Chapter 8.
Creating and Modifying Models

Topic: *Examples

This chapter provides examples to familiarize you with using ACIS to create and modify meaningful models. The examples are written in Scheme, and intended for use in the Scheme AIDE application.

For many C++ programmers, the prospect of learning and using Scheme may at first seem counter-productive. However, as you embark into the realm of 3D solid modeling with ACIS as your engine, the Scheme AIDE application can prove to be an invaluable tool. First, modeling concepts—whether very basic or somewhat advanced—can be tested without having to code and compile C++ applications. Secondly, the source code for all Scheme extensions is provided. Therefore, if you can do it in Scheme, you already have a working prototype of how to accomplish the same task in C++.

The examples in this chapter produce results that are readily visible because the Scheme AIDE application has a graphical interface that provides viewport and rendering control.

Flow of Examples

This chapter begins by creating simple line (edge) entities that can be viewed, saved, and restored. Edges are then connected into a wire body; then, the wire body is offset, etc. In general, each example builds on the previous example, and the chapter is written assuming you'll go from one example to the next.

The general flow of the examples in this chapter is:

- Creating edges
- Offsetting wire bodies
- Creating surfaces
- Creating solids
- Modifying models
- Analyzing models (geometric analysis and physical properties)

SAT Files

ACIS models can be saved to a disk file known as a *part save file*, for later retrieval. They can be saved in either text mode (file extension .sat) or binary mode (file extension .sab). The part save file is also known as a *SAT file* (whether text or binary mode). Refer to the *Kernel Component Manual* for information about save files.

The Scheme examples in this chapter save their results to SAT files in text mode (.sat). The SAT files for the examples are named tmp<*name*>.sat, where <*name*> is the description of the example abbreviated.

Running the Examples

Topic: Ignore

In order to run the examples in this chapter, you must first establish the appropriate environment for running Scheme AIDE. Because Scheme AIDE is an end-user application, you only need to start the application, set up for display, and then enter the commands from a Scheme example at the command prompt. Refer to Chapter 6, *Using Scheme AIDE*, for more information.

Starting Scheme AIDE

To start Scheme AIDE, simply run the application's executable file. The exact name of the file depends on your platform and the components you have installed. Refer to Chapter 6, *Using Scheme AIDE*, for instructions on starting the executable on your platform. If you have not already familiarized yourself with using Scheme AIDE, you should review the information in Chapter 6 to help get you started.

Creating a Viewport

In order to display the entities you create in Scheme AIDE, you need to define a viewport (view window). A display list viewport is created using the view:dl command. If you are running Scheme AIDE in Windows on a PC, the view:gl command can be used to create an OpenGL viewport that renders surfaces and solid objects. All Scheme examples in this chapter assume that you already have a viewport defined. Your acisinit.scm file could be updated such that it always creates a viewport. Display list viewports and other types are viewports are covered in more detail in the *Graphic Interaction Component Manual*.

In the following example, a display list viewport is created and then a simple linear edge is created that starts at the origin and ends at the *xyz* location (30, 30, 0). In this example, if a viewport had already been created when Scheme AIDE was started, the view:dl command would simply create another display list viewport of the same model. The order of creation of the model versus the viewport is not important.

Scheme Example

```
; Create a view for display.
(define my_view (view:dl))
;; my_view
; Create a linear edge.
(define my_linear (edge:linear (position 0 0 0)
    (position 25 0 0)))
;; my_linear
```

Clearing the Part

After you've run one example and are ready to start the next during a single Scheme AIDE session, you might want to clear out the entities from the previous example. The part:clear Scheme extension deletes the entities in the part. It is entered as:

```
(part:clear)
```

Creating and Saving Edges

Topic: *Creating Entities

The first examples create some simple line (edge) entities that can be viewed, saved, and restored. Edges are the simplest topologic elements that are visible. Even though it is possible to create lower-level geometric elements, such as curves, these must be attached to some topologic element in order to be displayed.

The following example builds on the view creation example by adding more edges and saving these to a SAT file, called tmpedge.sat. It defines Scheme variables for the positions in order to allow the same positions to be used in multiple operations without re-entry of the data.

