3d*Trans*SubWin*BorderColor:	Magenta
poly3d*Trans*Geometry:	=150x500+500+0
poly3d*Trans*CursorColor:	Green
poly3d*View*BackGround:	NavyBlue
poly3d*View*BorderColor:	Red
poly3d*View*BorderWidth:	3
poly3d*View*Geometry:	=500x500+0+0
poly3d*View*CursorColor:	Red

Note the above options are the same as for the irit solid modeler itself.

If poly3d is used under SGI gl library, you can set the prefered windows location in poly3d.cfg. No color control is provided at this time.

11.4 Usage

The program is controlled via a transformation menu. The object can be rotated, translated, or scaled in screen or object based corrdinate system and with orthographic or perspective projected.

Two operations are fundamental to the operation of poly3d:

EXECUTE	<Return> key on the keyboard, or left mouse button if exists.
ABORT	<Space> key on the keyboard, or right mouse button if exists.

The cursor may be moved via a mouse (if exists) or using the numeric keypad/arrows (shifted for faster movement). The ABORT may be useful in large data sets when another transformation should be applied with no need to wait to the current one to complete.

The menus work in two modes in all implementations. If you click the EXECUTE button once on a transformation subwindow, that transformation will be applied. Most transformation subwindows have vertical bars in the middle. If the cursor is on the vertical bar the *amout* of transformation applied is *zero.* If the cursor is on the leftmost side of the subwindow, the *amount* will be maximized and if the cursor is on the rightmost side, the transformation *amount* will be maximized to the other direction (inverse transform). In addition, if one clicks the mouse (with the affect as above) but hold the EXECUTE button while dragging, continuous transformation will be applied. Altough implemented in all systems, you would probably like to try this on very simplistic models on slow machines.

Keyboard control support is for MSDOS systems only.

11.5 Output Files

poly3d can save viewing matrices (to be used by poly3d-h, poly3d-r, or irit2ray for example), postscript files of the current view and GIF images of the screen if poly3d was built with GIF support.

The postscript file can be directly sent to a laser printer. The viewing matrix should be appended *after* the data when any program is to use it. The last viewing matrix in the data will be the one used. For example

```
irit2ray -l -f 20 b58.dat generic0.mat
```

where generic0.mat is the matrix saved bu poly3d.

All files will be named genericX.EXT where EXT can be one of '.mat', '.ps' and '.gif' respectively. X is single digit so up to 10 distinguished files of each type can be saved each time.

12 poly3d-h - Hidden Line Removing Program

12.1 Introduction

poly3d-h is a program to remove hidden line of a given polygonal model. Freeform objects are preprocessed into polygons with controlled fineness.

The program performes 4 passes over the input:

1. Preprocesses and maps all polgons in scene, and sorts them.

2. Generates edges out of the polygonal model and sorts them (preprocesing 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 programs can handle CONVEX polygons only. From *IRIT* one can ensure a model is consisting of convex polygons only by the CONVEX command:

```
 CnvxObj = convex( Obj );
```

just before saving it into a file. Surfaces are always decomposed into triangles only.

poly3d-h output is of the form of polylines. It is a regular *IRIT* data file that can be viewed using poly3d, for example.

12.2 Command Line Options

```
poly3d-h [-b] [-m] [-i] [-e #Edges] [-f FineNess] [-4] [-z] DFiles [> OutFile]
```

- **-b**: BackFacing - if object is closed (such as most models created by *IRIT*) back facing polygons will be deleted, and therefore speed up the process by at list factor of two.
- **-m**: More - provide 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**: # Edges to use from each given polygon (default all). Very handy as '-e 1 -4' for freeform data.

- **-f FineNess**: Controls the fineness of the surface to polygon subdivision. This number is log based 2 of roughly the number of subdivisions of the surface in each axes (see cagd_lib for more).
- **-4**: Force four polygons per flat bilinear in surface to polygon conversion. Otherwise two polygons only.
- **-z**: Print version number and current defaults.

Some of the options may be turned on in poly3d-h.cfg. They can be then turned off in the command line as -?-.

12.3 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. executing 'poly3d-h -z' will show the current configuration as read from the configuration file.

UNIX and MSDOS DJGPP versions only:

The configuration file is being searched in the IRIT_PATH environment variable. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. Note IRIT_PATH must terminate with '/'. If the variables is not set only the current directory is being searched.

