```
            [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]
        ]
    ]
]
```

27 Bugs and Limitations

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

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

2. Avoid degenerate intersections that result with a point or a line. They will probably cause wrong propagation of the inner and outer part of one object relative to the other. Always extend your object beyond the other object.

3. If two objects have no intersection in their boundary, *IRIT* assumes they are disjoint: a union simply combines them, and the other Boolean operators return a NULL object. One should find a FAST way (3D Jordan theorem) to find the relation between the two (A in B, B in A, A disjoint B) and according to that, make a decision.

4. Sweep of a circular curve along a circular curve does *not* create an exact piece of a torus.

5. Since the boolean sum implementation constructs ruled surfaces with uniform speed, it might return a somewhat incorrect answer, given non-uniform input curves.

6. The parser is out of hand and is difficult to maintain. There are several memory leaks there that one should fix.

7. The X11 driver has no menu support (any easy way to have menus using Xlib!?).

8. IBM R6000 fails to run the drivers in -s- mode.

9. Rayshade complains a lot about degenerate polygons on irit2ray output. To alleviate the problem, change the 'equal' macro in common.h in libcommon of rayshade from EPSILON (1e-5) to 1e-7 or even lower.

```
    [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]
```

```
    ]

    ;Defines a Bspline surface with #UPTS * #VPTS control points, of order
    ;UORDER by VORDER. If the surface is rational, the rational component
    ;is introduced first.
    ;Points are printed row after row (#UPTS per row), #VPTS rows.
  | [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 -> E1 | E2 | E3 | E4 | E5 | P1 | P2 | P3 | P4 | P5

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

Some notes:

* This definition for the text file is designed to minimize the reading time and space. All information can be read without backward or forward referencing.

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

* Attributes should be ignored if not needed. The attribute list may have any length and is always terminated by a token that is NOT '['. This simplifies and disambiguates the parsing.

* Comments may appear between '[OBJECT ...]' blocks, or immediately after OBJECT OBJNAME, and only there.

A comment body can be anything not containing the '[' or the ']' tokens (signals start/end of block). Some of the comments in the above definition are *illegal* and appear there only of the sake of clarity.

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

* Objects may be contained in other objects to an arbitrary level.

Here is an example that exercises most of the data format:

```
This is a legal comment in a data file.
[OBJECT DEMO
    [OBJECT REAL_NUM
        And this is also a legal comment.
        [NUMBER 4]
    ]

    [OBJECT A_VECTOR
```

```
        .
        .
        .
    [{ATTRS} x y z]
  ]

  ;Defines a "cloud" of points.
| [POINTLIST {ATTRS} #PTS
    [{ATTRS} x y z]
    [{ATTRS} x y z]
        .
        .
        .
    [{ATTRS} x y z]
  ]

  ;Defines a Bezier curve with #PTS control points. If the curve is
  ;rational, the rational component is introduced first.
| [CURVE BEZIER {ATTRS} #PTS POINT_TYPE
    [{ATTRS} {w} x y z ...]
    [{ATTRS} {w} x y z ...]
        .
        .
        .
    [{ATTRS} {w} x y z ...]
  ]

  ;Defines a Bezier surface with #UPTS * #VPTS control points. If the
  ;surface is rational, the rational component is introduced first.
  ;Points are printed row after row (#UPTS per row), #VPTS rows.
| [SURFACE BEZIER {ATTRS} #UPTS #VPTS POINT_TYPE
    [{ATTRS} {w} x y z ...]
    [{ATTRS} {w} x y z ...]
        .
        .
        .
    [{ATTRS} {w} x y z ...]
  ]

  ;Defines a 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 ...]
```

25.2 Usage

Irit2Xfg converts freeform surfaces and polygons into polylines in a format that can be used by XFIG.
Example:

```
irit2Xfg -l -f 15 saddle.dat > saddle.xfg
```

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

```
x11drvs b58.dat
irit2Xfg -l -f 5 b58.dat irit.mat > saddle.xfg
```

where irit.mat is the viewing matrix created by x11drvs.

26 Data File Format

This section describes the data file format used to exchange data between *IRIT* and its accompanying tools.

```
[OBJECT {ATTRS} OBJNAME
    [NUMBER n]

  | [VECTOR x y z]

  | [CTLPT POINT_TYPE {w} x y {z}]

  | [STRING "a string"]

  | [MATRIX m00 ... m03
            m10 ... m13
            m20 ... m23
            m30 ... m33]

    ;A polyline should be drawn from first point to last. Nothing is drawn
    ;from last to first (in a closed polyline, last point is equal to first).
  | [POLYLINE {ATTRS} #PTS                    ;#PTS = number of points.
        [{ATTRS} x y z]
        [{ATTRS} x y z]
           .
           .
           .
        [{ATTRS} x y z]
    ]

    ;Defines a closed planar region. Last point is NOT equal to first,
    ;and a line from last point to first should be drawn when the boundary
    ;of the polygon is drawn.
  | [POLYGON {ATTRS} #PTS
        [{ATTRS} x y z]
        [{ATTRS} x y z]
```

24.3 Advanced Usage

One can specify surface qualities for individual surfaces of a model. Several such attributes are supported by Irit2Scn and can be set within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain surface should be finer than the rest of the scene, one can set a "resolution" attribute which specifies the *relative* FineNess resolution of this specific surface.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The patch can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the patch center. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces bu setting "twoperflat" or "fourperflat".

RTrace specific properties are controlled via the following attributes: "SCNrefraction", "SCNtexture", "SCNsurface. Refer to the RTrace manual for their meaning.

Example:

```
attrib( srf1, "SCNrefraction", 0.3 );
```

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise a color as set via *IRIT* COLOR command is used if set.

Example:

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

25 Irit2Xfg - IRIT To XFIG filter

25.1 Command Line Options

```
irit2xfg [-s Size] [-t XTrans YTrans] [-I #UIso[:#VIso]] [-S #SampPerCrv]
                              [-M] [-P] [-T] [-i] [-o OutName] [-z] DFiles
```

- **-s Size**: Size in inches of the page. Default is 7 inches.
- **-t XTrans YTrans**: X and Y translation. of the image. Default is (0, 0).
- **-I #UIso[:#VIso]**: Specifies the number of isolines per surface, per direction. If #VIso is not specified, #UIso is used for #VIso as well.
- **-S #SampPerCrv**: Specifies the log based 2 of number of samples per (iso)curve.
- **-M**: Dumps the control mesh/polygon as well.
- **-P**: Dumps the curve/surface as isocurves.
- **-T**: Talkative mode. Prints processing information.
- **-i**: Internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-o OutName**: Name of output file. By default the name of the first data file from *DFiles* list is used. See below on the output files.
- **-z**: Prints version number and current defaults.

- **-f FineNess**: An integer value controlling the fineness of surface to polygons process. Roughly speaking, it will be 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 affects the amount of polygons; The higher the order is, the more polygons are created. A Bspline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.
- **-o OutName**: Name of output file. By default the name of the first data file from *DFiles* list is used. See below on the output files.
- **-g**: Generates the geometry file only. See below.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

24.2 Usage

Irit2Scn converts freeform surfaces and polygons into polygons in a format that can be used by RTrace. Two files are created, one with a '.geom' extension and one with '.scn'. Since the number of polygons can be extremely large, the geometry is isolated in the '.geom' file and is included (via '#include') in the main '.scn' file. The latter holds the surface properties for all the geometry as well as viewing and RTrace specific commands. This allows for the changing of the shading or the viewing properties while editing small ('.scn') files.

If '-g' is specified, only the '.geom' file is created, preserving the current '.scn' file.

In practice, it may be useful to create a low resolution approximation of the model, change the viewing/shading parameters in the '.scn' file until a good view and/or surface quality is found, and then run Irit2Scn once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2scn -l -f 5 b58.dat
```

creates b58.scn and b58.geom with low resolution (FineNess of 5).

One can ray trace this scene after converting the scn file to a sff file, using scn2sff provided with the RTrace package.

Once done with the parameter setting of RTrace, a fine approximation of the model can be created with:

```
irit2scn -l -g -f 25 b58.dat
```

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

One can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by the display devices:

```
wntdrvs b58.dat
irit2scn -l -f 5 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by wntdrvs. The output name, by default, is the last input file name, so you might want to provide an explicit name with the -o flag.

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

```
os2drvs b58.dat
irit2ray -l -f 5 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by os2drvs. The output name, by default, is the last input file name, so you might want to provide an explicit name with the -o flag.

23.3 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.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The rectangle can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the patch center. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces bu setting "twoperflat" or "fourperflat".

RAYSHADE specific properties are controlled via the following attributes: "specpow", "reflect", "transp", "body", "index", and "texture". Refer to RAYSHADE manual for their meaning.

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 the *IRIT* COLOR command is being used if set.

Example:

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

24 Irit2Scn - IRIT To SCENE (RTrace) filter

SCENE is the format used by the RTrace ray tracer. This filter was donated by Antonio Costa (acc@asterix.inescn.pt), the author of RTrace.