Scheme Example

```
; Create a view for display.
(define my_view (view:dl))
;; my_view
; Define the positions to be used.
(define my_point1 (position 0 0 0))
; my_point1
(define my_point2 (position 25 0 0))
; my_point2

; Create edge 1.
(define my_linear (edge:linear my_point1 my_point2))
;; my_linear
; Create edge 2.
(define my_semi_circ (edge:circular my_point1 25 0 270))
;; my_semi_circ

; Create a law for one of the edges.
(define my_law (law "vec( 5*sin(x), x, 0)"))
;; my_law
; Create a sine edge.
(define my_sine (edge:law my_law 0 10))
;; my_sine
; Save the results to an output file.
(part:save "tmpedge1.sat")
;; #t
```

As example shows, a model can be saved in Scheme AIDE using the part:save Scheme extension. Similarly, a model can be loaded using part:load.

Some of the Scheme extensions available for creating various edge types are:

edge:bezier	Creates a cubic bezier curved edge from four control points.
edge:circular	Creates an arc with the specified center position and radius.
edge:circular–3curve	Creates an edge specified by an edge:circular tangent to three curves.
edge:circular–3pt	Creates an arc passing through three positions.
edge:circular–center–rim	Creates an arc given a center position and one or two positions on the arc.
edge:circular–diameter	Creates an arc passing through two positions based on the diameter.
edge:conic	Creates a rho conic edge in which the geometrical definition represents a hyperbola or parabola.
edge:elliptical	Creates a full or partial ellipse by specifying the start and end angles.
edge:law	Creates an edge whose characteristics are determined by a given law.
edge:linear	Creates a linear edge between two locations.

Offsetting a Wire Body

Topic: *Creating Entities, *Offsetting

A body is the highest level topological entity. Solid bodies contain lumps, shells, subshells, faces, loops, coedges, edges, and vertices. Wire bodies contain wires, coedges, edges, and vertices. A simple list of edges can be used to create topology, and ACIS creates the other associated topological entities. The result is a model whose elements know about each other and the topology can be traced.

Many operations, such as wire offsetting, utilize the features of a body topological element and the inherent ability to trace its topology through adjacent wires and edges. This is why a (wire) body is required as input to these operations.

When a wire body is offset, sometimes gaps are created at certain junctures. Consider a wire body made up of a square and a circle that is offset towards the inside, as shown in figure 8-1. The resulting gaps can be filled as one of three types:

Natural . . . . . Extends the two shapes along their natural curves; e.g., along the circle and along the straight edge, until they intersect.

Round . . . . . Creates a rounded corner between the two segments.

Extend . . . . . Draws two straight lines from the ends of each segment until they intersect.

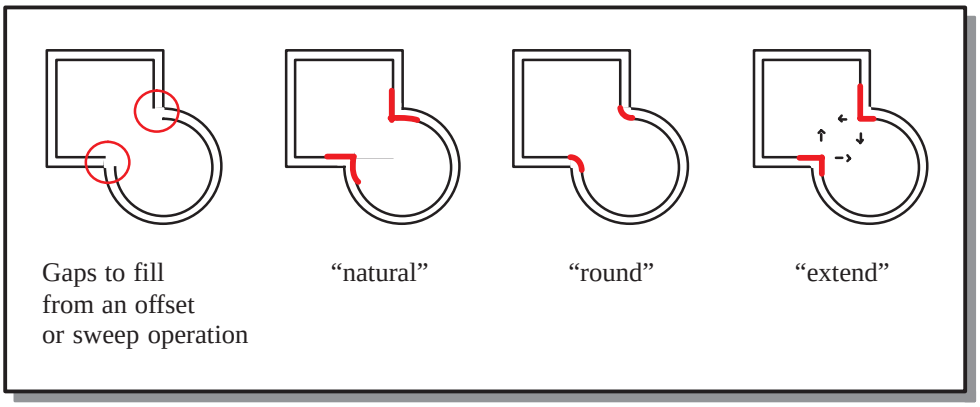


Figure 8-1. Gap Fill Types

In the following example, four linear edges are created. Even though the end points of each edge were picked such that the edges appeared to intersect in a simple rectangular frame, the edges are independent and know nothing about one another. In other words, the edges are separate topological entities but are not part of a topology.

