

# SurfaceWorks 5.0 Reference: What's New

<b>SURFACEWORKS 5.0 REFERENCE: WHAT'S NEW</b> .....	<b>1</b>
<b>What's New in SurfaceWorks 5.0</b> .....	<b>3</b>
<b>The Property Manager</b> .....	<b>3</b>
Using the Property Manager Effectively .....	4
The Selection Set.....	4
The Surfer View .....	4
<b>Advanced Modeling</b> .....	<b>5</b>
<b>NUBFitSurf Command</b> .....	<b>5</b>
Examples .....	6
The Command .....	6
<b>New Entities:</b> .....	<b>7</b>
<b>Copy family</b> .....	<b>7</b>
CopyPoint, CopyCurve, CopySurf .....	7
CopyBead, CopyRing.....	8
CopyMagnet .....	9
<b>Rotated Curve</b> .....	<b>10</b>
<b>Rotated Point</b> .....	<b>11</b>
<b>Rotated Surface</b> .....	<b>12</b>
<b>Intersection Entities</b> .....	<b>13</b>
Intersection Bead.....	16
Intersection Magnet.....	17
<b>Offset Curve</b> .....	<b>18</b>
<b>New Procedural Entities</b> .....	<b>21</b>
Procedural Curve.....	21
Procedural Snake .....	22
<b>RPYFrame</b> .....	<b>24</b>
<b>BGraph — B-spline Graph</b> .....	<b>25</b>
<b>Variable Support</b> .....	<b>26</b>
Constants and variables .....	26
Variables.....	26
Variables as supports.....	28
User interface .....	29
Interaction with object transformations .....	30



P.O. Box 684 / 54 Herrick Rd.  
Southwest Harbor, Maine 04679 U.S.A.  
voice 207-244-4100  
fax 207-244-4171  
email [support@aerohydro.com](mailto:support@aerohydro.com)  
website [www.aerohydro.com](http://www.aerohydro.com)

© 2005 AeroHydro, Inc.

---

# What's New in SurfaceWorks 5.0

The new functionality added to the release of SurfaceWorks 5.0 is the following:

- A “Property Manager” for Insertion and Editing
- Advanced Modeling Option
- New Headings in the Surfer View
- All RG Entities editable through the Property Manager.
- Command to break surfaces with breaklines into separate surfaces.
- New Entities

---

## The Property Manager

The biggest change to SurfaceWorks is the adoption of the “Property Manager”. (Fig.1) An example is shown below for a Copy Surface. All fields are directly editable and in most cases the changes take place with no confirmation needed. This Property Manager appears upon selection of the object. If more than one object is selected a Multiple Edit Property Manager (Fig.2) appears. One notable improvement to Multiple Edit is the ability to change the weights of multiple objects. User Data is a new attribute, editable in both the Property Manager and Multiple Edit. A text string of up to 40 characters can be entered here.

The same Property Manager is used for the Insertion of objects (shown). One difference between Insertion and Editing is that a confirmation would be needed to complete the Insert process.

Fig.1

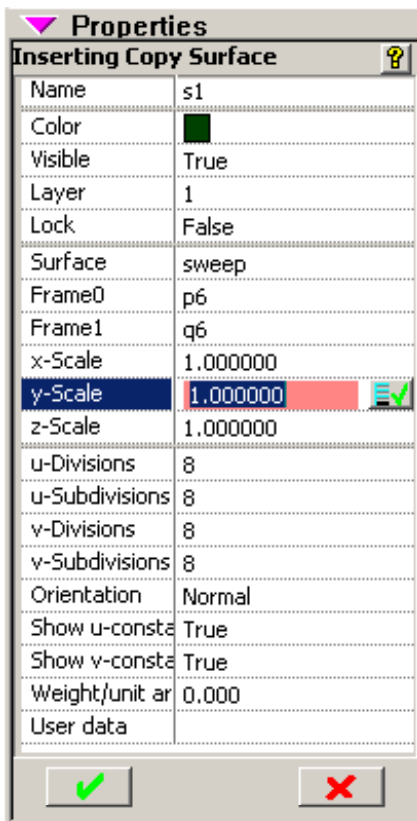
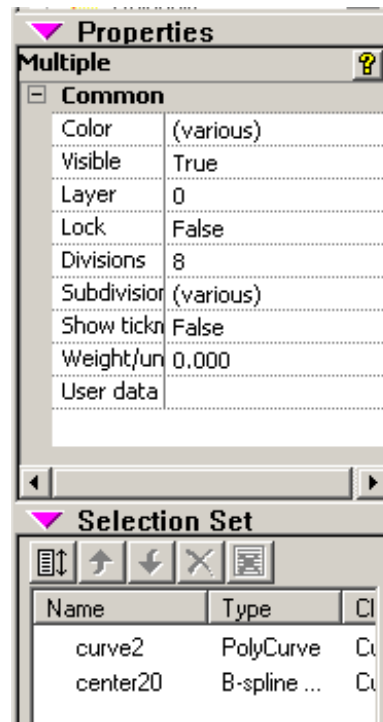


Fig.2



## Using the Property Manager Effectively

Using the Property Manager for insertion of new entities is pretty straightforward. Select the entity from the menu, change the properties to suit the situation, and click the green check mark to complete the creation of the entity.

The process of editing existing entities showcases the functionality of the Property Manager much better. The first thing you will notice is that a change made to most Attribute fields results in an immediate change to the entity. In the above Copy Surface example this would include all fields except those that reside under the 'Geometry' sub-heading. A minor exception to that would be fill-in attributes like Divisions or User Data. The program needs to know when the user has finished data entry. Pressing <Enter> or tabbing to the next field will complete the data entry.

In the case of the Copy Surface the parents are added, by selecting an item to have it added to replace an existing item, or by filling in a value for scale factors. When filling in a value field it is easiest to simply confirm with a press of the <Enter> button on the keyboard, but a check-mark button is provided as well. (Fig. 1)

When you select a Support field you enter what we call "Support Picking Mode" and can only get out of the mode by offering a confirmation. The confirmation methods are clicking the green check-mark in the field, pressing <Enter>, or tabbing to the next field. The fact that a confirmation is needed can also be a drawback to streamlined editing, so a method was implemented of querying supports without entering "Support Picking Mode". The parent field is actually two different fields, the name or title, and the value. If you click in the value field, you are switched to the mode of picking supports, but if your intention to just make a query of the support, simply click in the name field. If the support(s) is visible in the model they will be highlighted for your examination.

In previous versions of SurfaceWorks, if there were multiple objects in the selection set, and the user desired to edit one particular object in the set (Fig. 3), double clicking would invoke an Edit dialog. The new scheme does not have Edit Dialogs, so an alternate method needed to be employed.

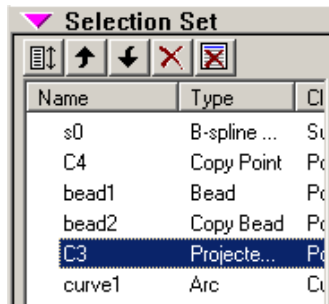


Fig. 3

### The Selection Set

A new button has been added to the selection set.



Clicking this button will clear the selection set of all but the selected item, enabling access to the Property Manager.