MSDOS BC++ version only:

The configuration file in MSDOS system can be in any directory which is in your path - the same place as the executable is probably a good choice. The program will use 80x87 if it detects one - uses the Turbo C 80x87 autodetect, or will run (much!) slower without it...

12.4 Usage

As this program is not interactive, usage is quite simple, and only control available is using the command lines options.

13 poly3d-r - A Simple Data Rendering Program

13.1 Introduction

poly3d-r is a simple rendering program for data files created by the *IRIT* solid modeler. poly3d-r generates GIF images with 8 bits per pixel. As a result rendered images are of medium quality. Although reasonably fast, one should use one of several raytracing public domain programs available (such as RAYSHADE which irit2ray can generate data to) for high quality images.

poly3d-r uses cosine shading approximation, and flat/Gouraud interpolation. The program performes 4 passes over the input:

1. Process the input (parsing.)

2. Prepare the polygons by sorting them by their Y after mapping then into screen space.

3. Evaluate colors for vertices (using polygon normals if flat shading, or by vertex normals for Gouraud shading).

4. Scan the data by scan line basis and dump out image.

This programs can handle CONVEX polygons only. From *IRIT* one can ensure a model is consisting of convex polygons only by the CONVEX command:

```
CnvxObj = convex( Obj );
```

just before saving it into a file. Surfaces are always decomposed into triangles only.

13.2 Command Line Options

```
poly3d-r [-a Ambient] [-c N] [-l X Y Z] [-2] [-m] [-s Xsize Ysize]
      [-S SubSample] [-g] [-b] [-M Mask] [-f FineNess] [-z] DFiles > Image.gif
```

- **-a Ambient**: Sets the ambient intensity (must be in [0.0..1.0] range).
- **-c N**: number of bits per pixel N (must be in [1..8] range).
- **-l X Y Z**: specify the light source normal direction. This vector does not to be unit vector. Only one light source is supported.
- **-2** : Force emulation of 2 light sources at opposite directions as selected via [-l]. This may be useful for models that has no plane specified (i.e. model has no PLANE attribute for its polygons), as the program guess the equation from the points themselves, and which can be to the opposite direction.
- **-m**: More - provide some more information on the data file(s) parsed.
- **-s Xsize Ysize**: specify image dimensions. As the models created by *IRIT* are mapped to a unit domain (X in [-1..1], Y in [-1..1]) by the viewing matrix, objects must be scaled up. The scaling up factor is MIN(Xsize, Ysize), which guarantee nothing of the original image will be clipped.
- **-b**: Purge back facing polygons. If the scene contains closed objects (such as the ones usually created by *IRIT*), the back facing polygons can be deleted. This would not change the image, but will speed up the process at about objects with polygons with no PLANE specified would almost definitely create wrong image.
- **-g**: Use Gouraud shading interpolation (flat shading is used by default). This is somewhat slower, but gives nicer results.
- **-S SubSample**: sub sample, and uses box filter to low pass filter the image, using SubSample as grid side of SubSample by SubSample. This obviously make things slower, but guess what - it looks much better.
- **-M Mask**: Create a new GIF file named Mask that is a binary image set to 1 at any pixel covered by one of the objects or 0 otherwise. As a color of an object can become equal to the background at some point, there is no way to find whether pixel is background or an object in the background color. The Mask can be used instead. This Mask can be used when combining images (such as gifcomp utility in the gif_lib). This image is a binary alpha channel.
- **-f FineNess**: Controls the fineness of the surface to polygon subdivision. This number is log based 2 of roughly the number of subdivisions of the surface in each axes (see cagd_lib for more).
- **-z**: Print version number and current defaults.

The image is dumped to stdout as a GIF image which can be redirected to a file or to be piped to any program that reads GIF images from stdin.

Some of the options may be turned on in poly3d-r.cfg. They can be then turned off in the command line as -?-.

13.3 Configuration

The program can be configured using a configuration file named poly3d-r.cfg. This is a plain ascii file you can edit directly and set the parameters according to the comments there. executing 'poly3d-r -z' will show the current configuration as read from the configuration file.

UNIX and MSDOS DJGPP versions only:

The configuration file is being searched in the IRIT_PATH environment variable. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. Note IRIT_PATH must terminate with '/'. If the variables is not set only the current directory is being searched.

MSDOS BC++ version only:

The configuration file in MSDOS system can be in any directory which is in your path - the same place as the executable is probably a good choice. The program will use 80x87 if it detects one - uses the Turbo C 80x87 autodetect, or will run (much!) slower without it...