Offsetting occurs to the outside of a rectangle. By default, the wire offsetting fills the external gap using the “round” option. The result is a second wire body that is mostly rectangular except for rounded corners. The radius of the rounded corner is the wire offset amount. Refer to figure 8-2.

Scheme Example

```
; Define the positions to be used
(define my_point1 (position 0 0 0))
;; my_point1
(define my_point2 (position 20 0 0))
;; my_point2
(define my_point3 (position 20 10 0))
;; my_point3
(define my_point4 (position 0 10 0))
;; my_point4

; Create edge 1 as a sine edge.
(define my_linear1 (edge:linear my_point1 my_point2))
;; my_linear1
; Create edge 2 as a linear edge.
(define my_linear2 (edge:linear my_point2 my_point3))
;; my_linear2
; Create edge 3 as a spline curve.
(define my_linear3 (edge:linear my_point3 my_point4))
;; my_linear3
; Create edge 4 as a spline curve.
(define my_linear4 (edge:linear my_point4 my_point1))
;; my_linear4

; Create a wire-body from a list of edges.
(define my_wirebody (wire-body
    (list my_linear1 my_linear2 my_linear3 my_linear4)))
;; my_wirebody
(define my_offset (wire-body:offset my_wirebody 5 "r"))
;; my_offset
; If desired, you could delete the original wire body;
; it is no longer needed.
;(entity:delete my_wirebody)
;; ()
; Save the results to an output file.
(part:save "tmpoffset1.sat")
;; #t
```

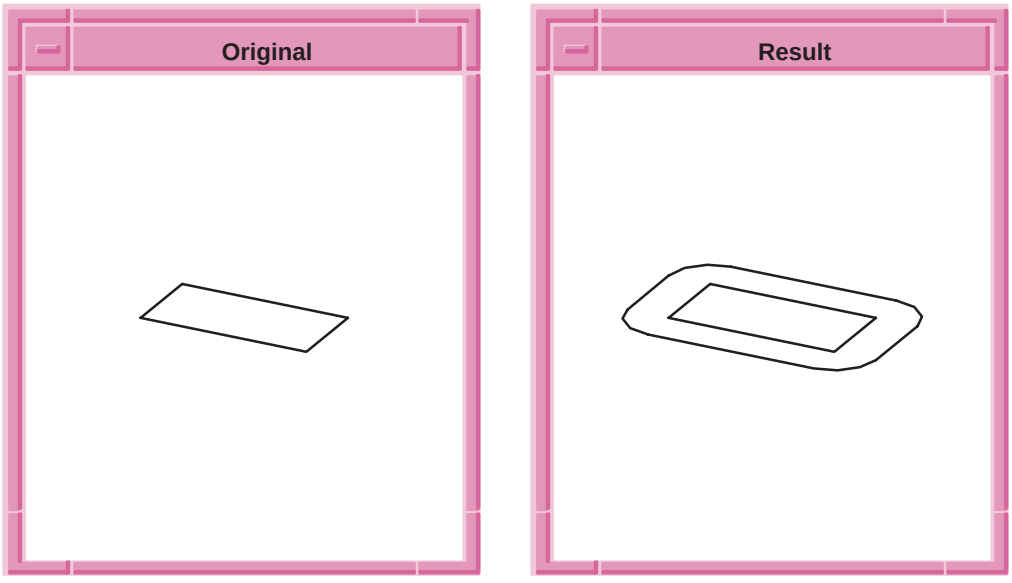


Figure 8-2. Wire Body Offset

Some of the Scheme extensions related to wire bodies are:

wire–body . . . . . Creates a wire body from a list of edges.

wire–body:kwire . . . . . Creates a planar wire specified by a sequence of points and “bulge” factors.

wire–body:offset . . . . . Creates a new wire by offsetting the given wire using offset and twist laws.

wire–body:points . . . . . Creates a wire body from a list of positions.

Creating Surfaces

Topic: *Creating Surfaces, *Skinning and Lofting

ACIS contains many standard surface types that can be created quickly, and if desired, already as face topological elements. These include planar faces, cylindrical faces, conical faces, spherical faces, torus faces, spline faces, and faces using law mathematical functions.