24.1 Command Line Options

```
irit2scn [-l] [-4] [-f FineNess] [-o OutName] [-g] [-T] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated as a single polygon along their linear direction. Although most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch.

- **-I #UIso[:#VIso]**: Specifies the number of isolines per surface, per direction. If #VIso is not specified, #UIso is used for #VIso as well.
- **-S #SampPerCrv**: Specifies the log based 2 of the number of samples per (iso)curve.
- **-z**: Prints version number and current defaults.

23.2 Usage

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

If '-g' is specified, only the '.geom' file is created, preserving the current '.ray' file.

In practice, it may be useful to create a low resolution approximation of the model, change the viewing/shading parameters in the '.ray' file until a good view and/or surface quality is found, and then run Irit2Ray once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2ray -l -f 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 curved surface. That will cause RAYSHADE to issue an "Inconsistent triangle normals" warning. This problem will not exist if high fineness is used. One can ray trace this scene using a command similar to:

```
RAYSHADE -p -W 256 256 b58.ray > b58.rle
```

Once done with parameter setting for RAYSHADE, a fine approximation of the model can be created with:

```
irit2ray -l -g -f 25 b58.dat
```

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

Interesting effects can be created using the depth cue support and polyline conversion of irit2ray. For example

```
irit2ray -G 5 -P -p -0.0 0.5 solid1.dat
```

will dump solid1 as a set of polylines (represented as truncated cones in RAYSHADE) with varying thickness according to the z depth. Another example is

```
irit2ray -G 5 -P -p -0.1 1.0 saddle.dat
```

which dumps the isolines extracted from the saddle surface with varying thickness.

Each time a data file is saved in *IRIT*, it can be saved with the viewing matrix of the last INTERACT by saving the VIEW_MAT object as well. I.e.:

```
save( "b58", b58 );
```

```
attrib( poly, "dash", "[0.006 0.0015 0.001 0.0015]" );
```

Surface color is controlled (for color postscript only - see -c) in two levels. If the object has an RGB attribute, it is used. Otherwise, a color as set via the *IRIT* COLOR command is used.

Example:

```
attrib( Ball, "rgb", "255,0,0" );
```

An object can be drawn as "tubes" instead of full lines. The ratio between the inner and the outer radii of the tube is provided as the TUBULAR attribute:

```
attrib( final, "tubular", 0.7 );
```

23 Irit2Ray - IRIT To RAYSHADE filter

23.1 Command Line Options

```
irit2ray [-l] [-4] [-G GridSize] [-f FineNess] [-o OutName] [-g]
  [-p Zmin Zmax] [-P] [-M] [-T] [-I #UIso[:#VIso]] [-S #SampPerCrv]
                                                        [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.
- **-G GridSize**: Usually objects are grouped as *lists* of polygons. This flags will coerce the usage of the RAYSHADE *grid* structure, with *GridSize* being used as the grid size along the object bounding box's largest dimension.
- **-f FineNess**: An integer value controlling the fineness of surface to polygons process. Roughly speaking, it will be 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 affects the amount of polygons; The higher the order is, the more polygons are created. A Bspline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.
- **-o OutName**: Name of output file. By default the name of the first data file from the *DFiles* list is used. See below on the output files.
- **-g**: Generates the geometry file only. See below.
- **-p Zmin Zmax**: Sets the ratios between the depth cue and the width of the dumped *polylines*. See also -P. Closer lines will be drawn wider.
- **-P**: Forces dumping polygons as polylines with thickness controlled by -p.
- **-M**: If -P (see -P and -p) then converts the control mesh/polygon to polylines which are represented as a sequence of truncated cones.
- **-T**: Talkative mode. Prints processing information.

- **-i**: Internal edges (created by *IRIT*) - the default is not to display them, and this option will force displaying them as well.
- **-o OutName**: Name of output file. Default is stdout.
- **-d [Zmin Zmax]**: Sets the ratios between the depth cue and the width of the dumped data. See also -W, -p. Closer lines/points will be drawn wider/larger. Zmin and Zmax are optional. The object's bounding box is otherwise computed and used.
- **-D [Zmin Zmax]**: Same as -d, but depth cue the color or gray scale instead of width. You might need to consider the sorting option of the illustrt tool (-s of illustrt) for proper drawings. Only one of -d and -D can be used.
- **-p PtType PtSize**: Specifies the way points are drawn. PtType can be one of H, F, C for Hollow circle, Full Circle, or Cross. PtSize specifies the size of the point to be drawn, in inches. Vectors will also be drawn as points, but with an additional thin line to the origin. See also -d.
- **-u**: Forces a unit matrix transformation, i.e. no transformation.
- **-z**: Prints version number and current defaults.

22.2 Usage

Irit2Ps converts freeform surfaces and polygons into a postscript file.

Example:

```
irit2ps solid1.dat > solid1.ps
```

Surfaces are converted to polygons with fineness control:

```
irit2ps -S 5 -c -W 0.01 saddle.dat > saddle.ps
```

creates a postscript file for the saddle model, in color, and with lines 0.01 inch thick.

22.3 Advanced Usage

One can specify several attributes that affect the way the postscript file is generated. The attributes can be generated within *IRIT*. See also the ATTRIB *IRIT* command.

If a certain object should be thinner or thicker than the rest of the scene, one can set a "width" attribute which specifies the line width in inches of this specific object.

Example:

```
attrib( srf1, "width", 0.02 );
```

will force srf1 to have this width, instead of the default as set via the '-W' flag.

If a (closed) object, a polygon for example, needs to be filled, a "fill" attribute should be set, with a value equal to the gray level desired.

Example:

```
attrib( poly, "fill", 0.5 );
```

will fill poly with %50 gray.

Dotted or dashed line effects can be created using a "dash" attribute which is a direct PostScript dash string. A simple form of this string is "[a b]" in which a is the drawing portion (black) in inches, followed by b inches of white space. See the postScript manual for more about the format of this string. Here is an example for a dotted-dash line.

- **-f FineNess**: An integer value controlling the fineness of surface to polygons process. Roughly speaking, it sets 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 affects the amount of polygons; The higher the order is, more polygons are created. A Bspline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

21.2 Usage

Irit2Plg converts freeform surfaces and polygons into polygons in a format that can be used by the REND386 renderer.

Example:

```
irit2plg solid1.dat > solid1.plg
```

Surfaces are converted to polygons with fineness control:

```
irit2plg -f 10 - view.mat < saddle.dat > saddle.plg
```

Note the use of '-' for stdin.

22 Irit2Ps - IRIT To PS filter

22.1 Command Line Options

```
irit2ps [-s Size] [-I #UIso[:#VIso]] [-S #SampPerCrv] [-M] [-P]
        [-W LineWidth] [-c] [-C] [-T] [-i] [-o OutName] [-d [Zmin Zmax]]
                    [-D [Zmin Zmax]] [-p PtType PtSize][-u] [-z] DFiles
```

- **-s Size**: Controls the size of the postscript output in inches. Default is to fill the entire screen.
- **-I #UIso[:#VIso]**: Specifies the number of isolines per surface, per direction. If #VIso is not specified, #UIso is used for #VIso as well.
- **-S #SampPerCrv**: Specifies the log based 2 of number of samples per (iso)curve.
- **-M**: Dumps the control mesh/polygon as well.
- **-P**: Dumps the curve/surface as isocurves.
- **-W #LineWidth**: Sets the line drawing width in inches. Default is as thin as possible. This option will overwrite only those objects that do *not* have a "width" attribute. See also -d.
- **-c**: Creates a *color* postscript file.
- **-C**: Curve mode. Dumps freeform curves and surfaces as cubic Bezier curves. Higher order curves and surfaces and/or rationals are approximated by cubic Bezier curves. This option generates data files that are roughly a third of piecewise linear postscript files (by disabling this feature, -C-), but takes a longer time to compute.
- **-T**: Talkative mode. Prints processing information.

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

One can overwrite the viewing matrix by appending a new matrix in the end of the command line, created by a display device:

```
xgldrvs b58.dat
irit2nff -l -f 5 b58.dat irit.mat
```

where irit.mat is the viewing matrix created by xgldrvs.

20.3 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.

Example:

```
attrib( srf1, "resolution", 2 );
```

will force srf1 to have twice the default resolution, as set via the '-f' flag.

Almost flat patches are converted to polygons. The rectangle can be converted into two polygons (by subdividing along one of its diagonals) or into four by introducing a new point at the center of the patch. This behavior is controlled by the '-4' flag, but can be overwritten for individual surfaces by setting a "twoperflat" or a "fourperflat" attribute.

NFF specific properties are controlled via the following attributes: "kd", "ks", "shine", "trans", "index". Refer to the NFF manual for detail.

Example:

```
attrib( srf1, "kd", 0.3 );
attrib( srf1, "shine", 30 );
```

Surface color is controlled in two levels. If the object has an RGB attribute, it is used. Otherwise, a color, as set via the *IRIT* COLOR command, is used if set.

Example:

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

21 Irit2Plg - IRIT To PLG (REND386) filter

PLG is the format used by the rend386 real time renderer for the IBM PC.

21.1 Command Line Options

```
irit2plg [-l] [-4] [-f FineNess] [-T] [-z] DFiles
```

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although, most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free form shape (saddle like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.
- **-4**: Four - Generates four polygons per flat patch. Default is 2.

- **-l**: Linear - forces linear (degree two) surfaces to be approximated by a single polygon along their linear direction. Although, most of the time, linear direction can be exactly represented using a single polygon, even a bilinear surface can have a free-form shape (saddle-like) that is not representable using a single polygon. Note that although this option will better emulate the surface shape, it will create unnecessary polygons in cases where one is enough.

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

- **-c**: Output files should be filtered by cpp. When set, the usually huge geometry file is separated from the main nff file that contains the surface properties and view parameters. By default all data, including the geometry, are saved into a single file with type extension '.nff'. Use of '-c' will pull out all the geometry into a file with the same name but a '.geom' extension, which will be included using the '#include' command. The '.nff' file should, in that case, be preprocessed using cpp before being piped into the nff renderer.

- **-f FineNess**: An integer value controlling the fineness of surface to polygons process. Roughly speaking, it sets 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 affects the amount of polygons; the higher the order is, the more polygons are created. A Bspline surface is first converted into piecewise Bezier to make sure C1 discontinuities will show up in the polygonal approximation.

- **-o OutName**: Name of output file. By default the name of the first data file from the *DFiles* list is used. See below on the output files.

- **-g**: Generates the geometry file only. See below.

- **-T**: Talkative mode. Prints processing information.

- **-z**: Prints version number and current defaults.

20.2 Usage

Irit2Nff converts freeform surfaces into polygons in a format that can be used by an NFF renderer. Usually, one file is created with '.nff' type extension. Since the number of polygons can be extremely large, a '-c' option is provided, which separates the geometry from the surface properties and view specification, but requires preprocessing by cpp. The geometry is isolated in a file with extension '.geom' and included (via '#include') in the main '.nff' file. The latter holds the surface properties for all the geometry as well as the viewing specification. This allows for the changing of shading or the viewing properties while editing small ('.nff') files.

If '-g' is specified, only the '.geom' file is created, preserving the current '.nff' file. The '-g' flag can be specified only with '-c'.

In practice, it may be useful to create a low resolution approximation of the model, change viewing/shading parameters in the '.nff' file until a good view and/or surface quality is found, and then run Irit2Nff once more to create a high resolution approximation of the geometry using '-g'.

Example:

```
irit2nff -c -l -f 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
```

- Generates vertices of polygons in the input data set as points in output data controlled via the '-p' flag. set.
- Applies a Z sort to the output data, if '-s', so drawing in order of the data will produce a properly hidden surface removal drawing.