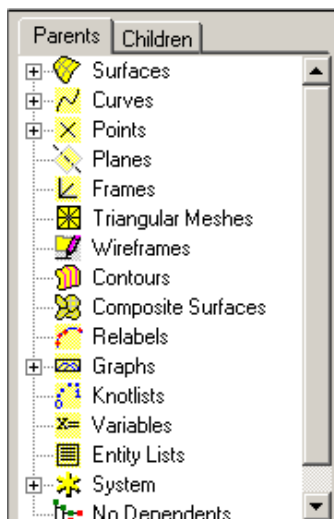


Fig. 4

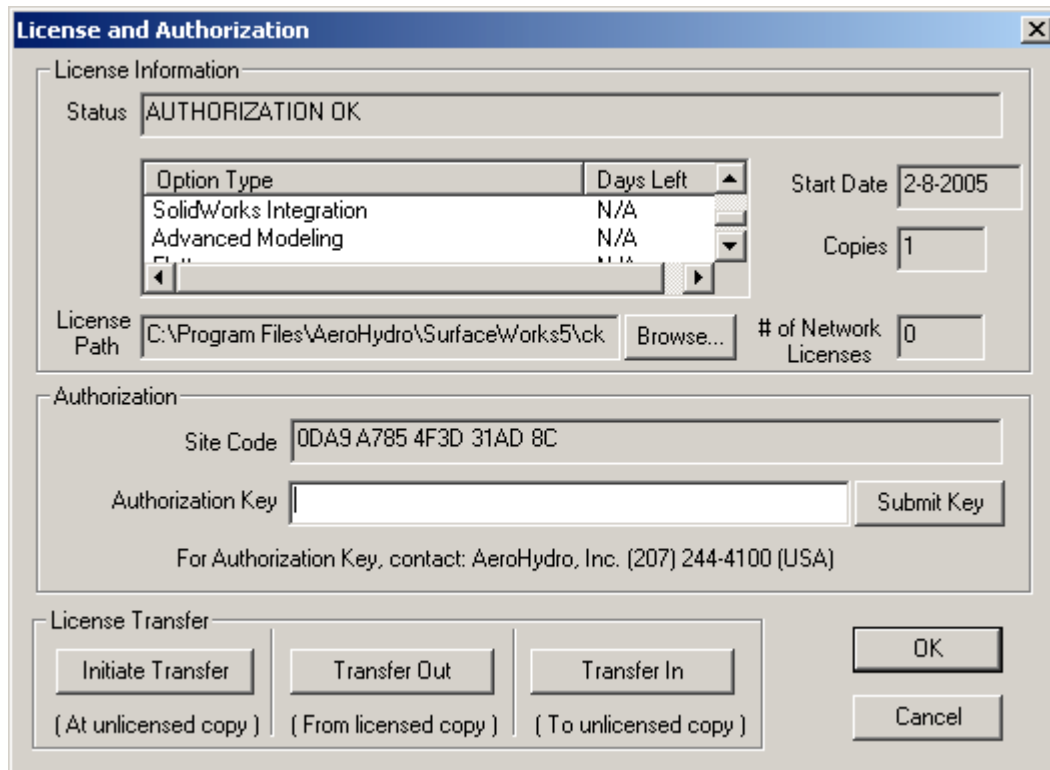
### The Surfer View

New classes of entities have been added to SurfaceWorks 5.0. They are reflected in the Surfer View (Fig. 4) with the new headings; Graphs, Knotlists, and Variables. A heading for System objects has been added to the Surfer View as well. This should make it easier to access planes for mirrors, as one example.

---

## Advanced Modeling

A new option has been added to the SurfaceWorks License. If Advanced Modeling is authorized, an additional set of new entities and functionality will be added to the program. When this scheme was originally conceived, it was thought the Advanced Modeling option would enable SurfaceWorks to have all the functionality of a released version of MultiSurf. The reality at this time is all entities can be turned on with Advanced Modeling, but many of the special tools and commands are yet to be ported to the SurfaceWorks User Interface.



---

## NUBFitSurf Command

**Nonuniform B-spline fitting of a surface, taking breaklines into account.**

The surface is broken along breaklines into "panes". The "panes" are fitted with compatible NUBS surfaces, and are (optionally) reassembled into a single NURBS surface with multiple knots as needed at

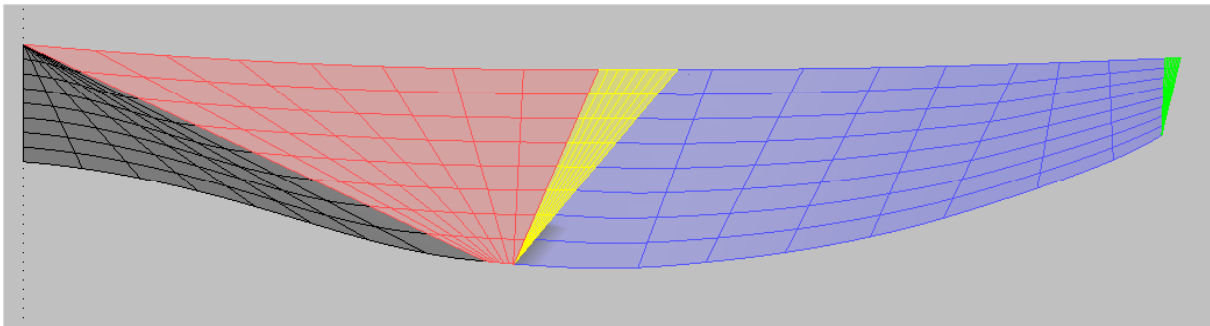
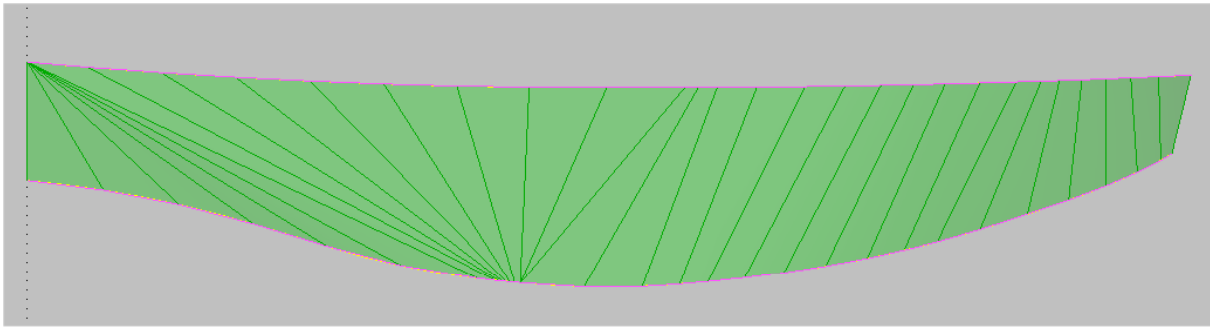
the breaklines. It has been our experience, if the surface is intended for export, the surface should be left as separate panes. In most cases these panels can be re-assembled in the destination CAD package.

In SolidWorks: Insert/Surface/Knit

One big caveat: These new surfaces DO NOT update with the parent surface.

## Examples

The top figure shows a typical developable side panel for a metal boat hull, which would be very hard to approximate in an IGES export or a transfer to SolidWorks. In the lower figure, the panel has been divided into 5 separate surfaces. The command is detailed below the images.



## The Command

**NUBSurfFit**[ utype [ nulim [ vtype [ nvlm [ tol [assemble [lt]]]]]]]

**utype** (default 3 = cubic) The degree of B-Spline used to fit the surface in the u-direction

**nulim**" The max number of control points in the u-direction

**vtype** (default 3 = cubic) The degree of B-Spline used to fit the surface in the v-direction

**nvlm** The max number of control points in the v-direction

**tol** tolerance (default = 1e-5 x surface size)

**assemble** (0|1) into single NURBS surface (default = 1)

**lt** Surface type: 1 for LtNURBSurf (default =1)

The command (Tools/Command Window) used to fit the above surface in 5 panes is:

**NUBSurfFit 3 32 1 2 .001 0 1**

NUBSurfFit - Fit the selected surface using a degree 3 spline in the u-direction and a max of 32 control points. Fit the v-direction with a degree 1 spline with 2 control points. The tolerance will be 1E-3. The surface will be broken into separate panes, which will be LtNurbs Surfaces.

---

## New Entities:

---

### Copy family

A set of six new entities are on the Insert menu: CopyPoint, CopyCurve, CopySurf, CopyBead, CopyRing and CopyMagnet. Along with CopySnake, which has been on the menu since version 4.0, these all share a common theme: copying some geometry from one environment into another, with some optional transformations in the process.

### CopyPoint, CopyCurve, CopySurf

Each of these entities takes a support of the same class (the basis object); two frames; and three scale factors. The two “environments” are two frames, designated *Frame0* (the “source” frame) and *Frame1* (the “destination” frame). The process of constructing the Copy object has three steps:

- 1 transform the basis object into frame coordinates in *Frame0*;
- 2 apply the three stretching factors to the x, y, z coordinates in *Frame0*;
- 3 create the Copy at those same x, y, z coordinates in *Frame1*.

All this sounds like a lot of Frames. Although Frames are versatile and not hard to create, remember that any point can serve as a frame; its x, y, z axes are parallel to the global X, Y, Z axes respectively. In practice, some 90% of Copy objects will use points for one or both frame supports.

The simplest application of Copy entities is to create a 1:1 copy of an object, parallel to its original orientation, but at another location. This is the best way yet to make a parallel copy of a curve. Suppose:

- 1 the basis curve is ‘c0’,
- 2 there is a point ‘pt0’ at its t = 0 end, and
- 3 there is a point ‘pt1’ where the copy is supposed to start.

Make a CopyCurve from ‘c0’ with *Frame0* = ‘pt0’ and *Frame1* = ‘pt1’, and 1, 1, 1 scale factors. The two frames are parallel in this case, so the transformation is just a parallel displacement equal to the displacement from ‘pt0’ to ‘pt1’.