ACIS also provides techniques for creating more free-form surfaces. These techniques include covering, skinning, lofting, and net surfaces, all of which create a surface from a wireframe or edges.

Covering . . . . . Requires that the edges be a closed loop over which a surface is fit.

Skinning . . . . . Passes a surface through a disjoint set of edges.

Lofting . . . . . Starts with a surface and extends this surface to pass through a disjoint set of edges.

Net surfaces . . Stretches a surface across a set of bi-directional curves.

Of these techniques, covering is the easiest but least flexible. Aside from requiring a bounding closed loop of edges, covering provides little control over the internal shape of the surface. The initial wire body does not have to be planar. However, in a nonplanar case, unique surface characteristics away from the bounding edges cannot be specified.

Net surfaces probably provides the most control over the shape of the resulting surface, but requires more planning in defining the *uv* curves, in getting the *u* curves to intersect with or be within a specified tolerance of the *v* curves, and in selecting an appropriate tolerance for interpolation.

In the following example, which is illustrated in figure 8-3, interpolates a surface through a network of bi-directional curves. The given wires define the cross-sections to be interpolated by the resulting sheet body. There must be at least four wire bodies, two in each direction. The start points of the curves in the *v* direction must lie on the first curve in the *u* direction, and vice versa. The end points of the *v* curves must lie in the last curve in the *u* direction, and vice versa.

If all of the curves intersect, then the surface passes through the curves and their intersections. If any of the *u* curves of the network do not intersect all of *v* curves at some point, the intersection is interpolated. The maximum distance for the interpolation is governed by the tolerance argument. The default for this tolerance value is SPAresfit.

Scheme Example

```
; Establish the correct options for viewing.
(option:set ”sil” #f)
;; #t
(option:set ”u_par” 5)
;; –1
(option:set ”v_par” 7)
;; –1
```

```
; Create a series of points to use later for splines.
(define v1 (list (position 0 0 0) (position 5 10 0)
    (position 10 5 0) (position 15 15 0)
    (position 20 0 0)))
;; v1
(define v2 (list (position 0 10 5) (position 5 5 5)
    (position 10 15 5) (position 15 10 5)
    (position 20 10 5)))
;; v2

(define v3 (list (position 0 20 10)
    (position 5 15 10) (position 10 20 10)
    (position 15 5 10) (position 20 20 10)))
;; v3
(define v4 (list (position 0 15 15)
    (position 5 10 15) (position 10 15 15)
    (position 15 0 15) (position 20 15 15)))
;; v4

(define u1 (list (position 0 0 0) (position 0 10 5)
    (position 0 20 10) (position 0 15 15)))
;; u1
(define u2 (list (position 10 5 0) (position 10 15 5)
    (position 10 20 10) (position 10 15 15)))
;; u2
(define u3 (list (position 20 0 0) (position 20 10 5)
    (position 20 20 10) (position 20 15 15)))
;; u3

; Create a series of spline curve wire-bodies.
; in the u and v direction.
(define my_v1 (wire-body (edge:spline v1)))
;; my_v1
(define my_v2 (wire-body (edge:spline v2)))
;; my_v2
(define my_v3 (wire-body (edge:spline v3)))
;; my_v3
(define my_v4 (wire-body (edge:spline v4)))
;; my_v4

(define my_u1 (wire-body (edge:spline u1)))
;; my_u1
(define my_u2 (wire-body (edge:spline u2)))
;; my_u2
(define my_u3 (wire-body (edge:spline u3)))
;; my_u3
```

```
; Create a net surface from the uv curves.
(define net1 (sheet:net-wires
    (list my_v1 my_v2 my_v3 my_v4)
    (list my_u1 my_u2 my_u3) #t ))
;; net1
; Save the results to an output file.
(part:save "tmpnetsurf1.sat")
;; #t
```