Here is a more complex example. Make sure tubular is properly set via "attrib(solid1, "tubular", 0.7);" and invoke:

```
illustrt -s -p -l 0.1 -t 0.05 solid1.dat |
    irit2ps -W 0.05 -d 0.2 0.6 -p h 0.05 -u - > solid.ps
```

makes sure all segments piped into irit2ps are shorter than 0.1, generates points for the vertices, sorts the data in order to make sure hidden surface removal is correctly applied, and trims the far edge by 0.05 at an intersection point. Irit2ps is invoked with depth cueing activated and a default width of 0.05, points are drawn as hollowed circles of default size 0.05, and lines are drawn tubular.

17 Dat2Irit - Data To IRIT file filter

17.1 Command Line Options

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

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

17.2 Usage

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

Example:

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

18 Dxf2Irit - DXF (Autocad) To IRIT filter

Due to lack of real documentation on the DXF format (for surfaces), this filter is not really complete. It only work for polygons, and is provided here only for those desperate enough to try and fix it...

19 Irit2Dxf - IRIT To DXF (Autocad) filter

Due to lack of real documentation on the DXF format (for surfaces), this filter is not really complete. It works only for polygons, and is provided here only for those desperate enough to try and fix it...

20 Irit2Nff - IRIT To NFF filter

20.1 Command Line Options

```
irit2nff [-l] [-4] [-c] [-f FineNess] [-o OutName] [-T] [-g] [-z] DFiles
```

16.2 Command Line Options

```
illustrt [-I #IsoLines] [-S #SampPerCrv] [-s] [-M] [-P] [-p]
         [-l MaxLnLen] [-a] [-t TrimInter] [-o OutName] [-T] [-z] DFiles
```

- **-I #IsoLines**: Specifies number of isolines per surface, per direction.
- **-S #SampPerCrv**: Specifies the log based 2 of number of samples per (iso)curve.
- **-s**: sorts the data in Z depth order that emulates hidden line removal once the data are drawn.
- **-M**: Dumps the control mesh/polygon as well.
- **-P**: Dumps the curve/surface as isocurves.
- **-p**: Dumps vertices of polygons/lines as points.
- **-l MaxLnLen**: breaks long lines into shorter ones with maximal length of MaxLnLen. This option is necessary to achieve good depth depending line width in the '-d' option of irit2ps.
- **-a**: takes into account the angle between the two (poly)lines that intersect when computing how much to trim. See also -t.
- **-t TrimInter**: Each time two (poly)line segments intersect in the projection plane, the (poly)line that is farther away from the viewer is clipped TrimInter amount from both sides. See also -a.
- **-o OutName**: Name of output file. Default is stdout.
- **-T**: Talkative mode. Prints processing information.
- **-z**: Prints version number and current defaults.

16.3 Usage

illustrt is a simple line illustration tool. It process geometry such as polylines and surfaces and dumps geometry with attributes that will make nice line illustrations. illustrt is geared mainly toward its use with irit2ps to create postscript illustrations. Here is a simple example:

```
illustrt -s -l 0.1 solid1.dat | irit2ps -W 0.05 -d 0.2 0.6 -u - > solid.ps
```

make sure all segments piped into irit2ps are shorter than 0.1 and sort them in order to make sure hidden surface removal is correctly applied. Irit2ps is invoked with depth cueing activated, and a default width of 0.05.

illustrt dumps out regular *IRIT* data files, so output can be handled like any other data set. illustrt does the following processing to the input data set:

- Converts surfaces to isocurves ('-I' flag) and isocurves and curves to polylines ('-S' flag), and converts polygons to polylines. Polygonal objects are considered closed and even though each edge is shared by two polygons, only a single one is generated.
- Finds the intersection location in the projection plane of all segments in the input data set and trims away the far segment at both sides of the intersection point by an amount controlled by the '-t' and '-a' flags.
- Breaks polylines and long lines into short segments, as specified via the '-l' flag, so that width depth cueing can be applied more accurately (see irit2ps's '-d' flag) as well as the Z sorting.

- **-S SubSample**: subsamples, and uses a box filter to low-pass filter the image, using a grid size of SubSample by SubSample. This obviously makes things slower, but guess what - it looks much better.
- **-M Mask**: Creates a second GIF file named Mask that is a binary image set to 1 at any pixel covered by one of the objects, and 0 otherwise. As a color of an object can become equal to the background at some point, there is no way to find whether a pixel is representing the background or an object in the background color. The Mask can be used instead. This Mask can be used when combining images (such as the gifcomp utility in 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 axis (see cagd_lib for more).
- **-z**: Prints version number and current defaults.

The image is dumped to stdout as a GIF image which can be redirected to a file or 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 -?-.

15.3 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. 'poly3d-r -z' will display the current configuration as read from the configuration file.

The configuration file is searched in the directory specified by the IRIT_PATH environment variable. For example 'setenv IRIT_PATH /u/gershon/irit/bin/'. If the IRIT_PATH variable is not set, the current directory is searched.

15.4 Usage

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

Some Remarks:

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

2. The color of the object can be extracted 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.

Example:

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

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

16 Illustrt - Simple line illustration filter

16.1 Introduction

illustrt is a filter that processes IRIT data files and dumps out modified IRIT data files. illustrt can be used to make simple nice illustrations of data. The features of illustrt include depth sorting, hidden line clipping at intersection points, and vertex enhancements. illustrt is designed to closely interact with irit2ps, although it is not neceessary to use irit2ps on illustrt output.

15 poly3d-r - A Simple Data Rendering Program

15.1 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 consider one of several raytracing public domain programs available (such as RAYSHADE, for which irit2ray can generate data) for high quality images.

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

1. Processes the input (parsing.)

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

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

4. Scans the data in a scan line fashion and dumps out the image.

This program can handle CONVEX polygons only. From *IRIT* one can ensure that a model consists of convex polygons only using the CONVEX command:

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

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

15.2 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 between 0.0 and 1.0).
- **-c N**: number of bits per pixel N (must be between 1 and 8 bits).
- **-l X Y Z**: specifies the light source normal direction. This vector does not have to be a unit vector. Only one light source is supported.
- **-2** : Forces emulation of 2 light sources at opposite directions as selected via [-l]. This may be useful for models that have no plane specified (i.e., the models has no PLANE attribute for their polygons), as the program guesses the equation from the points themselves, and which can be in the opposite of the desired direction.
- **-m**: More - provides some more information on the data file(s) parsed.
- **-s Xsize Ysize**: specifies 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 guarantees none of the original image will be clipped.
- **-b**: Purges 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 by about %15. Using this option for incomplete boundary objects and/or objects with polygons with no PLANE specified will almost definitely create an incorrect image.
- **-g**: Use Gouraud shading interpolation (flat shading is used by default). This is somewhat slower, but creates nicer results.

14.2 Command Line Options

```
poly3d-h [-b] [-m] [-i] [-e #Edges] [-f FineNess] [-H] [-4] [-q] [-z] [-c]
                                        [-o OutName] DFiles > OutFile
```

- **-b**: BackFacing - if an object is closed (such as most models created by *IRIT*), back facing polygons can be deleted, therefore speeding up the process by at least a factor of two.
- **-m**: More - provides some more information on the data file(s) parsed.
- **-i**: Internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-e n**: Number of edges to use from each given polygon (default all). Handy as '-e 1 -4' for freeform data.
- **-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 axis.
- **-H**: Dumps both visible lines and hidden lines as separated objects. Hidden lines will be dumped using a different (dimmer) color and (a narrower) line width.
- **-4**: Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only.
- **-q**: Quiet mode. No printing aside from fatal errors. Disables -m.
- **-o OutName**: Name of output file. Default is stdout.
- **-z**: Prints version number and current defaults.
- **-c**: Clips data to screen (default). If disabled ('-c-'), data outside the view screen ([-1, 1] in x and y) are also processed.

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

14.3 Configuration

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

The configuration file is searched in the directory specified by the IRIT_PATH environment variable. For example, 'setenv IRIT_PATH /u/gershon/irit/bin/'. If the IRIT_PATH variable is not set, the current directory is searched.

14.4 Usage

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

```
poly3d-h solid1.dat | irit2ps - > solid1.ps
```

All the utilities have command line options. If an option is set by a '-x' then '-x-' resets the option. The command line options overwrite the settings in config files, and the reset option is useful for cases where the option is set by default, in the configuration file.

All utilities can read a sequence of data files. However, the *last* transformation matrices found (VIEW_MAT and PRSP_MAT) are actually used.

Example:

```
poly3d-h solid1.dat | x11drvs solid1.dat - solid1.mat
```

x11drvs will display the original solid1.dat file with its hidden version, as computed by poly3d-h, all with the solid1.mat, ignoring all other matrices in the data stream.

Under unix, compressed files with a postfix ".Z" will be *automatically* uncompressed on read and write. The following is legal under unix,

```
poly3d-h solid1.dat.Z | x11drvs solid1.dat.Z - solid1.mat
```

where solid1.dat.Z was saved from within IRIT using the command

```
save( "solid1.dat.Z", solid1 );
```

or similar. The unix system's "compress" and "zcat" are used for the purpose of (un)compressing the data via pipes. See also SAVE and LOAD.

13 poly3d - A Data Display Program

This program has been retired. The display devices should be used instead. See the display drivers section.

14 poly3d-h - Hidden Line Removing Program

14.1 Introduction

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

The program performs 4 passes over the input:

1. Preprocesses and maps all polygons in a scene, and sorts them.

2. Generates edges out of the polygonal model and sorts them (preprocessing for the scan line algorithm) into buckets.

3. Intersects edges, and splits edges with non-homogeneous visibility (the scan line algorithm).

4. Applies a visibility test of each edge.

This program can handle CONVEX polygons only. From *IRIT* one can ensure that a model consists of convex polygons only using the CONVEX command:

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

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

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

The following commands are valid for the STATE COMMAND above,

MoreSense:	More sensitive mouse control.
LessSense:	Less sensitive mouse control.
ScrnObject:	Toggle screen/object transformation mode.
PerspOrtho:	Toggles perspective/orthographic trans. mode.
DepthCue:	Toggles depth cueing drawing.
DrawSolid:	Toggles isocurve/shaded solid drawing.
DblBuffer:	Toggles single/double buffer mode.
AntiAlias:	Toggles anti aliased lines.
DrawIntrnl:	Toggles drawing of internal lines.
DrawVNrml:	Toggles drawing of normals of vertices.
DrawPNrml:	Toggles drawing of normals of polygons.
DSrfMesh:	Toggles drawing of control meshes/polygons.
DSrfPoly:	Toggles drawing of curves/surfaces.
4PerFlat:	Toggles 2/4 polygons per flat surface regions.
MoreIso:	Doubles the number of isolines in a surface.
LessIso:	Halves the number of isolines in a surface.
FinrAprx:	Doubles the number of samples per curve.
CrsrAprx:	Halves the number of samples per curve.
LngrVecs:	Doubles the length of displayed normal vectors.
ShrtrVecs:	Halves the length of displayed normal vectors.
WiderLns:	Doubles the width of the drawn lines.
NarrwLns:	Halves the width of the drawn lines.
FinrAdapIso:	Doubles the number of adaptive isocurves.
CrsrAdapIso:	Halves the number of adaptive isocurves.
FinerRld:	Doubles the number of ruled surfaces in adaptive isocurves.
CrsrRld:	Halves the number of ruled surfaces in adaptive isocurves.
RuledSrfApx:	Toggles ruled surface approximation in adaptive isocurves.
AdapIsoDir:	Toggles the row/col direction of adaptive isocurves.
Front:	Selects a front view.
Side:	Selects a side view.
Top:	Selects a top view.
Isometry:	Selects an isometric view.