If the Copy needs to be the same shape, but a different size, you can apply equal scale factors, e.g. 0.5, 0.5, 0.5 for a half-scale curve or surface. If the copy needs to be scaled differently in 2 or 3 directions, use unequal scale factors. To make a mirror image, one or more of the scale factors can be negative -- or the destination frame needs to have the opposite orientation (“handedness”) from the source frame.

If the Copy needs to be in an entirely different location and orientation, then build a frame in the required position, and use it as the *Frame1* support.

There’s no requirement that *Frame1* be different from *Frame0*. Sometimes a Copy into the same frame is useful. Suppose you want a duplicate of a surface in the same place, with no change except different color, different divisions, or a different layer? Then you can use the same point (any point, in fact) for *Frame0* and *Frame1*.

CopyPoint provides the best way yet to make the fourth corner of a rectangle (or parallelogram) when three corners are known. Suppose the three known corners are ‘pt1’, ‘pt2’, ‘pt3’. Then the corner opposite to ‘pt2’ is the CopyPoint made with ‘pt1’ as basis point, ‘pt2’ as source frame, and ‘pt3’ as destination frame. (This is more convenient than a BlendPoint, because you don’t have to mess around with any weights.)

Sample files: COPYCURVE.MS2, RPY.MS2, RPY-JET.MS2.

## CopyBead, CopyRing

The CopyBead and CopyRing are very similar to one another. As with other similarly named pairs of beads and rings, the only difference is that a CopyRing, residing on a snake, also marks a position on the snake's host surface, and so it can serve in any role where a magnet is needed. Typically, if the support is a snake, we would always make a ring rather than a bead.

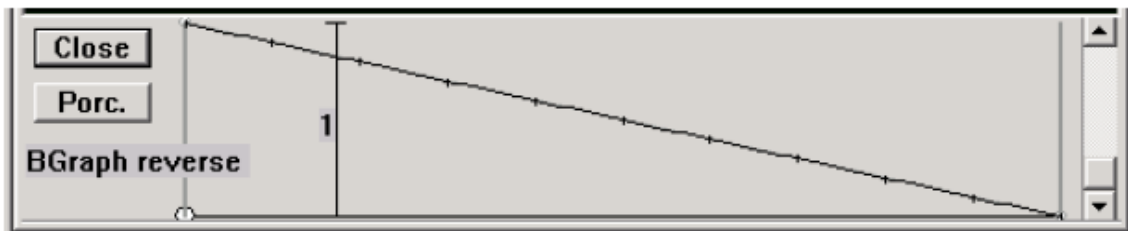
In the case of beads and rings, the two "environments" are the  $t$  parameter spaces of two curves or snakes. The data for a CopyBead is a basis bead (or ring), a destination curve, and a graph.

In its simplest manifestation (with graph = '\*'), a CopyBead is a bead on a curve, 'curve1', which copies the  $t$  position of the basis bead (or ring) on *its* host curve (or snake) 'curve0'. We think of the CopyBead as being "slaved" to its supporting bead, often functioning as a "remote control".

For the simplest example, picture two Lines 'line0', 'line1' (both with default relabel); an AbsBead 'e0' on 'line0', and a CopyBead 'e1' made by copying 'e0' onto 'line1'. As you drag 'e0', 'e1' will continuously update to remain at the same  $t$  location, and therefore the same proportional position along its line, as 'e0'.

The *graph* support greatly enriches the behavior of CopyBead and CopyRing by permitting a transformation of the parameter between the source curve and destination curve. The default graph '\*' in this context signifies no transformation, i.e.,  $h(t) = t$ . A more complex graph produces a more complex motion or response of the CopyBead. Since there are few limits on the complexity of a graph you can specify practically any relationship between the positions of the two beads.

A simple graph example that is frequently useful is the type-1 BGraph with values  $\{1, 0\}$ , i.e.,  $h(t) = 1 - t$ . We usually give this graph the name 'reverse'. If introduced into the above example of two lines, it makes the CopyBead go to the complementary position  $1 - t$ , i.e. it traverses 'line1' from 1 to 0 as the basis bead 'e0' goes from 0 to 1.



CopyBeads and CopyRings are frequently useful in the related topics of animation and procedural entities. A ProcCurve, ProcSnake or ProcCvSurf that uses 2 or more driving beads is difficult to investigate because you have to move all those beads to new positions to visualize the construction taking place at any particular location. If you replace all but one driving bead with CopyBeads, you can easily slide the whole construction along the full length of the driving bead's supporting curve and see

what's going on. If the construction fails at some position as you drag, the moving point or curve will disappear; if you accept the drag at that position, you'll be able to see exactly how the construction fails, by the set of errors that pop up in the Error View.

Animation also benefits from CopyBeads and CopyRings. The sample file PITCHPOLE.MS2 illustrates this. Open the file, and drag Bead "go" around the circular arc.

(Pitchpoling occurs when a vessel is caught by a steep breaking sea that somersaults her, head over heels. Like a collision at sea, a pitchpole can ruin your whole day.)

This model uses about a dozen CopyBeads, one CopyRing, and several CopyCurves and CopySurfs. If you turn on layer 1 you can see how it works. The CopyBeads (bright red) are all slaved to the AbsBead 'go'. They are control points for a BCurve 'constr\_wave' (bright green), which creates the evolving wave profile. If you drag 'go' around the arc, you will see each CopyBead slide correspondingly down its supporting curve. The supporting curves are mostly uniformly labeled lines; a few near the wave crest are curved or have relabels to control the details of the wave profile.

A CopyCurve 'waveside' is used to make a 1:1 copy of 'constr\_wave' down at sea level; the sea surface is TranSurf 'wave' made from 'waveside' and 'wavesweep'.

The motion of the boat, in relation to the wave crest, is controlled by CopyRing 'r0' (also slaved to 'go'), that slides along a LineSnake 'n0' on 'wave'. Frame3 'F0' is located at 'r0' and arranged to have its x and y axes tangent to the 'wave' surface at all times. The boat surfaces are several CopySurfs made from a base boat model positioned at the origin (on layer 2). Note that the boat is white in color; if you are not seeing it, make sure that your background is not white as well.

Sample files: COPYBEAD1.MS2, COPYBEAD2.MS2, COPYRING1.MS2

## CopyMagnet

A CopyMagnet is a copy of a magnet or ring from the u,v-space of a "source" surface to the u,v-space of a "destination" surface. The u,v-spaces are the two "environments". The characteristic data for a CopyMagnet is:

a basis magnet

the destination surface

u-graph

v-graph

In its simplest form, with both graphs equal to the default graph '\*', the CopyMagnet reads the u,v position of the basis magnet, and locates itself at that same u,v position on the destination surface. The graphs allow the u and/or v parameters to be transformed in the process, similar to the effects of the graph of a CopyBead. Thus the CopyMagnet is "slaved" to the position of its basis magnet, with an optional transformation in between.

The CopyMagnet has utility for making ProcPtSurfs, similar to the use of CopyBeads and CopyRings for the other Proc entities. That is, by use of a single AbsMagnet and CopyMagnets slaved to it, you can easily move the construction to various u,v positions, verifying that the construction takes place as expected, or locating and "debugging" regions where the construction fails.

Sample file: Copy Magnet.MS2

---

## Rotated Curve

**Characteristic data** *curve* = basis curve  
*line* = axis of rotation  
*angle* = angle of rotation

**Description** A Rotated Curve is formed by rotating a basis *curve* around an axis *line* through a specified *angle*, creating a new curve object, which is an exact copy of the basis *curve* in a new position.

*curve* can be any curve or snake object. *line* is a Line object.

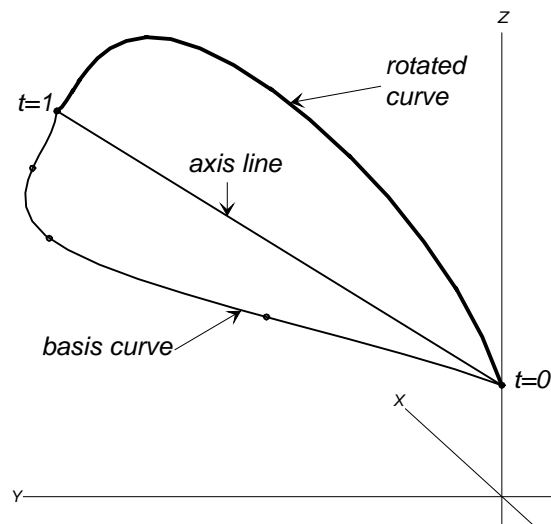
*angle* is specified in degrees. *angle* = 0 puts the RotatCurve right on top of the basis *curve*. To determine the direction of increasing *angle*, use the right hand rule: point your right thumb along the *axis* line, with the tip toward the  $t=1$  end – your fingers curl (point) in the direction of increasing *angle*.