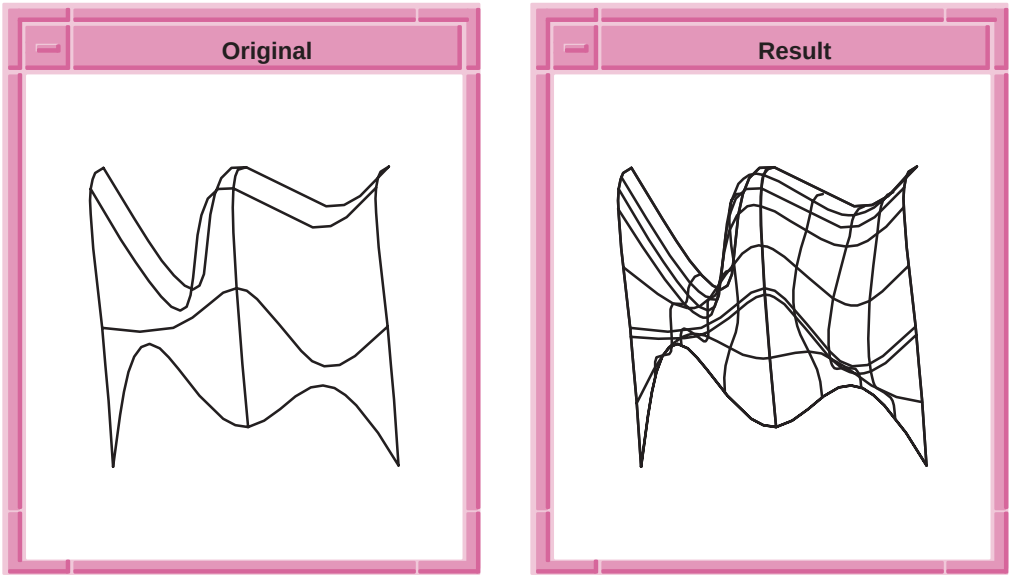


Figure 8-3. Net Surface

Some of the Scheme extensions related to surfaces are:

face:cone . . . . . Creates a conical face relative to the active WCS.

face:cylinder . . . . . Creates a cylindrical face relative to the active WCS.

face:intersect . . . . . Gets the intersection curve between two faces.

face:law . . . . . Creates a new face whose uv parameters are determined by law functions.

face:offset . . . . . Creates a new face offset from a given face.

face:plane . . . . . Creates a planar face.

face:sphere . . . . . Creates a spherical face.

face:spline–ctrlpts . . . . . Creates a spline face using control points.

face:spline–grid . . . . . Creates a spline face using a grid.

face:torus . . . . . Creates a toroidal face.

section . . . . . Creates a data structure used as input to the sheet:loft–wire extension.

sheet:cover . . . . . Modifies a sheet body by covering each of its simple loops of external edges by a face.

sheet:cover–wires . . . . . Creates a sheet body from a wire body.

sheet:face . . . . . Creates a sheet body from a given face.

sheet:loft–wires . . . . . Creates a sheet body that interpolates a series of wires.

sheet:net–wires . . . . . Creates a sheet body that interpolates a series of wires.

sheet:planar–wire . . . . . Creates a planar sheet body from a planar wire body.

sheet:skin–wires . . . . . Creates a sheet body that interpolates a series of wires.

Creating Solids

Topic: *Creating Solids

ACIS is a solid modeler that can combine wireframe, surface, and solid models. A *wireframe model* defines an object only by its edges and vertices. A *surface model* is similar to a wireframe model, but defines an object by its visible surfaces, including faces. A *solid model* defines an object in terms of its size, shape, density, and physical properties (weight, volume, center of gravity, etc.).

In the preceding sections, only the functionality of wireframe and surface models were exercised. In this section, solid models are explored. ACIS provides a variety of ways to create solids. The examples in this chapter create solids by:

- Using solid primitives
- Enclosing a void

- Sweeping a surface

Solid Primitives

Topic: *Creating Solids

ACIS provides many standard basic solid types that can be created quickly, and if desired, already as body topological elements. These include blocks, cones, cylinders, prisms, pyramids, spheres, tori, and “wiggles.” These are solid *primitives*.

The following example creates several simple solid primitives.