13.4 Usage

As this program is not interactive, usage is quite simple, and only control available is using the command lines options.

Some Remarks:

1. If the input file is degenerate (2 vertices are identical etc.) they will be ignored is the next passes. Use [-m] if you want to know about them.

2. The color of the object can be extract via the COLOR attribute as set via the *IRIT* COLOR command. In addition to this fixed set of colors, one can specify the color in RGB space using the ATTRIB command. For example:

```
    attrib( Srf17, "rgb", "255,155,55" );
```

Each of R G B must be integer is the range [0..255].

14 Irit2Ray - IRIT To RAYSHADE filter

14.1 Command Line Options

```
 irit2ray [-l] [-4] [-G GridSize] [-f FineNess] [-o OutName] [-g] [-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. Not using this option will better emulate the surface shape but will create unnecessary polygons in cases where one is enough.

- **-4**: Four - Generate 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 RAYSHADE *grid* constructure, with *GridSize* being used as the grid size along the object bounding box largest dimension.

- **-f FineNess**: An integer value controling the fineness of surface to polygons process. Roughly speaking it will set to the number of polygons along one Bezier patch direction. A Bezier patch will have order of $FineNess^2$ polygons then. The Order of the surface also affect the amount of polygons; The higher the order is, more polygons are created. A B-spline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.

- **-o OutName**: Name of output files. 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.
- **-z**: Print version number and current defaults.

14.2 Usage

Irit2Ray converts freeform surfaces into polygons in format that can be used by RAYSHADE. Two files are created, one with '.geom' extension and one with '.ray'. Since the number of polygons can be extremely large, the geometry is isolated in the '.geom' and is included (via '#include') in the main '.ray' file. The later holds the surface properties for all the geometry as well as viewing and RAYSHADE specific commands. This allows changing shading or 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 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'.

Here is a simple example:

```
irit2ray -l -f 5 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 curves surface. That will cause rayshade to issue an "Inconsistant 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 25 b58.dat
```

which will only recreate b58.geom (becuase of the -g option).

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", list( view_mat, b58 ) );
```

However one can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by poly3d:

```
poly3d b58.dat
irit2ray -l -f 5 b58.dat generic0.mat
```

where generic0.mat is the viewing matrix created by poly3d.

14.3 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 than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. For 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 behaviour 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". Refer to RAYSHADE manual for their meaning. For example:

```
attrib( srf1, "transp", "0.3" );
attrib( srf1, "texture", "wood" );
```

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 being used if set. The later allowes you to specify only the 3 additive, 3 substractive and white and black and so is very limited for rendering purposes. Example:

```
attrib( tankBody, "rgb", "244,164,96" );
```

Current implementation allows white spaces in neither the attribute name nor in the attribute value.

15 Irit2Nff - IRIT To NFF filter

15.1 Command Line Options

```
irit2nff [-l] [-4] [-c] [-f FineNess] [-o OutName] [-g] [-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. Not using this option will better emulate the surface shape but will create unnecessary polygons in cases where one is enough.

- **-4**: Four - Generate four polygons per flat patch. Default is 2.

- **-c**: Output files should be filtered by cpp. By doing so, the usually huge geometry file is seperated from the main nff file that contains the surface properties and view parameters. By default all data, including the geometry, is 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 '.geom' extension and which will be included using '#include' command. The '.nff' file should, in that case, be preprocessed using cpp before piped into the nff renderer.

- **-f FineNess**: An integer value controling the fineness of surface to polygons process. Roughly speaking it will set to the number of polygons along one Bezier patch direction. A Bezier patch will have order of $FineNess^2$ polygons then. The Order of the surface also affect the amount of polygons; The higher the order is, more polygons are created. A B-spline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.
- **-o OutName**: Name of output files. 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.
- **-z**: Print version number and current defaults.

15.2 Usage

Irit2Nff converts freeform surfaces into polygons in format that can be used by 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 and which separate 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 later holds the surface properties for all the geometry as well as the viewing specification. This allows changing shading or 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'.

Here is a simple example:

```
irit2nff -c -l -f 5 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 25 b58.dat
```

which will only recreate b58.geom (becuase of the -g option).

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", list( view_mat, b58 ) );
```

However one can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by poly3d:

```
poly3d b58.dat
irit2nff -l -f 5 b58.dat generic0.mat
```

where generic0.mat is the viewing matrix created by poly3d.