Obviously not all state options are valid for all drivers. The *IRIT* server defines in iritinit.irt several user-defined functions that exercise some of the above state commands, such as VIEWSTATE and VIEWSAVE.

In addition to state modification via communication with the *IRIT* server, modes can be interactively modified on most of the display devices using a pop-up menu that is activated using the *right button in the transformation window*. This pop up menu is somewhat different in different drivers, but its entries closely follow the entries of the above state command table.

12 Utilities - General Usage

The *IRIT* solid modeler is accompanied by quite a few utilities. They can be subdivided into two major groups. The first includes auxiliary tools such as illustrt and poly3d-h. The second includes filters such as irit2ray and irit2ps.

All these tools operate on input files, and most of the time produce data files. In all utilities that read files, the dash ('-') can be used to read stdin.

Example:

- **UnitMatrix:** Forces a Unit matrix. That is, input data are *not* transformed at all. Same as '-u'.
- **DrawSolid:** Requests a shaded surface rendering, as opposed to isocurve surface representation.
- **DoubleBuffer:** Requests drawing using a double buffer, if any.
- **NumOfIsolines:** Specifies number of isolines per surface, per direction. Same as '-I'.
- **SamplesPerCurve:** Specifies the log based 2 of number of samples per (iso)curve. Same as '-S'.
- **LineWidth:** Sets the linewidth, in pixels. Default is one pixel wide. Same as '-l'
- **AdapIsoDir:** Selects the direction of the adaptive isoline rendering.
- **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 axis. Same as '-f'.
- **DepthCue:** Set depth cueing on. Drawings that are closer to the viewer will be drawn in more intense color. Same as '-c'.
- **FourPerFlat:** Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only. Same as '-4'.
- **AntiAlias:** Request the drawing of anti aliased lines.
- **DrawSurfaceMesh:** Draws control mesh/polygon of curves and surfaces, as well. Same as '-M'.
- **DrawSurfacePoly:** Draws curves and surfaces (surfaces are drawn using a set of isocurves, see -I, or polygons, see -f). Same as '-P'.
- **StandAlone:** Runs the driver in a Stand alone mode. Otherwise, the driver will attempt to communicate with the *IRIT* server. Same as '-s'.
- **TransMode:** Selects between object space transformations and screen space transformation.
- **ViewMode:** Selects between perspective and orthographic views.
- **NormalLength:** Sets the length of the drawn normals in thousandths of a unit. Same as '-L'.
- **DebugObjects:** Debug objects. Prints to stderr all objects read from the communcation port with the server *IRIT*. Same as '-d'.
- **DebugEchoInput:** Debug input. Prints to stderr all characters read from the communcation port with the server *IRIT*. Lowest level of communications.

11.3 Interactive mode setup

Commands that affect the status of the display device can also be sent via the communication port with the *IRIT* server. The following commands are recognized,

CLEAR	Clears the display area. All objects displayed are deleted.
DCLEAR	Delayed clear. Same as CLEAR but delayed until the next object is sent from the server. Useful for animation.
DISCONNECT	Closes connection with the server, but does not quit.
EXIT	Closes connection with the server and quits.
MSAVE NAME	Saves the transformation matrix file by the name NAME.
BEEP	Makes some sound.
STATE COMMAND	Changes the state of the display device. See below.

- **-N**: Draws normals of polygons.
- **-i**: Draws internal edges (created by *IRIT*) - default is not to display them, and this option will force displaying them as well.
- **-c**: Sets depth cueing on. Drawings that are closer to the viewer will be drawn in more intense color.
- **-m**: Provides some more information on the data file(s) parsed.
- **-I #IsoLines**: Specifies number of isolines per surface, per direction.
- **-S #SampPerCrv**: Specifies the log based 2 of number of samples per (iso)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 axis.
- **-l LineWidth**: Sets the linewidth, in pixels. Default is one pixel wide.
- **-d**: Debug objects. Prints to stderr all objects read from communcation port with the server *IRIT*.
- **-D**: Debug input. Prints to stderr all characters read from communcation port with the server *IRIT*. Lowest level of communication.
- **-L NormalLen**: Sets the length of the drawn normals in thousandths of a unit.
- **-4**: Forces four polygons per almost flat region in the surface to polygon conversion. Otherwise two polygons only.
- **-b BackGround**: Sets the background color as three RGB integers in the range of 0 to 255.
- **-M**: Draw control mesh/polygon of curves and surfaces, as well.
- **-P**: Draws curves and surfaces (surfaces are drawn using a set of isocurves, see -I, or polygons, see -f).
- **-z**: Prints version number and current defaults.

11.2 Configuration Options

The configuration file is read *before* the command line options are processed. Therefore, all options in this section can be overriden by the appropriate command line option, if any.

- **TransPrefPos:** Preferred location (Xmin, YMin, Xmax, Ymax) of the transformation window.
- **ViewPrefPos:** Preferred location (Xmin, YMin, Xmax, Ymax) of the viewing window.
- **BackGround:** Background color. Same as '-b'.
- **Internal:** Draws internal edges. Same as '-i'.
- **DrawVNormal:** Draws normals of vertices. Same as '-n'.
- **DrawPNormal:** Draws normals of polygons. Same as '-n'.
- **MoreVerbose:** Provides some more information on the data file(s) parsed. Same as '-m'.

11 Display devices

The following display device drivers are available,

Device Name	Invocation	Environment
xgldrvs	xgldrvs -s-	SGI 4D GL regular driver.
xgladap	xgladap -s-	SGI 4D GL adaptive isocurve experminal driver.
x11drvs	xgldrvs -s-	X11 driver.
wntdrvs	wntdrvs -s-	IBM PC Windows NT driver.
os2drvs	os2drvs -s-	IBM PC OS2 2.x driver.
amidrvs	amidrvs -s-	AmigaDOS 2.04+ driver.
nuldrvs	nuldrvs -s- [-d] [-D]	A device to print the object stream to stdout.

All display devices are clients communicating with the server (*IRIT*) using IPC (inter process communication). On Unix and Window NT sockets are used. A Windows NT client can talk to a server (*IRIT*) on a unix host if hooked to the same netwrok. On OS2 pipes are used, and both the client and server must run on the same machine. On AmigaDOS exec messages are used, and both the client and server must run on the same machine.

The server (*IRIT*) will automatically start a client display device if the IRIT_DISPLAY environment variable is set to the name and options of the display device to run. For example:

```
setenv IRIT_DISPLAY xgldrvs -s-
```

The display device must be in a directory that is in the **path** environment variable. Most display devices require the '-s-' flags to run in a non-standalone mode, or a client-server mode. Most drivers can also be used to display data in a standalone mode (i.e., no server). For example:

```
xgldrvs -s solid1.dat irit.mat
```

Effectively, all the display devices are also data display programs (poly3d, which was the display program prior to version 4.0, is retired in 4.0). Therefore some functionality is not always as expected. For example, the quit button will always force the display device to quit, even if poped up from *IRIT*, but will not cause *IRIT* to quit as might logically expected. In fact, the next time *IRIT* will try to communicate with the display device, it will find the broken connection and will start up a new display device.

All display devices recognize all the command line flags and all the configuration options in a configuration file, as described below. The display devices will make attemps to honor the requests, to the best of their ability. For example, only xgldrvs can render shaded models, and so only it will honor a **DrawSolid** configuration options.

11.1 Command Line Options

```
???drvs [-s] [-u] [-n] [-N] [-i] [-c] [-m] [-I #IsoLines] [-S #SampPerCrv]
        [-f FineNess] [-l LineWidth] [-d] [-D] [-L NormalLen] [-4]
        [-b BackGround] [-M] [-P] [-z] DFiles
```

- **-s**: Runs the driver in a Standalone mode. Otherwise, the driver will attempt to communicate with the *IRIT* server.
- **-u**: Forces a Unit matrix. That is, input data are *not* transformed at all.
- **-n**: Draws normals of vertices.

10.7.34 POLY_TYPE

A constant defining an object of type poly.

10.7.35 RED

A constant defining a RED color.

10.7.36 ROW

A constant defining the ROW direction of a surface mesh.

10.7.37 SGI

A constant designating an SGI system, in the MACHINE variable.

10.7.38 STRING_TYPE

A constant defining an object of type string.

10.7.39 SURFACE_TYPE

A constant defining an object of type surface.

10.7.40 SUN

A constant designating a SUN system, in the MACHINE variable.

10.7.41 TRUE

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

10.7.42 UNIX

A constant designating a generic UNIX system, in the MACHINE variable.

10.7.43 VECTOR_TYPE

A constant defining an object of type vector.

10.7.44 WHITE

A constant defining a WHITE color.

10.7.45 YELLOW

A constant defining a YELLOW color.

10.7.20 MAGENTA

A constant defining a MAGENTA color.

10.7.21 MATRIX_TYPE

A constant defining an object of type matrix.

10.7.22 MSDOS

A constant designating an MSDOS system, in the MACHINE variable.

10.7.23 NUMERIC_TYPE

A constant defining an object of type numeric.

10.7.24 OFF

Synonym of FALSE.

10.7.25 ON

Synonym for TRUE.

10.7.26 P1

A constant defining a P1 (X and W coordinates) rational control point type.

10.7.27 P2

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

10.7.28 P3

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

10.7.29 P4

A constant defining a P4 rational control point type.

10.7.30 P5

A constant defining a P5 rational control point type.

10.7.31 PI

The constant of 3.141592...

10.7.32 PLANE_TYPE

A constant defining an object of type plane.