*relabel* is used to relabel the curve. The program's default *relabel* '\*' produces the "natural" labeling, in which the parameter  $t$  is distributed the same as for the basis curve.

### Example

#### Rotated Curve.ms2

'cspline' is the basis curve and 'line\_1' is the axis line for the rotated curve 'rotcurve' which has been rotated 60 degrees from the basis curve's position. Check out the right hand rule – point your right thumb along 'line\_1', with the tip toward  $t=1$  – *angle* increases in the direction your fingers point, that is from the basis curve toward the rotated curve (and onward around the axis line).



*Rotated Curve.MS2 at Lat 0, Lon 0.*

### See also

Rotated Point, Rotated Surf

---

## Rotated Point

**Characteristic data**    *point* = basis point  
                                  *line* = axis of rotation  
                                  *angle* = angle of rotation

**Description**            A Rotated Point is created by rotating a basis *point* around an axis *line* through a specified *angle*, creating a new point object.

*point* can be any point object. *line* is a Line object.

*angle* is specified in degrees. *angle* = 0 puts the RotatPoint right on top of the basis *point*. To determine the direction of increasing *angle*, you can use the right hand rule: point your right thumb along the *axis* line, with the tip toward the t=1 end — your fingers curl (point) in the direction of increasing *angle*.

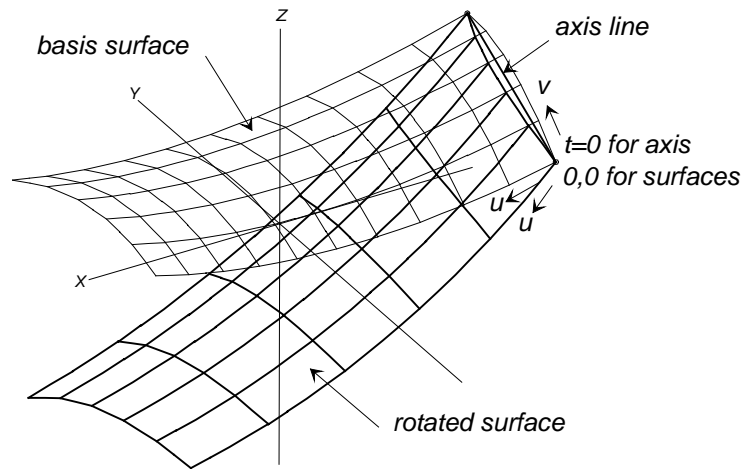
Rotated entities allow you to build angular relationships and rotational symmetries into your model.

**Example**                 **Rotated Point.MS2**

This model is a regular pentagon. 'p1', 'p2', 'p3', and 'p4' are four rotated points all of which use 'p0' for the basis point and 'axis' for the axis line of rotation (in the right hand figure, 'axis' is pointing straight up at you). The angles of rotation are 72, 144, 216, and 288 degrees respectively. Check out the right hand rule — point your right thumb along 'axis', from the origin toward 'up' — *angle* increases, as it should, from 'p0' to 'p1' to 'p2', etc. (not from 'p0' to 'p4' to 'p3', etc.).



and-white figure — the surfaces, though, are identical except for their position in space. Note that the u-divisions for the two surfaces actually are coordinated: basis surface u-divisions = 10x2, RotatSurf u-divisions = 5x4.



Rotated Surface.MS2 at Lat -30, Lon 60.

---

## Intersection Entities

Reference for IntBead, IntMagnet, IntRing, IntRing2, and IntSnake entities.

### Introduction

Intersection and projected entities solve various problems involving intersections between objects. The intersection of a curve and a surface will generally be a point, which can be made either as a bead on the curve, or as a magnet on the surface. The intersection of a snake and a surface (different from the surface the snake belongs to) will also generally be a point, which can be made either as a ring on the snake, or as a magnet on the surface. The intersection of two surfaces is a snake which could lie in either surface.

In general, you cannot expect two 3-dimensional curves or two snakes on different surfaces to intersect unless they have been specially constructed to do so. However, two snakes belonging to the same surface can easily intersect at a point. And you can construct two curves to intersect in a variety of ways — for instance, you can put a bead on the first curve and use that bead as one of the control points of a CCurve second curve.

### Basic definitions

The basic data for intersection entities are the names of two objects that logically can intersect. We think of the first of these objects as *being cut* and the other as *doing the cutting* or being *the cutter*. The object resulting from the intersection is a bead, ring, magnet, or snake *on the object being cut*.

### The object being cut

In general, you have two alternative ways of specifying the curve, snake, or surface which is being cut:

- to specify a *curve*, you can name either the curve itself *or* a bead lying on that curve

- to specify a *snake*, you can name either the snake itself *or* a bead or ring lying on that snake
- to specify a *surface*, you can name either the surface itself *or* a magnet or ring lying on that surface

This flexibility is indicated in the entity specification by slash punctuation, for example: *magnet/surface* or *bead/curve*.

**Why name a bead, ring, or magnet rather than a curve, snake, or surface.** To find an intersection, the program must *search* for it — it sort of casts back and forth (a bit like a hound searching for an elusive scent), refining its search with each step. Whether it gets to the intersection and how accurate its solution is depends on where it starts its search. As you might imagine, it is more likely to be successful if it starts its search close to the intersection.

By default, the program starts searching at the middle of the curve or surface being cut:  $t = .5$  on a curve or snake, or  $u = .5, v = .5$  on a surface. It uses this default when you just name a curve, snake, or surface (as opposed to a bead, ring, or magnet) when designating the object being cut. In many cases this will be accurate enough.

In other cases though, especially when there is more than one possible intersection, you may need to give the program a better starting point for its search. You do this by naming a bead, ring, or magnet when designating the object being cut — the point object performs 3 functions:

- it designates the curve, snake, or surface being cut
- it designates which one of multiple intersections you want
- it helps the program find the intersection by giving it a good starting point to begin its search

Place the point object close to the intersection you want — the closer, the better (within reason — you want the *program* to calculate the intersection, so don't go to the trouble of calculating it yourself!). For example, let's take the case of an IntMagnet intersection between a surface and a curve that crosses the surface twice. If you were to create a magnet and drag it right next to the intersection you want (placing it by eye should be fine), the magnet would both designate the surface being cut and give the program a close starting place to begin its search for the IntMagnet location.

### The cutter

In order to allow cutting flexibility, the actual cutter for most of the intersection entities is designated as a combination of a *mirror/surface* and a *point*. This combination defines the “cutting surface” (note that this cutting surface is not a real SurfaceWorks object; it is only an imaginary construction surface):

*mirror/surface* defines an infinite family of potential cutting surfaces. It can be either a mirror (plane, Line, or point) object or a surface object.

*point* specifies which one of the infinite family of cutting surfaces is the actual cutting surface. *point* specifies the distance from *mirror/surface* to the cutting surface. When *point* lies on *mirror/surface*, the distance is zero. *point* can be any point object.

The following table shows the effect of the various choices for cutters:

**mirror/surface    cutting surface (a construction surface; not a real object)**

plane ( <i>mirror</i> )	the parallel plane which passes through <i>point – point</i> specifies the perpendicular distance from the specified plane to the cutting plane
line ( <i>mirror</i> )	the circular cylinder which has its axis along this line and which passes through <i>point – point</i> specifies the radius of the cylinder
point ( <i>mirror</i> )	the sphere which is centered at the mirror and passes through <i>point – point</i> specifies the radius of the sphere
surface ( <i>surface</i> )	the parallel surface which passes through <i>point – point</i> specifies the perpendicular distance from the specified surface to the cutting surface

When *mirror/surface* is a surface, usually the cutting surface you want is the surface itself, not one of its parallel surfaces. In this case, you can use any magnet on the surface for *point*; this assures a zero offset.

(A parallel surface is a surface that is spaced a uniform perpendicular distance away from the surface it is based on. An OffsetSurf with uniform offset is a parallel surface.)

### Intersections

In all cases of calculation of intersections, there is the possibility that the desired intersection is in fact nonexistent. Just because two surfaces *might* intersect doesn't mean they actually *do* intersect. Just because a curve *can* pass through a surface doesn't mean it actually *will*. Don't worry – nonexistent intersections are trapped as geometry errors.

At the same time, there is often the possibility that two objects will intersect in two or more places. SurfaceWorks handles this by giving you the option to “point” to the correct intersection with a bead, ring, or magnet on the object being cut.