Scheme Example

```
; In the previous Scheme example, options were changed.
; If not already done so through a restart,
; turn them back to the defaults.
(option:set ”sil” #t)
;; #f
(option:set ”u_par” –1)
;; 5
(option:set ”v_par” –1)
;; 7

; Create solid block.
(define my_block
    (solid:block (position 10 10 10)
    (position 15 20 30)))
;; my_block

; Create solid cylinder.
(define my_cyl
    (solid:cylinder (position –10 –5 –15)
    (position 0 –15 –15) 10))
;; my_cyl

; Create solid cone.
(define my_cone
    (solid:cone (position 20 –15 0)
    (position 40 –15 10) 10 2))
;; my_cone

; Create a solid prism.
(define my_prism (solid:prism 10 5 5 6))
;; my_prism

; Create a solid pyramid.
(define my_pyramid (solid:pyramid 5 10 15 10 15))
;; my_pyramid
```

```
; Create solid sphere.
(define my_sphere (solid:sphere (position –20 20 30) 5))
;;my_sphere

; Create solid torus.
(define my_torus (solid:torus (position 20 –20 –20) 7 3))
;; my_torus

; Save the results to an output file.
(part:save ”tmpsolprim1.sat”)
;; #t
```

Some of the Scheme extensions related to creation of standard solids are:

solid:block	Creates a solid block.
solid:cone	Creates a right circular cone.
solid:cylinder	Creates a right circular cylinder.
solid:prism	Creates a solid prism.
solid:pyramid	Creates a solid pyramid.
solid:sphere	Creates a sphere centered at the specified position.
solid:torus	Creates a solid torus.
solid:wiggle	Creates a rectangular block with a spline surface top.

Using Surfaces to Enclose a Void

Topic: *Creating Solids

The techniques for creating surfaces provide a significant amount of flexibility in creating user-defined, free-form models. However, surface models are just that: models of surfaces. Surfaces have no depth and aren’t solids. Even when the edges of the surfaces are cleverly chosen to match the edge boundaries of other surfaces, the model will not be a solid—unless explicitly defined as such.

If you create surfaces individually, you can combine the surfaces into a volume. If there was overlapping at the individual surface boundaries, the overlap remains. All faces, edges, and vertices that are not necessary to support the topology of the solid entity can be removed.

Some of the Scheme extensions related to solid creation from a surface are:

sheet:2d . . . . . Modifies a single–sided sheet body into a double–sided sheet body.

sheet:enclose . . . . . Converts a closed 2D sheet into a solid volume.

Sweeping Surfaces into Solids

Topic: *Creating Solids, *Sweeping

ACIS also provides techniques for creating more free-form solids. These include revolving faces about an axis, revolving wires about an axis, sweeping planar faces along a path, rigid sweeping, and using law mathematical functions to do nonplanar sweeps.

The most flexible of the solid creation techniques is sweeping using law mathematical functions. Laws in sweeping permit:

Drafting the profile . . . . . As the profile is swept, the ending profile has been offset an equal distance, which facilitates removal of the item from a mold.

Twisting the profile . . . . . This enables not only solid contours whose profile "corkscrews" around the sweep path, but also solid contours whose profile meanders or zigzags along the sweep path.

Orienting the profile . . . . . Twisting orients the profile based on the "travel distance" along the path. A rail law, on the other hand, specifies the orientation of the profile with respect to local features of the sweep path.

Filling gaps . . . . . The curves can be extended along their natural curves, along straight lines until they intersect, or as rounded corners.

Scaling the profile . . . . . Scaling results in proportional changes that do not affect the topology.