10.7.33 POINT_TYPE

A constant defining an object of type point.

10.7.6 CURVE_TYPE

A constant defining an object of type curve.

10.7.7 CYAN

A constant defining a CYAN color.

10.7.8 E1

A constant defining an E1 (X only coordinate) control point type.

10.7.9 E2

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

10.7.10 E3

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

10.7.11 E4

A constant defining an E4 control point type.

10.7.12 E5

A constant defining an E5 control point type.

10.7.13 FALSE

A zero constant. May be used as Boolean operand.

10.7.14 GREEN

A constant defining a GREEN color.

10.7.15 HP

A constant designating an HP system, in the MACHINE variable.

10.7.16 IBMOS2

A constant designating an IBM system running under OS2, in the MACHINE variable.

10.7.17 IBMNT

A constant designating an IBM system running under Windows NT, in the MACHINE variable.

10.7.18 KV_FLOAT

A constant defining a floating end condition uniformly spaced knot vector.

10.7.19 KV_OPEN

A constant defining an open end condition uniformly spaced knot vector.

10.5.10 PRSP_MAT

Predefined matrix object (MatrixType) to hold the perspective matrix used/set by VIEW and/or INTERACT commands. See also VIEW_MAT.

10.5.11 RESOLUTION

Predefined numeric object (NumericType) that sets the accuracy of the polygonal primitive geometric objects and the approximation of curves and surfaces. Holds the number of divisions a circle is divided into (with minimum value of 4). If, for example, RESOLUTION is set to 6, then a generated CONE will effectively be a six-sided pyramid. Also controls the fineness of freeform curves and surfaces when they are approximated as piecewise linear polylines, and the fineness of freeform surfaces when they are approximated as polygons.

10.5.12 VIEW_MAT

Predefined matrix object (MatrixType) to hold the viewing matrix used/set by VIEW and/or INTERACT commands. See also PRSP_MAT.

10.6 System constants

The following constants are used by the various functions of the system to signal certain conditions. Internally, they are represented numerically, although, in general, their exact value is unimportant and may be changed in future versions. In the rare circumstance that you need to know their values, simply type the constant as an expression.

Example:

```
MAGENTA;
```

10.7 AMIGA

A constant designating an AMIGA system, in the MACHINE variable.

10.7.1 APOLLO

A constant designating an APOLLO system, in the MACHINE variable.

10.7.2 BLACK

A constant defining a BLACK color.

10.7.3 BLUE

A constant defining a BLUE color.

10.7.4 COL

A constant defining the COLumn direction of a surface mesh.

10.7.5 CTLPT_TYPE

A constant defining an object of type control point.

10.5.1 AXES

Predefined polyline object (PolylineType) that describes the XYZ axes.

10.5.2 COPLANAR

Predefined Boolean object (NumericType). If TRUE the polygonal Boolean operations will support coplanar faces but is somewhat slower. Default is FALSE.

10.5.3 DRAWCTLPT

Predefined Boolean variable (NumericType) that controls whether curves' control polygons and surfaces' control meshes are drawn (TRUE) or not (FALSE). Default is FALSE.

10.5.4 DUMPLVL

Content of objects assigned to variables can be displayed by executing the command 'ObjName;' where ObjName is the name of the object. This variable (NumericType) controls the way the data are 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.

Default value is 1.

10.5.5 ECHOSRC

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

10.5.6 FLAT4PLY

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

10.5.7 INTERCRV

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

10.5.8 MACHINE

Predefined numeric object (NumericType) holding the machine type as one of the following constants: MSDOS, SGI, HP, SUN, APOLLO, UNIX, IBMOS2, IBMNT, and AMIGA.

10.5.9 POLYSORT

Predefined numeric object (NumericType) that determines the directional axis along which the polygons are sorted during Boolean operations. Default is the x axis or 0. Set to 1 for y sorting. or to 2 for z sorting.

However, since VIEW is a user defined function, the following will not use VIEW_MAT as one would expect:

```
VIEW( view_mat, TRUE );
```

because VIEW_MAT will be renamed inside the VIEW user defined function to a local (to the user defined function) variable.

In iritinit.irt one can find several other useful VIEW related functions:

VIEWCLEAR	Clears all data displayed on the display device.
VIEWREMOVE	Removes the object specified by name from display.
VIEWDISC	Disconnects from display device (which is still running) while allowing IRIT to connect to a new device.
VIEWEXIT	Forces the display device to exit.
VIEWSAVE	Request sdisplay device to save transformation matrix.
BEEP	An emulation of the BEEP command of versions prior to 4.0.
VIEWSTATE	Allows to change the state of the display device.

For the above VIEW related functions, only VIEWREMOVE, VIEWSAVE, and VIEWSTATE require a parameter, which is the file name and view state respectively. The view state can be one of several commands. See the display device section for more.

Examples:

```
VIEWCLEAR();
VIEW( axes, off );
VIEWSTATE( "LngrVecs" );
VIEWSTATE( "DrawSolid" );
VIEWSAVE( "matrix1" );
VIEWREMOVE( "axes" );
VIEWDISC();
```

10.4.26 VIEWOBJ

```
VIEWOBJ( GeometricTreeType Object )
```

Displays the (geometric) object(s) as given in **Object**. **Object** may be any GeometricType or a list of other GeometricTypes nested to an arbitrary level.

Unlike *IRIT* versions prior to 4.0, VIEW_MAT is not explicitly used as the transformation matrix. In order to display with a VIEW_MAT view, VIEW_MAT should be listed as an argument (in that exact name) to VIEWOBJ. Same is true for the perspective matrix PRSP_MAT.

Example:

```
VIEWOBJ( list( view_mat, Axes ), FALSE );
```

displays the predefined object **Axes** in the viewing window using the viewing matrix VIEW_MAT.

10.5 System variables

System variables are predefined objects in the system. Any time *IRIT* is executed, these variable are automatically defined and set to values which are sometimes machine dependent. These are *regular* objects in any other sense, including the ability to delete or overwrite them. One can modify, delete, or introduce other objects using the IRITINIT.IRT file.

10.4.21 SYSTEM

```
SYSTEM( StringType Command )
```

Executes a system command **Command**. For example,

```
SYSTEM( "ls -l" );
```

10.4.22 TIME

```
TIME( NumericType Reset )
```

Returns the time in seconds from the last time TIME was called with **Reset** TRUE. This time is CPU time if such support is available from the system (times function), and is real time otherwise (time function). The time is automatically reset at the beginning of the execution of this program.

Example:

```
Dummy = TIME( TRUE );
  .
  .
  .
TIME( FALSE );
```

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

10.4.23 VARLIST

```
VARLIST()
```

List all the currently defined objects in the system.

10.4.24 VECTOR

```
VectorType VECTOR( NumericType X, NumericType Y, NumericType Z )
```

Creates a vector type object, using the three provided NumericType scalars.

10.4.25 VIEW

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

Displays the (geometric) object(s) as given in **Object**.

If **ClearWindow** is non-zero (see TRUE/FALSE and ON/OFF) the window is first cleared (before drawing the objects).

Example:

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

displays the predefined object **Axes** in the viewing window on top of what is drawn already.

In version 4.0, this function is emulated (see iritinit.irt) using the VIEWOBJ function. In order to use the current viewing matrix, VIEW_MAT should be provided as an additional parameter. For example,

```
VIEW( list( view_mat, Obj ), TRUE );
```

```
PRINTF("this is a string \"%s\" and this is an integer %8d.\\n",
      list("STRING", 1987));
PRINTF("this is a vector [%8.5lvf]\\n", list(vector(1,2,3)));
dumplvl = 9;
PRINTF("this is a object %8.6lDf...\\n", list(axes));
PRINTF("this is a object %10.8lDg...\\n", list(axes));
```

This implementation of PRINTF is somewhat different than the C programming language's version, because the backslash *always* escapes the next character during the processing stage of IRIT's parser. That is, the string

```
'\\tThis is the char \"\\%\"\\n'
```

is actually parsed by the IRIT's parser into

```
'\tThis is the char "\%"\n'
```

because this is the way the IRIT parser processes strings. The latter string is the one that PRINTF actually see.

10.4.18 PROCEDURE

```
ProcName = PROCEDURE(Prm1, Prm2, ... , PrmN):LclVal1:LclVar2: ... :LclVarM:
    ProcBody;
```

A procedure is a function that does not return a value, and therefore the return variable (see FUNCTION) should not be used. A procedure is identical to a function in every other way. See FUNCTION for more.

10.4.19 SAVE

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

Saves the provided **Object** in the specified file name **FileName**. No extension type is needed (ignored if specified), and ".dat" is always used. **Object** can be any object type, including list, in which structure is saved recursively. See also LOAD. If a display device is actively running at the time SAVE is invoked, its transformation matrix will be saved with the same name but with extension type of ".mat" instead of ".dat".

Under unix, files will be saved compressed if the given file name has a postfix of ".Z". The unix system's "compress" will be invoked via a pipe for that purpose.

10.4.20 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)'.

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

10.4.15 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)'.

10.4.16 PAUSE

```
PAUSE( NumericType Flush )
```

Waits for a keystroke. Nice to have if a temporary stop in a middle of an included file (see INCLUDE) is required. If **Flush** is TRUE, then the input is first flushed to guarantee that the actual stop will occur.

10.4.17 PRINTF

```
PRINTF( StringType CtrlStr, ListType Data )
```

A formatted printing routine, following the concepts of the C programming language's *printf* routine. **CtrlStr** is a string object for which the following special '%' commands are supported:

%d, %i, %o, %x, %X	Prints the numeric object as an octal or hexadecimal integer.
%e, %f, %g,	Prints the numeric object in several formats of
%E, %F	floating point numbers.
%s	Prints the string object as a string.
%pe, %pf, %pg	Prints the three coordinates of the point object.
%ve, %vf, %vg	Prints the three coordinates of the vector object.
%Pe, %Pf, %Pg,	Prints the four coordinates of the plane object.
%De, %Df, %Dg,	Prints the given object in IRIT's data file format.

All the '%' commands can include any modifier that is valid in the C programming language printf routine, including l (long), prefix character(s), size, etc. The point, vector, plane, and object commands can also be modified in a similar way, to set the format of the numeric data printed.

Also supported are the newline and tab using the backslash escape character:

```
PRINTF("\\tThis is the char \"\\%\"\\n", nil());
```

Backslashes should be escaped themselves as can be seen in the above example. Here are few more examples:

10.4.10 INCLUDE

```
INCLUDE( StringType FileName )
```

Executes the script file **FileName**. Nesting of include file is allowed up to 10 levels deep. If an error occurs, all open files in all nested files are closed and data are waited for at the top level (standard input).