There is another awkward possibility — that the intersection actually has the wrong dimensionality. For example, take the IntMagnet, which being a magnet, is in the class of point objects. Normally, a surface would be cut by a curve at one or more isolated points. However, it is possible that part or all of the cutting curve actually lies in the surface, so the intersection is really a curve. Similarly, the intersection of two surfaces would usually be a curve; but it could be a surface, if the two happen to coincide over some area. In general, these higher-dimensional, coincidence type intersections will produce non-useful results or geometry errors.

### Usage

Intersection entities provide important capabilities in SurfaceWorks. However, we present them with a cautionary note. Because they involve iterative solutions, they are slower to evaluate, less reliable, and somewhat less precise than other entities of the same class. They involve more error possibilities. Often, other entities provide a better – simpler, faster, more direct, or more reliable – way to accomplish a design task. A prime example of this dichotomy is provided by the conventional way of handling surface-surface joins and intersections in 3D CAD:

- (1) construct the two surfaces so they deliberately intersect
- (2) solve for the curve of intersection
- (3) use the intersection curve to trim off the unwanted portions of the original surfaces

You *can* do this same thing in SurfaceWorks, but it doesn't mean you *should*! Very often, surfaces that join accurately and durably can better be made by joining two surfaces to a common edge curve, or by starting a surface from a snake on another surface: you design the intersection first, then make surfaces that join correctly onto it (for an example, look at CYLTRAN.MS2.

See IntBead, IntMagnet, IntRing, IntRing2, IntSnake.

## Intersection Bead

**Characteristic data** *bead/curve* = curve to be intersected (object being cut)  
*mirror/surface* = basis cutting plane or surface  
*point* = specifies the actual cutting plane or surface

**Description** An Intersection Bead is a point on a curve, located where a cutting surface intersects the curve.

**The object being cut** is designated by *bead/curve* (either a bead or a curve):

When *bead/curve* is a bead, it serves two purposes:

- (1) It specifies which curve the IntBead is to lie on: the curve that already supports *bead*.
- (2) In case of multiple intersections of the curve with the cutting surface, the IntBead will be located at the intersection nearest to *bead*.

When *bead/curve* is a curve, it specifies directly which curve the IntBead is to lie on. In case of multiple intersections, the one closest to  $t = 0.5$  will be taken.

**The cutting object** is designated as a combination of a *mirror/surface* and a *point*:

*mirror/surface* defines an infinite family of potential cutting surfaces. It can be either a mirror (plane, Line, or point) object or a surface object.

*point* specifies which one of the infinite family of cutting surfaces is the actual cutting surface. *point* specifies the distance from *mirror/surface* to the cutting surface. When *point* lies on *mirror/surface*, the distance is zero. *point* can be any point object.

The following table shows the effect of the various choices for cutters:

<b><i>mirror/surface</i></b>	<b>cutting surface (a construction surface; not a real object)</b>
plane ( <i>mirror</i> )	the parallel plane which passes through <i>point</i> – <i>point</i> specifies the perpendicular distance from the specified plane to the cutting plane
line ( <i>mirror</i> )	the circular cylinder which has its axis along this line and which passes through <i>point</i> – <i>point</i> specifies the radius of the cylinder
point ( <i>mirror</i> )	the sphere which is centered at the mirror and passes through <i>point</i> – <i>point</i> specifies the radius of the sphere

surface (*surface*) the parallel surface which passes through *point* — *point* specifies the perpendicular distance from the specified surface to the cutting surface

When *mirror/surface* is a surface, usually the cutting surface you want is the surface itself, not one of its parallel surfaces. In this case, you can use any magnet on the surface for *point*; this assures a zero offset.

## Example

### Piston.ms2

This model, the skeleton of a reciprocating engine, shows an example of the use of an IntBead to control the length of a moving part in a mechanism.

Bead 'e1' (green) is at the intersection of the crankshaft axis and the cylinder axis. RelPoint 'tdc' (magenta) is in a fixed position above 'e1', establishing the eccentric radius of the crank (0.35).

Bead 'e5' is the center of the crankpin. RelPoint 'p5' is a fixed distance (1.25) above Bead 'e5'; this distance establishes the length of the connecting rod (which runs from Bead 'e5' to IntBead 'e2').

IntBead 'e2' (blue) lies on the cylinder axis and uses both 'p3' and 'p5':

- the object being cut is the cylinder axis
- the cutting object is designated by the combination of:
  - mirror* = Bead 'e5' — since this is a point object, the infinite family of potential cutting surfaces are spheres, centered at 'e5'
  - point* = RelPoint 'p5' — it designates the actual cutting sphere, out of the infinite family of potentials; in this case, since RelPoint 'p5' is 1.25 units from 'p3', the cutting sphere is the one with radius = 1.25 (this is the length of the connecting rod)

You can vary the *t-location* of Bead 'e5' to crank the engine

## Intersection Magnet

**Characteristic data** *magnet/surface* = surface to be intersected (object being cut)  
*bead/curve* = cutting curve

**Description** An Intersection Magnet is a point on a surface, located where a curve (extended if necessary) intersects the surface.

**The object being cut** is designated by *magnet/surface* (a magnet, ring, or surface):

When *magnet/surface* is a magnet, it serves two purposes:

- (1) It specifies which surface the IntMagnet lies on: the surface that supports *magnet*.

- (2) In case the line intersects the surface more than once, the IntMagnet will be located at the intersection nearest to *magnet*.

When *magnet/surface* is a surface, it specifies directly which surface the IntMagnet is to lie on. In case of multiple intersections, the one closest to  $u = 0.5, v = 0.5$  will be taken.

**The cutting object** is designated by *bead/curve* (either a bead or a curve):

When *bead/curve* is a bead, the cutting object is the curve the bead belongs to.

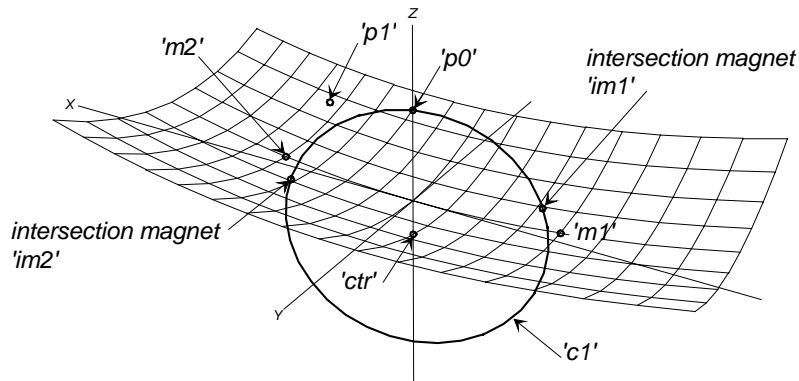
When *bead/curve* is a curve, it specifies the cutting object directly.

## Example

### Intersection Magnet.MS2

This example uses IntMagnets to locate the two intersections of a surface and a circle.

The TranSurf 'patch' (cyan) and the circle 'c1' (green) intersect at two points. AbsMagnets 'm1' and 'm2' (magenta), located in the vicinity of the two intersections, are used to give the program good starting places from which to search for the intersections for each of the IntMagnets 'im1' and 'im2' (red).



Intersection Magnet.MS2 at Lat 30, Lon 60.

See also IntBead, IntRing, IntSnake

---

## Offset Curve

**Characteristic data** *snake* = basis curve  
*offset1* and *offset2* = offset distances at ends of *snake*  
*graph* = name of a BGraph object or '\*' for default graph

**Description** An Offset Curve is constructed from a snake by taking each point on the snake and moving it an offset distance along the normal to the surface at that point. *offset1* and *offset2* are signed decimal numbers which specify the offset distance at the  $t=0$  and  $t=1$  ends of *snake*, respectively.

The *graph* supplies a "blending function" which controls how the end offset values *offset1*, *offset2* are distributed along snake to find points on

the OffsetCurv. A point on the OffsetCurv at parameter value  $t$  is located by these steps:

- (1)  $t$  is passed through the specified relabel function to get the natural parameter  $t'$ .
- (2) *graph* is evaluated at parameter  $t'$ , resulting in a value  $B$  of the blending function.
- (3) *snake* is evaluated at  $t'$ .
- (4) The resulting point is displaced along the normal to the surface by a weighted combination of the end offsets:  $(1-B) \text{offset1} + B \text{offset2}$ .

The default *graph* '\*' in this context is a straight line from 0,0 to 1,1 (just like the default *relabel* '\*'); this produces a linear blending of the end offsets.