15.3 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 than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface. For 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 behaviour is controlled by the '-4' flag, but can be overwritten for individual surfaces bu setting "twoperflat" or "fourperflat".

NFF specific properties are controlled via the following attributes: "kd", "ks", "shine", "trans", "index". Refer to NFF manual for detail. For 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 *IRIT* COLOR command is being used if set. The later allowes you to specify only the 3 additive, 3 substractive and white and black and so is very limited for rendering purposes. Example:

```
attrib( tankBody, "rgb", "244,164,96" );
```

Current implementation allows white spaces in neither the attribute name nor in the attribute value.

16 Dat2Irit - Data To IRIT file filter

16.1 Command Line Options

```
  dat2irit [-z] DFiles
```

- `-z`: Print version number and current defaults.

16.2 Usage

It may be sometimes desired to convert .dat data files into a form that can be fed in back to *IRIT* - a '.irt' file. This filter will do exactly that. Example:

```
  dat2irit b58.dat > b58-new.irt
```

17 Data File Format

This section describes the data file format used to exchange data between *IRIT* and its accompanied 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 close polyline last pt 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 region boundary. Last point is NOT equal to first
    ;and a line from last point to first should be drawn when the polygon
    ;boundary is drawn.
  | [POLYGON {ATTRS} #PTS
        [{ATTRS} x y z]
        [{ATTRS} x y z]
            .
            .
            .
        [{ATTRS} x y z]
    ]

    ;Defines a cloud of points. This entry is not supported by IRIT.
  | [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 raw after raw (#UPTS per raw), #VPTS raws.
  | [SURFACE BEZIER {ATTRS} #UPTS #VPTS 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.
  | [CURVE BSPLINE {ATTRS} #PTS ORDER POINT_TYPE
        [KV {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 raw after raw (#UPTS per raw), #VPTS raws.
  | [SURFACE BSPLINE {ATTRS} #UPTS #VPTS UORDER VORDER POINT_TYPE
        [KV {ATTRS} kv0 kv1 kv2 ...]                ;U Knot vector
        [KV {ATTRS} kv0 kv1 kv2 ...]                ;V Knot vector
        [{ATTRS} {w} x y z ...]
        [{ATTRS} {w} x y z ...]
           .
           .
           .
        [{ATTRS} {w} x y z ...]
    ]
]

POINT_TYPE -> E2 | E3 | P2 | P3
```

```
ATTRS -> [ATTRNAME ATTRVALUE]
         | [ATTRNAME ATTRVALUE] ATTRS
```

Some notes:

1. This new definition for the text file is design to minimize the reading time and space. All information can be read without backward or forward referencing (as used to be in the old format).

2. An OBJECT can hold any number of geometry entities such as POLYGONs or CURVEs. It is not recommended at this time to have more than one curve or one surface in an object since this feature is not fully implemented for free form objects.

3. An OBJECT must not hold different geometry or other entities. I.e. CURVEs, SURFACEs, and POLYGONs must all be in different OBJECTs.

4. 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 simplified and disambiguous the parsing.

5. Comments may appear between '`[OBJECT ...]`' blocks, or immediatelly 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.

6. It is prefered that geometric attributes such as NORNALs will be saved in the geometry strurcture level (POLYGON, CURVE or vertices) while graphical and others such as COLORs will be saved in the OBJECT level.

7. Objects may be contained in other objects to any 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]
      ]
  ]
]
```

18 Bugs and Limitations

Like any program of more than one line it is far from been perfect. Some limitations as well as simplifications are layed out below.

1. No intersection of co-planar polygons is allowed. Such case results are undefined. Most of the time, one can move one of the operands in the Boolean operation by an EPSILON. Such EPSILON should be in the order of 10-3 if the system uses float and and 10-6 if doubles are used. (the UNIX and MSDOS DJGPP version uses doubles, MSDOS BC++ uses floats).

2. If the intersection curve of two objects falls exactly on polygon boundaries, for all polygons, the system will scream that the two object do not intersect at all. Again, try to move one by EPSILON into the other. I probably should fix this one - that suppose to be relatively easy.

3. Avoid degeneracies - intersection that results with a point or a line 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.

4. 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 NULL object. One should find 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.

5. Sweep of a circular curve along circular curve does *not* create an exact piece of a torus. This is probably due to the fact that both curves are rationals.

6. No degree raising for Bspline surfaces of order larger than two.