A script file can contain any command the solid modeler supports.

Example:

```
INCLUDE( "general.irt" );
```

includes the file "general.irt".

10.4.11 INTERACT

```
INTERACT( GeometryTreeType Object )
```

A user-defined function (see iritinit.irt) that does the following, in order:

1. Clear the display device.
2. Display the given **Object**.
3. Pause for a keystroke.

This user-defined function in version 4.0 of *IRIT* is an emulation of the INTERACT function that used to exist in previous versions.

Example:

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

displays and interacts with the object **Obj** and the predefined object **Axes**. VIEW_MAT will be used to set the starting transformation.

See VIEW and VIEWOBJ for more.

10.4.12 LIST

```
ListType LIST( AnyType Elem1, AnyType Elem2, ... )
```

Constructs an object as a list of several other objects. Only a reference is made to the Elements, so modifying Elem1 after being included in the list will affect Elem1 in that list next time list is used!

Each inclusion of an object in a list increases its internal **used** reference. The object is freed iff in **used** reference is zero. As a result, attempt to delete a variable (using FREE) which is referenced in a list removes the variable, but the object itself is freed only when the list is freed.

10.4.13 LOAD

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

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

Under unix, compressed files can be loaded if the given file name has a postfix of ".Z". The unix system's "zcat" will be invoked via a pipe for that purpose.

```
factorial = function(x):return = x; # Dummy function.
factorial = function(x):
    if (x <= 1, return = 1, return = x * factorial(x - 1));
```

Overloading is valid inside a function as it is outside. For example, for

```
add = FUNCTION(x, y):
   return = x + y;
```

the following function calls are all valid:

```
add(1, 2);
add(vector(1,2,3), point(1,2,3));
add(box(vector(-3, -2, -1), 6, 4, 2), box(vector(-4, -3, -2), 2, 2, 4));
```

Finally, here is a more interesting example that computes an approximation of the length of a curve, using the sqr function defined above:

```
distptpt = FUNCTION(pt1, pt2):
    return = sqrt(sqr(coord(pt1, 1) - coord(pt2, 1)) +
                  sqr(coord(pt1, 2) - coord(pt2, 2)) +
                  sqr(coord(pt1, 3) - coord(pt2, 3)));

crvlength = FUNCTION(crv, n):pd:t:t1:t2:dt:pt1:pt2:i:
    return = 0.0:
    pd = pdomain(crv):
    t1 = nth(pd, 1):
    t2 = nth(pd, 2):
    dt = (t2 - t1) / n:
    pt1 = coerce(ceval(crv, t1), e3):
    for (i = 1, 1, n,
         pt2 = coerce(ceval(crv, t1 + dt * i), e3):
         return = return + distptpt(pt1, pt2):
         pt1 = pt2);
```

Try, for example:

```
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 30) / 2;
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 100) / 2;
crvlength(circle(vector(0.0, 0.0, 0.0), 1.0), 300) / 2;
```

See PROCEDURE for more.

10.4.9 IF

```
IF( NumericType Cond, AnyType TrueBody { , AnyType FalseBody } )
```

Executes **TrueBody** (group of regular commands, separated by COLONs - see FOR loop) if the **Cond** holds, i.e., it is a numeric value other than zero, or optionally, if it exists, executes **FalseBody** if the **Cond** does not hold, i.e., it evaluates to a numeric value equal to zero.

Examples:

```
IF ( machine == IBMOS2, resolution = 5, resolution = 10 );
IF ( a > b, max = a, max = b );
```

sets the resolution to be 10, unless running on an IBMOS2 system, in which case the resolution variable will be set to 5 in the first statement, and set max to the maximum of a and b in the second statement.

```
step = 10;
rotstepx = rotx(step);
FOR ( a = 1, 1, 360 / step,
    view_mat = rotstepx * view_mat:
    view( list( view_mat, b, axes ), ON )
);
```

Displays b and axes with a view direction that is rotated 10 degrees at a time around the X axis.

10.4.6 HELP

```
HELP( StringType Subject )
```

Provides help on the specified Subject.
Example:

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

will list all *IRIT* help subjects.

10.4.7 FREE

```
FREE( GeometricType Object )
```

Because of the usually huge size of geometric objects, this procedure may be used to free them. Reassigning a value (even of different type) to a variable automatically releases the old variable's allocated space as well.

10.4.8 FUNCTION

```
FuncName = FUNCTION(Prm1, Prm2, ... , PrmN):LclVal1:LclVar2: ... :LclVarM:
    FuncBody;
```

Defines a function named FuncName with N parameters and M local variables (N, M ¿= 0). Here is a (simple) example of a function with no local variables and a single parameter that computes the square of a number:

```
sqr = FUNCTION(x):
   return = x * x;
```

Functions can be defined with optional parameters and optional local variables. A function's body may contain an arbitrary set of expressions including for loops, (user) function calls, or even recursive function calls, all separated by colons. The returned value of the function is the value of an automatically defined local variable named return. The return variable is a regular local variable within the scope of the function and can be used as any other variable.

If a variable's name is found in neither the local variable list nor the parameter list, it is searched in the global variable list (outside the scope of the function). Binding of names of variables is static as in the C programming language.

Because binding of variables is performed in execution time, there is a somewhat less restrictive type checking of parameters of functions that are invoked within a user's defined function.

A function can invoke itself, i.e., it can be recursive. However, since a function should be defined when it is called, a dummy function should be defined before the recursive one is defined:

```
ATTRIB(Glass, "rgb", "255,0,0");
ATTRIB(Glass, "reflect", 1.4);
```

sets the RGB color of the **Glass** object.

Attribute names are case insensitive. Spaces are allowed in the **Value** string, as well as the double quote itself, although the latter must be escaped:

```
ATTRIB(Glass, "text", "Say \"this is me\"");
```

10.4.2 COLOR

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

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

See the ATTRIB command for more fine control of colors using the RGB attribute.

10.4.3 COMMENT

```
COMMENT
```

Two types of comments are allowed:

1. One-line comment: starts anywhere in a line at the '#' character, up to the end of the line.

2. Block comment: starts at the COMMENT keyword followed by a unique character (anything but white space), up to the second occurrence of that character. This is a fast way to comment out large blocks.

Example:

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

10.4.4 EXIT

```
EXIT();
```

Exits from the solid modeler. NO warning is given!

10.4.5 FOR

```
FOR( NumericType Start, NumericType Increment, NumericType End, AnyType Body )
```

Executes the **Body** (see below), while the FOR loop conditions hold. **Start, Increment, End** are evaluated first, and the loop is executed while <= **End** if **Increment** > **0**, or while >= **End** if **Increment** < **0**. If **Start** is of the form "Variable = Expression", then that variable is updated on each iteration, and can be used within the body. The body may consist of any number of regular commands, separated by COLONs, including nesting FOR loops to an arbitrary level.

Example:

```
for ( a = 1, 1, 720 / step,
      view_mat = save_mat *
                 HOMOMAT( list( list( 1, 0, 0, 0 ),
                                list( 0, 1, 0, 0 ),
                                list( 0, 0, 1, -a * step / 500 ),
                                list( 0, 0, 0, 1 ) ) ):
      view(list(view_mat, b, axes), on)
);
```

looping and viewing through a sequence of perspective transforms, created using the HOMOMAT constructor.

10.3.2 ROTX

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

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

10.3.3 ROTY

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

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

10.3.4 ROTZ

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

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

10.3.5 SCALE

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

Creates a scaling by the **ScaleFactors** transformation matrix.

10.3.6 TRANS

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

Creates a translation by the **TransFactors** transformation matrix.

10.4 General purpose functions

10.4.1 ATTRIB

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

or

ATTRIB( AnyType Object, StringType Name, RealType Value )
```

Provides a mechanism to add a string or numeric attribute to an **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,

10.2.73 SYMBDIFF

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

or

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

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

Example:

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

10.2.74 TORUS

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

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

Example:

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

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

10.3 Object transformation functions

All the routines in this section construct a 4 by 4 homogeneous transformation matrix representing the required transform. These matrices may be concatenated to achieve more complex transforms using the matrix multiplication operator $*$. For example, the expression

```
m = trans( vector( -1, 0, 0 ) ) * rotx( 45 ) * trans( vector( 1, 0, 0 ) );
```

constructs a transform to rotate an object around the X = 1 line, 45 degrees. A matrix representing the inverse transformation can be computed as:

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

See also overloading of the - operator.

10.3.1 HOMOMAT

```
MatrixType HOMOMAT( ListType MatData )
```

Creates an arbitrary homogeneous transformation matrix by manually providing its 16 coefficients. Example:

10.2.70 SYMBDPROD

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

or

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

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

Example:

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

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

10.2.71 SYMBCPROD

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

or

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

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

Example:

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

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

10.2.72 SYMBSUM

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

or

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

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

Example:

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

is important, should improve the accuracy of the output. The parametric domains of **ScaleCrv** and **FrameCrv** do not have to match the parametric domain of **Axis**, and their domains are made compatible by this function.

Example:

```
Cross = arc( vector( 0.2, 0.0, 0.0 ),
             vector( 0.2, 0.2, 0.0 ),
             vector( 0.0, 0.2, 0.0 ) ) +
        arc( vector( 0.0, 0.4, 0.0 ),
             vector( 0.1, 0.4, 0.0 ),
             vector( 0.1, 0.5, 0.0 ) ) +
        arc( vector( 0.8, 0.5, 0.0 ),
             vector( 0.8, 0.3, 0.0 ),
             vector( 1.0, 0.3, 0.0 ) ) +
        arc( vector( 1.0, 0.1, 0.0 ),
             vector( 0.9, 0.1, 0.0 ),
             vector( 0.9, 0.0, 0.0 ) ) +
        ctlpt( E2, 0.2, 0.0 );
Axis = arc( vector( -1.0, 0.0, 0.0 ),
            vector(  0.0, 0.0, 0.1 ),
            vector(  1.0, 0.0, 0.0 ) );
Axis = crefine( Axis, FALSE, list( 0.25, 0.5, 0.75 ) );
ScaleCrv = cbezier( list( ctlpt( E2, 0.0, 0.01 ),
                          ctlpt( E2, 1.0, 0.5 ),
                          ctlpt( E2, 2.0, 0.01 ) ) );
Srf1 = SWEEPSRF( Cross, Axis, ScaleCrv, OFF );
Srf2 = SWEEPSRF( Cross, Axis, ScaleCrv, vector( 0.0, 1.0, 1.0 ) );
Srf3 = SWEEPSRF( Cross, Axis, ScaleCrv,
                 cbezier( list( ctlpt( E3,  1.0, 0.0, 0.0 ),
                                ctlpt( E3,  0.0, 1.0, 0.0 ),
                                ctlpt( E3, -1.0, 0.0, 0.0 ) ) ) );
```

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

10.2.69 SYMBPROD

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

or

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

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

Example:

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

10.2.66 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 **Direction**. The returned vector has a unit length.

Example:

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

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

10.2.67 SURFREV

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

or

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

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

Example:

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

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

10.2.68 SWEEPSRF

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

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

The resulting sweep is only an approximation of the real sweep. The scaling and axis placement will not be exact, in general. Refinement of the axis curve at the proper location, where accuracy

10.2.62 SPHERE

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

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

10.2.63 SRAISE

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

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

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

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

10.2.64 SREFINE

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

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

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

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

10.2.65 SREGION

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

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

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

extracts the region 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.

10.2.58 SMERGE

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

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

Example:

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

10.2.59 SMORPH

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

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

Example:

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

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

10.2.60 SNORMAL

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

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

Example:

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

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

10.2.61 SNRMLSRF

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

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

Example:

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

10.2.55 SEDITPT

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

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

Example:

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

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

10.2.56 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 contained in the surface parametric domain if **Srf** is a Bspline surface, or between zero and one if **Srf** is a Bezier surface. The returned control point has the same type as the control points of **Srf**.

Example:

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

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

10.2.57 SFROMCRVS

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

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

The surface interpolates the first and last curves only.

Example:

```
Crv1 = cbspline( 3,
                 list( ctlpt( E3, 0.0, 0.0, 0.0 ),
                       ctlpt( E3, 1.0, 0.0, 0.0 ),
                       ctlpt( E3, 1.0, 1.0, 0.0 ) ),
                 list( KV_OPEN ) );
Crv2 = Crv1 * trans( vector( 0.0, 0.0, 1.0 ) );
Crv3 = Crv2 * scale( vector( 0.0, 0.0, 2.0 ) )
            * trans( vector( 0.1, 0.1, 0.1 ) );
Srf = SFROMCRVS( list( Crv1, Crv2, Crv3 ), 3 );
```

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

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

10.2.53 SDERIVE

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

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

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

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

10.2.54 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 surface in which **Param** must be conatined in the parametric domain of the surface, or a Bezier surface in which **Param** must be in the range of zero to one.

It returns a list of 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.

10.2.51 SBEZIER

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

Creates a Bezier surface using the provided control mesh. **CtlMesh** is a list of rows, each of which is a list of control points. All control points must be of the same point type.

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

$$S(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{ij} B_i(u) B_j(v) \tag{8}$$

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

Example:

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

10.2.52 SBSPLINE

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

Creates a Bspline surface from the provided **UOrder** and **VOrder** orders, the control mesh **CtlMesh**, and the two knot vectors **KnotVectors**. **CtlMesh** is a list of rows, each of which is a list of control points. All control points must be of the same point type. **KnotVectors** is a list of two knot vectors. Each knot vector 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 the appropriate length and with open or floating end condition will be constructed automatically.

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

$$S(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} P_{ij} B_{i,\chi}(u) B_{j,\xi}(v) \tag{9}$$

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

Example:

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

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

10.2.49 PRISA

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

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

Example:

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

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

10.2.50 RULEDSRF

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

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

Example:

```
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.

10.2.46 OFFSET

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

or

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

Offsets **Crv** or **Srf**, by translating all the control points in the direction of the normal of the curve or surface by an **OffsetDistance** amount. Each control point has a *node* parameter value associated with it, which is used to compute the normal. The returned curve or surface only approximates the real offset. One may improve the offset accuracy using refinement (See AOFFSET). 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. See also AOFFSET.

10.2.47 PDOMAIN

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

or

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

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

Example:

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

10.2.48 POLY

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

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

Example:

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

10.2.43 GPOLYLINE

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

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

Example:

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

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

10.2.44 NIL

```
ListType NIL()
```

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

10.2.45 MERGPOLY

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

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

Example:

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

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 ) );
```

10.2.40 FFCOMPAT

```
FFCOMAPT( CurveType Crv1, CurveType Crv2 )
```

or

```
FFCOMAPT( SurfaceType Srf1, SurfaceType Srf2 )
```

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

Example:

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

See also SMORPH.

10.2.41 GBOX

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

Creates a parallelepiped - Generalized BOX polygonal object, defined by **Point** as base position, and **Dx, Dy, Dz** as 3 3D vectors to define the 6 faces of this generalized BOX. The regular BOX object is a special case of GBOX where **Dx** = vector(Dx, 0, 0), **Dy** = vector(0, Dy, 0), and **Dz** = vector(0, 0, Dz).

Dx, **Dy**, **Dz** must all be independent in order to create an object with positive volume.

Example:

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

10.2.42 GPOLYGON

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

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

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

Example:

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

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

10.2.38 CZEROS

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

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

Example:

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

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

10.2.39 EXTRUDE

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

or

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

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

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

Example:

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

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

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

Example:

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

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

10.2.35 CTANGENT

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

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

Example:

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

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

10.2.36 CTLPT

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

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

Examples:

```
CPt1 = CTLPT( E3, 0.0, 0.0, 0.0 );
CPt2 = CTLPT( P2, 0.707, 1.414, 1.414 );
```

constructs an **E3** point at the origin and a P2 rational point with a weight of 0.707.

10.2.37 CYLIN

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

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

Example:

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

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

10.2.32 CRVLNDST

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

or

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

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

Example:

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

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

10.2.33 CRVPTDST

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

or

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

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

Example:

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

finds the farthest point on **Crv** from point **Pt**.

10.2.34 CSURFACE

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

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

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

10.2.28 CRAISE

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

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

10.2.29 CREFINE

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

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

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

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

10.2.30 CREGION

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

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

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

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

10.2.31 CROSSEC

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

This feature is NOT implemented.

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

Example:

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

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

See also CON2.

10.2.26 CONVEX

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

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

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

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

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

10.2.27 COORD

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

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

Example:

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

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

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

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

See also COERCE.

10.2.23 COMPOSE

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

or

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

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

Example:

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

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

10.2.24 CON2

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

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

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

Example:

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

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

10.2.25 CONE

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

10.2.19 CIRCLE

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

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

10.2.20 CIRCPOLY

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

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

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

10.2.21 CMESH

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

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

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

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

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

and similar for the other parametric direction.

Example:

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

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

10.2.22 COERCE

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

Provides a coercion mechanism between different objects or object types. PointType, VectorType, PlaneType, CtlPtType can be all coerced to each other by using the **NewType** of POINT_TYPE, VECTOR_TYPE, PLANE_TYPE, or one of E1-E5, P1-P5 (CtlPtType). Similarly, CurveType and SurfaceType can be coerced to hold different CtlPtType of control points.

Example:

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

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

10.2.16 CEVAL

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

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

Example:

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

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

10.2.17 CEXTREMES

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

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

Example:

```
extremes = cextremes( crv, 0.0001, 1 );
```

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

10.2.18 CINFLECT

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

or

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

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

The sign of curvature scalar field is equal to,

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

Example:

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

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

```
for ( i = 1, 1, listsize( crvtr ),
    pt = ceval( crv, nth( crvtr, i ) ) ):
    snoc( pt, pt_crvtr )
);
interact( list( crv, pt_crvtr ) );
```

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

10.2.13 CDERIVE

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

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

Example:

```
Hodograph = CDERIVE( Crv );
```

10.2.14 CDIVIDE

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

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

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

Example:

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

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

10.2.15 CEDITPT

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

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

Example:

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

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

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

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

10.2.12 CCRVTR

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

or

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

Computes the extreme curvature points on **Crv** in the XY plane. This set includes not only points of maximum (convexity) and mimumum (concavity) curvature, but also points of zero curvature such as inflection points. Since this operation is partially numeric, **Epsilon** is used to set the needed accuracy. It returns the parameter value(s) of the location(s) with extreme curvature along the **Crv**. If, however, **Epsilon** is negative, the curvature scalar field curve is returned as a two dimensional rational vector field curve, for which the first dimension is equal to the parameter, and the second is the curvature value at that parameter.

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

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

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

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

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

See also CZEROS, CEXTREMES, and CCRVTR.

Example:

```
crv = cbezier( list( ctlpt( E2, -1.0,  1.0 ),
                     ctlpt( E2, -0.5, -2.0 ),
                     ctlpt( E2,  0.0,  2.0 ),
                     ctlpt( E2,  1.0, -1.0 ) ) );

crvtr = ccrvtr( crv, 0.001 );
pt_crvtr = nil();
pt = nil();
```

10.2.10 CBSPLINE

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

Creates a Bspline curve out of the provided control point list, the knot vector, and the specified order. **CtlPtList** is a list of control points, all of which must be of the same type (E1-E5, P1-P5), defining the curve's control polygon. The length of the **KnotVector** must be equal to the number of control points in **CtlPtList** plus the **Order**. The knot vector list may be specified as either **list(KV_OPEN)** or **list(KV_FLOAT)** in which a uniform open or uniform floating knot vector with the appropriate length is automatically constructed. The created curve is the piecewise polynomial (or rational),

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

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

Example:

```
s45 = sin(pi / 4);
HalfCirc = CBSPLINE( 3,
                  list( ctlpt( P3, 1.0,  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.