In many cases (including the default) the *graph* will run from 0,0 to 1,1, i.e. the first and last *graph* values will be 0. and 1. Whenever this is the case, the offset distance starts at *offset1* and ends at *offset2*. However, this is not required.

Should you want to directly specify the distribution of offset distance by the shape of the *graph*, here's a convenient way:

- (1) Set *offset1* to 0.
- (2) Set *offset2* to the final offset value.
- (3) Shape the *graph* so it expresses the desired distribution of offset distance and ends with 1.

For a constant offset, specify the same number for *offset1* and *offset2*, and use the default *graph*.

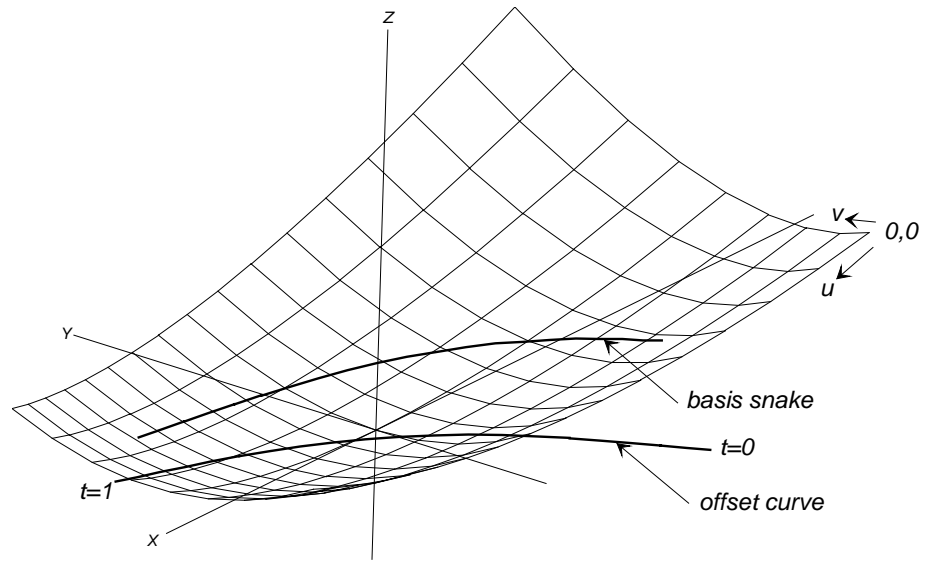
The natural labeling of an Offset Curve is identical to that of its basis *snake*.

Note: If you change *orientation* for the surface, the OffsetCurv will move to the other side of the surface.

## Example 1

### Offset Curve.ms2

The normal orientation switch (*orientation*) for the surface 'patch' is 0 (zero), which means the positive normal points up as you look at the model from this Lat -30, Lon 40 viewpoint. 'basis\_snake' (magenta) lies in 'patch, and 'offset\_curve' (blue) is offset in the *negative* normal direction from the snake, 1 unit from its  $t=0$  end and .5 unit from its  $t=1$  end.

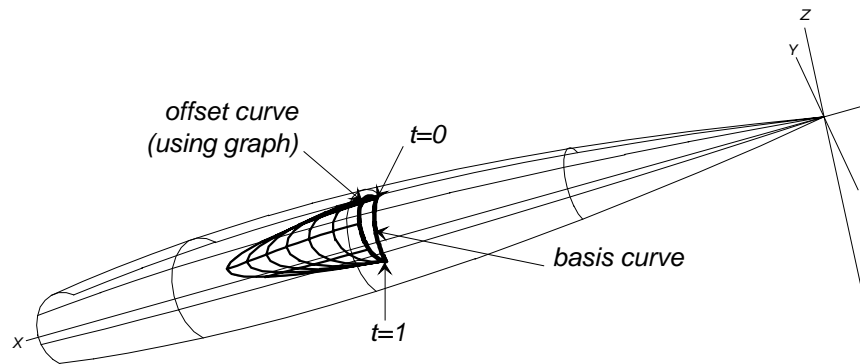


**OFFSET CURVE.MS2 at Lat -30, Lon 40.**

**Example 2**

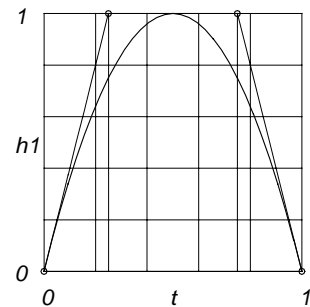
**Inlet.ms2**

In this example, an OffsetCurv with a graph is used to make the opening of an air inlet for the JET model.



**INLET.MS2 at Lat -30, Lon 60.**

'n1' (magenta) is a LineSnake on 'fuselage'. OffsetCurv 'lip' (cyan) is made from 'n1' with 0 (zero) offset specified at  $t = 0$  and 1.00 ft offset at  $t = 1$ . BGraph 'h1' is type-2 with 4 values: 0, 1, 1, 0. This forms a parabola (see graph at right) which specifies the distribution of offset: zero at both ends; maximum of 1.0 in the center.



The inlet is a Blister Surface using 'lip' for its apex (= curve).

**See also**

OffsetPt, OffsetSurf

---

# New Procedural Entities

## Procedural Curve

<b>Characteristic data</b>	<i>point</i> = the moving point whose path is the ProcCurve <i>bead/ring/graph1 ... bead/ring/graphN</i> = support(s) for <i>point</i> ; also define the ProcCurve modeling process
<b>Description</b>	<p>A Procedural Curve is defined by creating one <i>point</i> on the ProcCurve, then having SurfaceWorks repeat the <i>point</i> creation procedure for all positions of the supporting <i>bead/ring(s)</i> along their host curve(s) or snake(s).</p> <p><i>point</i> is any kind of point object constructed directly or indirectly from the 1 or more <i>bead/rings</i>.</p> <p>The <i>bead/rings</i> have to be AbsBeads or AbsRings.</p> <p>To generate the ProcCurve, the program varies the parameters of <i>bead/ring1</i> from t=0 to t=1, varying the other <i>bead/rings</i> in unison, and the ProcCurve is the resulting path of <i>point</i>. If the <i>bead/rings</i> don't all have the same t, the program preserves the t offsets between them. Any <i>graphs</i> included as supports are evaluated at each parameter value t and substituted, in order, for any decimal data values in <i>point</i>.</p>

### Example 1

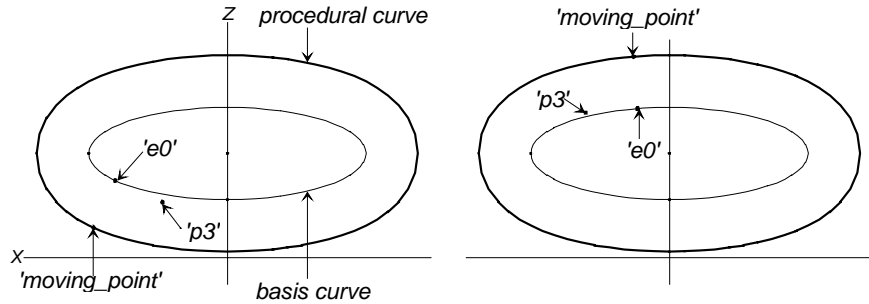
#### Procedural Curve.MS2

'rim' (white) is the ProcCurve in this example. It is made by repeating the procedure of making 'moving\_point' (yellow) for every position of 'e0' (green) along the basis curve 'c0' (cyan; an ellipse). In this case, *moving point* ('moving\_point') is only indirectly based on *bead* ('e0'). Here are the details:

'e0' (green) is an AbsBead that lies on the type-1 Conic 'c0' (an ellipse). 'p3' (red) is a TanPoint offset from 'e0' by 1.5 units. 'moving\_point' (yellow) is a RotatPoint based on 'p3' and rotated 90 degrees around the Line 'axis' (use the <x> view to see 'axis').

The program generates the ProcCurve by varying the parameter t for 'e0' from 0. to 1. — the resulting path of 'moving\_point' is the ProcCurve, a curve offset exactly 1.5 units from 'e0' in the plane perpendicular to 'axis'. 'rim' is an example of a parallel curve of 'c0'.

You can use Edit/ Show/ All to display the RuledSurf that is built between 'c0' and 'rim'.