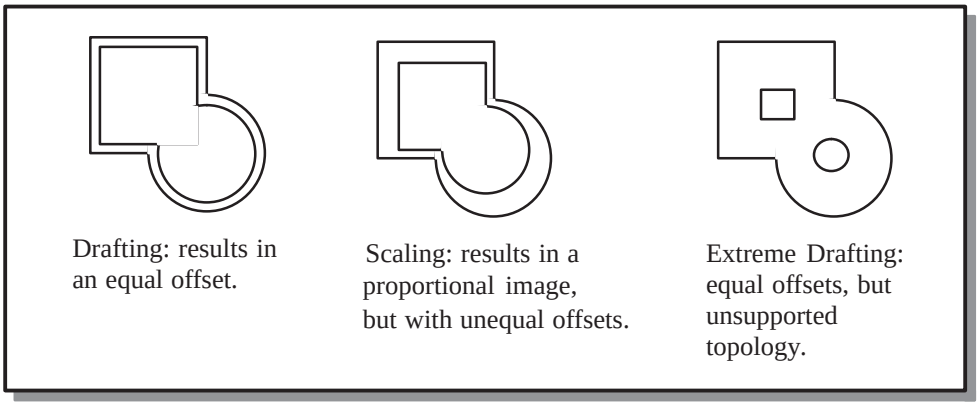


Figure 8-4. Drafting versus Scaling

As is evident from the control that laws add to sweeping, sweeping is a very powerful tool in creating free-form solids. The sweep path no longer is restricted to being planar. Sweeping does have the limitation that only planar (surface) profiles can be swept. However, a profile can be swept to meet a given face and that face does not have to be planar. Moreover, Boolean operations using nonplanar profiles can be performed on the end caps of the swept body to achieve the desired shape.

When a wire body is swept, it can be specified whether or not end capping is performed. In other words, the result can be specified to be either a solid or a "straw." When a surface (profile) is swept, it can be either one-sided or two-sided. The resulting solid is created with the end caps being single-sided and having their orientation in the proper direction for a solid.

The following example creates a wire offset. Because the wire offset in the example is planar and bounded, it can be used as a wire body profile for a solid sweep operation. Although not required, the wire body could be made into a simple surface profile using covering techniques, such as sheet:cover–wires. In addition, the simple surface profile can be (optionally) turned into a double-sided sheet using sheet:2d.

Scheme Example

```
;--------- Copy of the Offsetting a Wire Body Example
;--------- (below this line)
; Define the positions to be used
(define my_point1 (position 0 0 0))
;; my_point1
(define my_point2 (position 20 0 0))
;; my_point2
(define my_point3 (position 20 10 0))
;; my_point3
(define my_point4 (position 0 10 0))
;; my_point4

; Create edge 1 as a sine edge.
(define my_linear1 (edge:linear my_point1 my_point2))
;; my_linear1
; Create edge 2 as a linear edge.
(define my_linear2 (edge:linear my_point2 my_point3))
;; my_linear2
; Create edge 3 as a spline curve.
(define my_linear3 (edge:linear my_point3 my_point4))
;; my_linear3
; Create edge 4 as a spline curve.
(define my_linear4 (edge:linear my_point4 my_point1))
;; my_linear4

; Create a wire-body from a list of edges.
(define my_wirebody (wire-body
    (list my_linear1 my_linear2 my_linear3 my_linear4)))
;; my_wirebody
(define my_offset (wire-body:offset my_wirebody 5 "r"))
;; my_offset
(entity:delete my_wirebody)
;; ()
;--------- Copy of the Offsetting a Wire Body Example
;--------- (above this line)

(define my_profile my_offset)
;; my_profile
; Create a list of edges for use later.
(define my_list (entity:edges my_profile))
;; my_list

; Create a vector path for sweeping.
(define my_vec_path (gvector 0 0 20))
;; my_vec_path
```

```
; Sweep the profile along this vector. Create it as a solid.
(define my_sweep (sweep:along–vector my_profile #t
    my_vec_path))
;; my_sweep
; Save the results to an output file.
(part:save "tmpsweep1.sat")
;; #t
```