10.2.11 CCINTER

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

or

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

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

Example:

```
crv1 = cbspline( 3,
                 list( ctlpt( E2, 0, 0 ),
                       ctlpt( E2, 0.0.5 ),
                       ctlpt( E2, 0.5, 0.7 ),
```

10.2.7 BZR2BSP

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

or

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

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

Example:

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

10.2.8 BSP2BZR

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

or

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

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

Example:

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

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

10.2.9 CBEZIER

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

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

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

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

Example:

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

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

10.2.5 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 end points of the four curves should match as follows:

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

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

Example:

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

10.2.6 BOX

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

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

Example:

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

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

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

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

10.2.2 ARC

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

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

Example:

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

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

10.2.3 AOFFSET

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

or

SurfaceType AOFFSET( SurfaceType Srf NumericType OffsetDistance,
                     NumericType Epsilon, NumericType TrimLoops )
```

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

Example:

```
OffCrv = AOFFSET( Crv, 0.5, 0.03, TRUE );
```

computes an adaptive offset to **Crv** with **OffsetDistance** of 0.5 and **Epsilon** of 0.03 and trims the self-intersection loops. See also OFFSET.

10.2.4 BOOLONE

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

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

Example:

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

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

10.1.13 SIN

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

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

10.1.14 SQRT

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

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

10.1.15 TAN

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

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

10.1.16 THISOBJ

```
NumericType THISOBJ( AnyType Object )
```

Returns the object type of the given **Object**. This can be one of the constants NUMERIC_TYPE, STRING_TYPE, VECTOR_TYPE, POINT_TYPE, CTLPT_TYPE, MATRIX_TYPE, POLY_TYPE, CURVE_TYPE, or SURFACE_TYPE.

10.1.17 VOLUME

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

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

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

10.2 GeometricType returning functions

10.2.1 ADAPISO

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

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

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

10.1.5 ATAN

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

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

10.1.6 ATAN2

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

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

10.1.7 COS

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

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

10.1.8 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 LISTSIZE

```
NumericType LISTSIZE( ListType List | PolyType Poly )
```

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

Example:

```
len = LISTSIZE( list( 1, 2, 3 ) );
numPolys = LISTSIZE( axes );
```

will assign the value of 3 to the variable **len**, and set **numPolys** to the number of polylines in the axes object.

10.1.11 LN

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

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

10.1.12 LOG

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

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

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

9.10 Grammar

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

10 Function Description

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

10.1 NumericType returning functions

10.1.1 ABS

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

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

10.1.2 ACOS

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

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

10.1.3 AREA

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

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

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

10.1.4 ASIN

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

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

9.5 Overloading ^

The ^ operator is overloaded above the following domains:

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

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

9.6 Assignments

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

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

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

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

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

9.8 Logical operators &&, |||, !

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

9.9 Priority of operators

The following table lists the priority of the different operators.

9.2 Overloading –

The – operator is overloaded above the following domains:

As a binary operator:

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

As a unary operator:

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

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

9.3 Overloading *

The * operator is overloaded above the following domains:

```
NumericType * NumericType   -> NumericType
VectorType  * NumericType   -> VectorType    (Vector scaling)
VectorType  * VectorType    -> NumericType   (Inner product)
MatrixType  * NumericType   -> MatrixType    (Matrix Scaling)
MatrixType  * VectorType    -> VectorType    (Vector transformation)
MatrixType  * MatrixType    -> MatrixType    (Matrix multiplication)
MatrixType  * GeometricType -> GeometricType (Object transformation)
MatrixType  * ListType      -> ListType      (Object hierarchy transform.)
PolygonType * PolygonType   -> PolygonType   (Boolean INTERSECTION operation)
```

Note: Boolean INTERSECTION of two disjoint objects (no common volume) will result with an empty object. Object hierarchy transform transforms any transformable object (GeometricType) found in the list recursively.

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.

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

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

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

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

Example:

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

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

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

9 Operator overloading

The basic operators $+$, $-$, $*$, $/$, and $\hat{}$ are overloaded. This section describes what action is taken by each of these operators depending on its arguments.

9.1 Overloading +

The + operator is overloaded above the following domains:

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

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

Functions that create linear transformation matrices:

HOMOMAT	ROTX	ROTY	ROTZ	SCALE	TRANS

Miscellaneous functions:

ATTRIB	FOR	INCLUDE	NTH	SNOC	VIEW
CHDIR	FREE	INTERACT	PAUSE	SYSTEM	VIEWOBJ
COLOR	FUNCTION	LIST	PRINTF	TIME	
COMMENT	HELP	LOAD	PROCEDURE	VARLIST	
EXIT	IF	LOGFILE	SAVE	VECTOR	

Variables that are predefined in the system:

AXES	DUMPLVL	INTERCRV	PRSP_MAT
COPLANAR	ECHOSRC	MACHINE	RESOLUTION
DRAWCTLPT	FLAT4PLY	POLYSORT	VIEW_MAT

Constants that are 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, with the same precedence as in C.

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

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

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

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

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

+	BOOLSUM	CONVEX	GBOX	PROCEDURE	SREFINE
−	BOX	COORD	GPOLYGON	ROTX	SREGION
*	BSP2BZR	COS	GPOLYLINE	ROTY	STANGENT
/	BZR2BSP	COMPOSE	HELP	ROTZ	SURFREV
^	CBEZIER	CPOLY	HOMOMAT	RULEDSRF	SWEEPSRF
=	CBSPLINE	CRAISE	IF	SAVE	SYMBCPROD
==	CCINTER	CREFINE	INCLUDE	SBEZIER	SYMBDIFF
!=	CCRVTR	CREGION	INTERACT	SBSPLINE	SYMBDPROD
<	CDERIVE	CROSSEC	LIST	SCALE	SYMBPROD
>	CDIVIDE	CRVLNDST	LISTSIZE	SDERIVE	SYMBSUM
<=	CEDITPT	CRVPTDST	LN	SDIVIDE	SYSTEM
>=	CEVAL	CSURFACE	LOAD	SEDITPT	TAN
ABS	CEXTREMES	CTANGENT	LOG	SEVAL	TIME
ACOS	CHDIR	CTLPT	LOGFILE	SFROMCRVS	THISOBJ
ADAPISO	CINFLECT	CYLIN	MERGEPOLY	SIN	TORUS
ARC	CIRCLE	CZEROS	NIL	SMERGE	TRANS
AREA	CIRCPOLY	EXIT	NTH	SMORPH	VARLIST
ASIN	CMESH	EXP	OFFSET	SNOC	VECTOR
ATAN	COERCE	EXTRUDE	PAUSE	SNORMAL	VIEW
ATAN2	COLOR	FFCOMPAT	PDOMAIN	SNRMLSRF	VIEWOBJ
ATTRIB	COMMENT	FOR	POLY	SPHERE	VOLUME
AOFFSET	CON2	FREE	PRINTF	SQRT	
BOOLONE	CONE	FUNCTION	PRISA	SRAISE	

7 Functions and Variables

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

Functions that return a **NumericType**:

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

Functions that return a **GeometricType**:

ADAPISO	CDIVIDE	COORD	EXTRUDE	SBEZIER	SREFINE
ARC	CEDITPT	CPOLY	FFCOMPAT	SBSPLINE	SREGION
AOFFSET	CEVAL	CRAISE	GBOX	SDERIVE	STANGENT
BOOLONE	CEXTREMES	CREFINE	GPOLYGON	SDIVIDE	SURFREV
BOOLSUM	CINFLECT	CREGION	GPOLYLINE	SEDITPT	SWEEPSRF
BOX	CIRCLE	CROSSEC	MERGEPOLY	SEVAL	SYMBCPROD
BSP2BZR	CIRCPOLY	CRVLNDST	NIL	SFROMCRVS	SYMBDIFF
BZR2BSP	CMESH	CRVPTDST	OFFSET	SMERGE	SYMBDPROD
CBEZIER	COERCE	CSURFACE	PDOMAIN	SMORPH	SYMBPROD
CBSPLINE	COMPOSE	CTANGENT	POLY	SNORMAL	SYMBSUM
CCINTER	CON2	CTLPT	PRISA	SNRMLSRF	TORUS
CCRVTR	CONE	CYLIN	PROCEDURE	SPHERE	
CDERIVE	CONVEX	CZEROS	RULEDSRF	SRAISE	

5 Data Types

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

ConstantType	Scalar real type that cannot be modified.
NumericType	Scalar real type.
VectorType	3D real type vector.
PointType	3D real type point.
CtlPtType	Control point of a freeform curve or surface.
MatrixType	4 by 4 matrix (homogeneous transformation matrix).
PolygonType	Object consists of polygons.
PolylineType	Object consists of polylines.
CurveType	Object consists of curves.
SurfaceType	Object consists of surfaces.
GeometricType	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. List size is dynamically increased, as needed.

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

6 Commands summary

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

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

4 First Usage

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

Some important commands to begin with are,

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

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

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

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

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

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

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

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

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

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

3.1 IBM PC OS2 Specific Set Up

Under OS2 the IRIT_DISPLAY environment variable must be set (if set) to os2drvs.exe without any option (-s- will be passed automatically). os2drvs.exe must be in a directory that is in the PATH environment variable. IRIT_POS can be set to the X1 Y1 X2 Y2 dimensions of the window. IRIT_BIN_IPC can be used to signal binary IPC which is faster. Here is a complete example:

```
set IRIT_PATH=c:\irit\bin\
set IRIT_DISPLAY=os2drvs.exe
set IRIT_POS=230 120 550 460
set IRIT_BIN_IPC=1
```

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

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

3.2 IBM PC Window NT Specific Set Up

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

3.3 Unix Specific Set Up

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

predefined functions and procedures if you have some. This file will be searched much the same way **IRIT.CFG** is. The name of this initialization file may be changed by setting the StartFile entry in the configuration file. This file is far more important starting at version 4.0, because of the new function and procedure definition that has been added, and which is used to emulate BEEP, VIEW, and INTERACT for example.

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

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

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

For example,

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

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

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

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

IRIT_MALLOC	If set, apply dynamic memory consistency testing. Programs will execute much slower in this mode.
IRIT_DEBUG_FUNC	prints debugging information on user's defined functions. If ¿0, invocation and leaving of functions. If ¿2, parameters and returned values as well. If ¿4 global variable list is printed as well on invocation.
IRIT_SERVER_HOST	Internet Name of IRIT server (used by graphics driver).
IRIT_SERVER_PORT	Set a different socket port number than the default.

For example,

```
setenv IRIT_MALLOC 1
setenv IRIT_SERVER_HOST irit.cs.technion.ac.il
setenv IRIT_SERVER_PORT 5432
```

IRIT_MALLOC is useful for programmers, or when reporting a memory fatal error occurrence. The IRIT_SERVER_HOST/PORT controls the server/client (*IRIT*/Display device) communication.

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

See the section on the graphics drivers for more details.

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

The following command line options are available:

1 Introduction

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

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

2 Copyrights

3 Command Line Options and Set Up of *IRIT*

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

In addition, if it exists, a file by the name of **iritinit.irt** will be automatically executed before any other '.irt' file. This file may contain any *IRIT* command. It is the proper place to put your

Contents

IRIT 4.0 User's Manual

A Solid modeling Program

EMail: gershon@cs.technion.ac.il

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

This manual is for IRIT version 4.0.