**Procedural Curve.MS2 in <y> view: initial model (left); model with bead 'e0' moved, showing 'moving\_point' at another position on the path it sweeps out to form the ProcCurve (right).**

### Example 2

These mini-examples illustrate how ProcCurves could be used as alternative constructions for other kinds of SurfaceWorks entities:

- (1) If *point* is an OffsetPt made from a ring, the ProcCurve would be the same as an OffsetCurv with constant offset. (A graph could be used to vary the offset.)
- (2) If *point* is a RelPoint made from a bead or ring, the ProcCurve would be the same as a RelCurve with constant offset — just a translated copy of the basis curve.
- (3) If *point* is a MirrPoint, based on a bead, the ProcCurve would be the same as a MirrCurve.
- (4) If *point* is an AbsBead at  $t=.5$  on a line joining 'bead1' on 'curve1' and 'bead2' on 'curve2', the ProcCurve would be a curve lying midway between 'curve1' and 'curve2'. This curve would be the same as making the RuledSurf between 'curve1' and 'curve2' and putting a UVSnake on it at  $v=.5$ .

### See also

ProcSnake, ProcPtSurf, ProcCvSurf

See also: ProcSnake ProcPtSurf ProcCvSurf

## Procedural Snake

### Characteristic data

*magnet* = the moving point whose path is the ProcSnake  
*bead/ring/graph1 ... bead/ring/graphN* = support(s) for *magnet*; also define the ProcSnake modeling process

### Description

A Procedural Snake is defined by creating one point (*magnet*) on the ProcSnake, then having SurfaceWorks repeat the *magnet* creation procedure for all positions of the supporting *bead/ring(s)* along their host curve(s) or snake(s).

*magnet* is a magnet or ring constructed directly or indirectly from the 1 or more *bead/rings*.

The *bead/rings* have to be AbsBeads or AbsRings. (not relative to another point)

To generate the ProcSnake, the program varies the parameters of *bead/ring1* from  $t=0$  to  $t=1$ , varying the other *bead/rings* in unison, and the

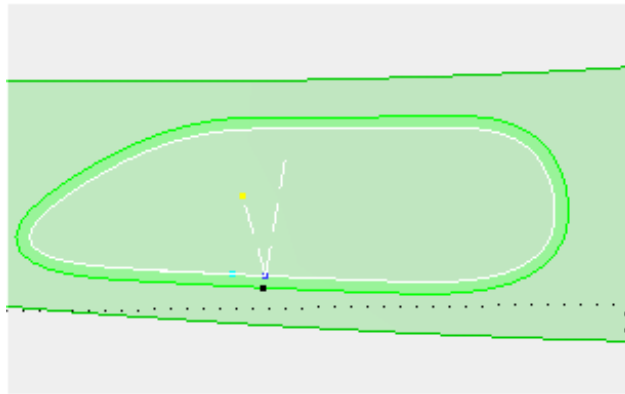
ProcSnake is the resulting path of *magnet*. If the *bead/rings* don't all have the same *t*, the program preserves the *t* offsets between them. Any *graphs* included as supports are evaluated at each parameter value *t* and substituted, in order, for any decimal data values in *point*.

For more information on specifying supports and using *graphs*, see "Collective Entity Information"

### Example 1

#### ProceduralSnake.ms2

Suppose we have made a closed snake 'portlight' on a hull or cabin side to represent the outline of a portlight, and we want to put a constant-width frame around it.



*Procedural Snake.MS2 in <y> view.*

'frame' (green) is the ProcSnake in this example. It is made by repeating the procedure of making 'moving\_magnet' (red) for every position of 'r0' (blue) along the basis curve 'portlight' (white; a NURBSnake). In this case, *moving magnet* ('moving\_magnet') is only indirectly based on *ring* ('r0'). Here is the procedure by which 'moving\_magnet' is created:

'r0' (blue) is an AbsRing that lies on the NURBSnake 'portlight'. 'tanpt' (cyan) is a TanPoint offset from 'r0' by 0.1 units. 'Pt1' is a Point in the frame created from the 'moving\_magnet', the 'tanpt' and the yellow Offset Point. 'moving\_magnet' (red; a ProjMagnet) is the projection of 'Pt1' onto the cabin side.

The program generates the ProcSnake by varying the parameter *t* for 'r0' from 0. to 1. — the resulting path of 'moving\_magnet' is the ProcSnake, a snake offset exactly 0.1 units from 'portlight'. You can use Edit/ Show/ All to display the RuledSurf that is built between 'c0' and 'rim'.

### Example 2

These mini-examples illustrate how ProcSnakes could be used as alternative constructions for other kinds of SurfaceWorks entities:

- (1) If *magnet* is a ProjMagnet made from a bead, the ProcSnake would be the same as a ProjSnake.
- (2) If *magnet* is an RelMagnet made from a bead, the ProcSnake would be the same as a RelSnake with constant offset.

---

## RPYFrame

### Description

“RPY” stands for “roll, pitch, yaw”, the three standard angles for describing the instantaneous attitude of a ship, and by extension, other vehicles.

### Characteristic data

The characteristic data for an RPYFrame is similar, with one exception, to that for an EulerFrame: it includes a *point* where the RPYFrame is located; a parent *frame* (default = ‘\*’, the global coordinate system), and 3 angles. However, the three angles are about different axes and are applied in a different sequence, making the RPYFrame easier to apply in many situations.

An RPYFrame has one additional piece of data: an *orientation* attribute, which is either 0 or 1. This attribute determines the RPYFrame’s *handedness* according to the following rules: in a right-handed frame, if you point the index finger of your right hand along the x axis and your middle finger along the y axis, then your thumb points in the positive z direction; in a left-handed frame, substitute your left hand.

Note: To date in SurfaceWorks, frames have always been right-handed coordinate systems. Enough applications have shown up for left-handed frames that we decided to introduce this option with the RPYFrame. If you wish to work exclusively with right-handed frames, just leave the *orientation* attribute at 0 for all your frames.

Construction of an RPYFrame starts by making a parallel copy of the parent *frame* at the location of *point*. Then the three rotations are applied as follows, *in this order*:

- 1 Roll -- rotation about the x axis of the RPYFrame
- 2 Pitch -- rotation about the y axis of the RPYFrame
- 3 Yaw -- rotation about the z axis of the RPYFrame.

The positive direction of any of these rotations is indicated by a right-hand or lefthand rule, depending on the handedness of the parent *frame*: if you grip any one of the axes with the appropriate hand, thumb along the axis in the positive direction, your fingers are curling in the direction of positive rotation.

Finally, if *orientation* is 1, the z axis is reversed. Thus the RPYFrame will have:

opposite “handedness” from its *frame* support when *orientation* is 1,  
the same “handedness” as its *frame* support when *orientation* is 0.

Frames made from the Frame3 entity are always right-handed. (Frame3 do not have the *orientation* parameter which could logically allow their handedness to be reversed.)

Sample files: RPY.MS2, RPY-JET.MS2

---

## BGraph — B-spline Graph

**Characteristic data** *value1 ... valueN* = values for graph control points  
*type* = type of B-spline used in graph: 1 = linear, 2 = quadratic, 3 = cubic, etc.

**Description** A BGraph is a graph of some function *f* vs. *t* from 0 to 1, represented as a sum of B-splines (with uniform knots):

$$f(t) = \sum_{j=1}^N f_j B_j(t) \quad (1)$$

*type* is a positive integer specifying the B-spline type used in the BGraph formulas: 1 = linear, 2 = quadratic, 3 = cubic, etc. The higher the type, the stiffer the B-spline (for more information on B-splines, see “Collective Entity Information”).

The braces enclose a list of values. As you may have noticed, the data format is the same as for a Relabel object, except the first and last components of the list are not forced to 0 and 1 — for instance, { -.20 -.02 0 } would be a perfectly legitimate values list for a BGraph.

Like KnotList and Relabel, a BGraph has no color or visibility, and it is not directly visible in the 3D modeling space, although its effects certainly may be visible.

Currently, BGraphs are used as blending or distributing functions in a number of entities.

**Example 1** **TBS\_Graph.MS2**

This example uses BGraphs to control the shape of a Tangent Boundary Surface.

**Example 2** **MastTaper.ms2**

This example shows a graph being used to taper the mast to 65% of the maximum fore/aft thickness. Another reason to use a graph in this example is to hold the maximum thickness until approx 3/4 of the length and then taper from there.

**See also** CenterPoint Boundary Surface, Tangent Boundary Surface, Offset Curve, Relative Curve, SweepSurf

---

# Variable Support

SurfaceWorks has a profound new capability for generation of parametric families of geometric models, in a new class of *real-valued objects*: the Variable entity type.

## Constants and variables

There are two ways to specify a real number in the syntactic definition of an object: with a constant value, or a variable. Variables are objects and carry a value that lies within a specified range.

### Constants

Constants are 1.23, -0.7, 12e-8, +98, etc. Although they have no units they are always considered to carry the adequate unit needed at the particular location where they are used.

For instance in

```
AbsPoint P 1 1 / 1 2 3 ;
```

1, 2 and 3 are considered as having length units.

In

```
AbsMagnet M 1 1 / S 0.123 0.456 ;
```

the u and v parameter values 0.123 and 0.456 are considered to be unitless.

## Variables

Variables carry a value; they can optionally have units (otherwise they are unitless), and take their value only within a certain range.

### Basic syntax

The simplest syntax for a variable is:

```
Variable name / value ;
```

For instance

```
Variable x / 177.69 ;
```

### Range

If the variable is only allowed within a certain range then the boundaries of the range may be specified in order:

```
Variable name / value (min,max) ;
```

If either *min* or *max* are not specified then it is meant that the lower or higher bound is not specified (or, equivalently, is minus or plus infinity, respectively).

```
Variable x / 177.69 (0,300) ;
```

```
Variable x / 177.69 (,300) ;
```

```
Variable x / 177.69 (0,) ;
```

```
Variable x / 177.69 (,) ;
```

The last case is equivalent to the case where the range is not specified. You can create a one-value variable in the following way:

```
Variable x / 177.69 (177.69, 177.69) ;
```

### Units

Variables may carry units, which are not specified by their names, but by their dimensions with respects to the fundamental units of the SI/MKSA system:

```

L    length
M    mass
T    time
I    intensity of electric current

```

For instance, velocity has unit  $LT^{-1}$ , acceleration,  $LT^{-2}$ , energy  $ML^2T^{-2}$ , etc.

The units of a variable can be specified with any combination of  $L$ ,  $M$ ,  $T$  and  $I$ , with the exponent introduced with '^':

```

Variable height / 100 L^1 ;
Variable mass   / 80 M^1 ;
Variable speed  / 90 L^1 T^-1 (0,300) ;

```

An exponent of 0 is equivalent to no unit of this type; if the exponent is one, it can be omitted; repeating the same unit amounts to multiplying it:

```

Variable x      / 0 M^0 ;
Variable height / 100 L ;
Variable area   / 100 L L ;

```

### Complete syntax

Variables can also have user attributes and be read-only (locked). The complete syntax for a variable is:

```
'Variable' name { attr, } { 'R:1' } '/' value { units } {range} ;'
```

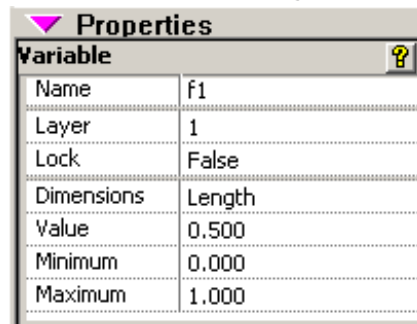
For instance:

```
Variable length A:Usage("profile") / 10 L (0,152) ;
```

### User interface

Figure 3 shows the Property Manager interface to create and Edit a Variable Object.

Fig. 1



## Variables as supports

Variables can be supports of all objects depending on a real value, and wherever an object regardless of its type can be a support, such as in an object list. Therefore they can be supports of points (as coordinates, offsets, angles, t-, u- and v-parameter), curves and surfaces (via knot lists and weights), etc.

```
Variable A / 10 L ;
Variable B / 5 L ;
Variable C / 30 L ;
Variable D / 25 L ;
AbsPoint P1 14 1 / 0 0 0 ;
AbsPoint P2 14 1 / A 0 0 ;
AbsPoint P3 14 1 / A B B ;
AbsPoint P4 14 1 / C B B ;
AbsPoint P5 14 1 / C 0 0 ;
AbsPoint P6 14 1 / D 0 0 ;
Variable k0 / 0.25 ;
Variable k1 / 0.75 ;
KnotList k1 / { 0 k0 0.5 k1 1.0 } ;
Variable w0 / 1 ;
Variable w1 / 3 ;
NURBCurve spline0 11 5 64x1 / * 2 k1 { P1 1 P2 w0 P3 w1 P4 1 P5 1 P6 w0 }
;
```

Note that since a variable is defined by three real numbers, these can be replaced by variables or formulas: you can for instance have a variable with a variable range:

```
Variable min / 0 ;
Variable max / 24 ;
Variable x / 7 (min,max) ;
```

In this case the objects used as value and range bounds must all carry the same units (as always, constant are assigned the units that match).

## User interface

When you are editing a real value in the data for any object, you can use an appropriately dimensioned real object (Variable or Formula) instead of a constant. For example, creating or editing a RadiusArc, one required data element is the Radius. When you select this item a constant or a real object can be entered. Figure 2 shows a Radius Arc with the Radius field active and a radius of 1. To enter a variable, go to the Surfer View and choose the appropriate Variable. (Figure 3) Figure 4 shows Variable 'f1' as a support, which has a value of 0.500.

Fig. 2


Properties	
Radius Arc	
Name	Arc1
Color	
Visible	True
Layer	0
Lock	False
Relabel	*
Type	Tangent to lines Point1-
Radius	1,000
Point1	Bead9
Point2	q6
Point3	Bead4
Divisions	8
Subdivisions	4
Show tickmarks	False
Show polyline	False
Weight/unit length	0.000
User data	

Fig. 3


















Parents	Children
+	 Surfaces
+	 Curves
+	 Points
+	 Planes
+	 Frames
+	 Triangular Meshes
	 Wireframes
	 Contours
	 Composite Surfaces
	 Relabels
+	 Graphs
	 Knotlists
-	 Variables
	 Entity Lists
+	 System
+	 No Dependents

Fig. 4

Properties	
Radius Arc	
Name	Arc1
Color	
Visible	True
Layer	0
Lock	False
Relabel	*
Type	Tangent to lines Point1-
Radius	f1 (0.500)
Point1	Bead9
Point2	q6
Point3	Bead4
Divisions	8
Subdivisions	4
Show tickmarks	False
Show polyline	False
Weight/unit length	0.000
User data	

## Interaction with object transformations

Object transformations are rotations, translations and scaling.

### Scaling of variables

Variables having a non-null length dimension will be scaled. For instance, in case of uniform scaling of 10

```
Variable radius          / 100 L ;  
Variable area            / 3 L^2 ;  
Variable curvature       / 1 L^-1 ;  
Variable kineticEnergy   / 4.2 M L^2 T^-2 ;
```

will become

```
Variable radius          / 1000 L ;  
Variable area            / 300 L^2 ;  
Variable curvature       / 0.1 L^-1 ;  
Variable kineticEnergy   / 420 M L^2 T^-2 ;
```

The lower and upper bound of the range get scaled only if they are not  $+\infty$  or  $-\infty$ .

```
Variable area            / 3 L^2 (1,5) ;  
Variable area            / 3 L^2 (1, ) ;  
Variable area            / 3 L^2 ( ,5) ;
```

becomes

```
Variable area            / 300 L^2 (100,500) ;  
Variable area            / 300 L^2 (100, ) ;  
Variable area            / 300 L^2 ( ,500) ;
```

### Scaling of objects depending on variables

Let

```
Variable d / 9 L ;  
AbsPoint P 14 1 / 3 d 5 ;
```

If the model gets scaled by, say, 10, only the constant parameters of P will be scaled, hence P will become:

```
AbsPoint P 14 1 / 30 d 50 ;
```

Since  $x$  will also be dilated, the coordinates of P will be 30, 90, 50 and everything will be fine. However, in case of non-uniform scaling, it is all wrong: variables are always dilated by the average scale factor  $(\alpha_x \alpha_y \alpha_z)^{1/3}$ , therefore P will end up dilated by  $\alpha_x$ ,  $(\alpha_x \alpha_y \alpha_z)^{1/3}$ ,  $\alpha_z$  instead of  $\alpha_x$ ,  $\alpha_y$ ,  $\alpha_z$ .

Physically realistic formulas (i.e. without additive unitless constants) will be automatically properly scaled, when their geometric supports get scaled. Unrealistic formula will be defeated since they will contain unscaled constants

Variable  $d$  / 9 L ;

Formula  $unscalable / \{ d + 1 \} L$  ;

Scale by 10 and  $d$  becomes 90,  $unscalable$  will be 91 instead of 100.

### ***Rotation and translation of objects depending on variables***

Variables do not get translated or rotated, therefore

AbsPoint P 14 1 / 3 d 5 ;

is untransformable, since its y coordinate cannot change. In conclusion:

*Variable used as world coordinates will defeat non-uniform scaling, translation and rotation..*

### ***How to create models well behaved with respects to transformations?***

- Use physically realistic formulas.
- Systematically create points relative to a frame, which will ensure proper rotation and translation.
- Use variables with caution if you plan to scale non-uniformly.