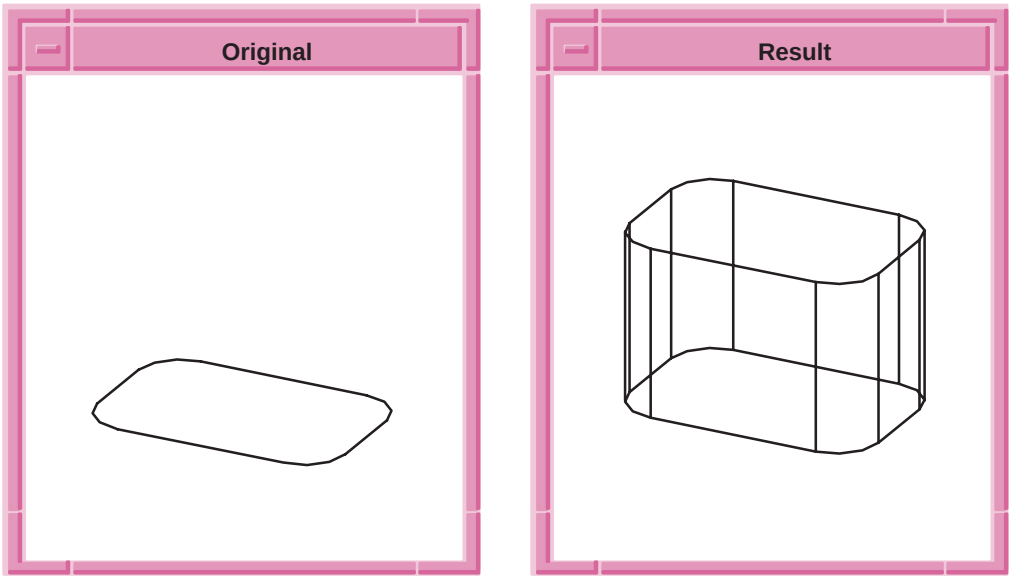


Figure 8-5. Sweeping a Surface to a Solid

This sweeping example does not use laws, because the sweeping path is planar and it is desired that the resulting surface have no twist. It defines a vector to be used for sweeping. The offset profile is swept along this vector into a solid.

A new variable, my_profile, which references the same entity as my_offset is defined in this example. Its purpose is to make the example somewhat clearer in that the created wire offset, my_offset, is now used as the profile for the sweeping operation.

Some of the Scheme extensions related to sweeping are:

solid:revolve–face . . . . . Creates a solid by revolving a face about an axis.

solid:revolve–wire . . . . . Creates a solid by revolving a wire about an axis.

solid:sweep–face	Creates a solid by sweeping a planar face along a vector.
solid:sweep–wire	Creates a solid by sweeping a planar wire along a vector.
sweep:about–axis	Sweeps a face or a wire about an axis as a sheet or as a solid.
sweep:law	Creates a surface or solid by sweeping a profile along a path.
sweep:rigid	Does a rigid sweep of a face or a wire along a path or a vector.

Modifying Models

Topic: *Modifying Models

Once the general shape of a solid is established, it can be refined through other processes. These might include:

- Moving and/or rotating the model, or transforming it in some other way
- Uniting or intersecting it with another solid
- Subtracting it from another solid, or vice versa
- Adding blends to the model, such as fillets and chamfers
- Performing local operations on the model, such as offsetting
- Performing shelling on the model to create a hollow solid body
- Performing free-form deformations on the model

Transforming a Model

Topic: *Modifying Models, *Transforms

Many ACIS creation techniques for solid and surface primitives are very basic. They tend to create objects at the working coordinate system origin with simple geometric relationships. The reason for this is that it simplifies the syntax of the primitive operations if positioning and orientation are left to later steps. Because of this, model entities may need to be moved.

ACIS refers to this operation as a *transformation*. A transformation can be used for such things as scaling, reflection, inversion, rotation, etc., in addition to repositioning.

Internally, ACIS transformations are essentially matrix operations performed on the modeling entity. When combining transformations and/or applying them to an entity, ACIS performs the more complex linear algebra and matrix tasks for you, allowing you to define in a more general sense the operation desired.

To move entities to more suitable locations, the Scheme extension entity:transform can be used with defined transformations.

The following example creates a simple block with one corner at the origin. A translation transformation is defined that moves the block along a vector. A rotation transformation is defined that rotates the block 45 degrees around a vector.

Scheme Example

```
; Create a solid block.
(define my_block (solid:block (position 0 0 0)
    (position 10 15 20)))
;; my_block
; my_block => #[entity 2 1]
; Create a copy of the block.
(define my_block2 (entity:copy my_block))
;; my_block2 => #[entity 3 1]
; Create a transform to move the block.
(define my_t_move (transform:translation (gvector 10 12.5 0)))
;; my_t_move
(define my_t_rotate (transform:rotation
    (position 15 20 0) (gvector –1 0 1) 45))
;; my_t_rotate
(define my_transform (transform:compose
    my_t_move my_t_rotate))
;; my_transform
; Apply the transform to the block.
(entity:transform my_block my_transform)
;; #[entity 2 1]
; Refresh the view so that both entities are visible.
(view:refresh)
;; #[view 293023412]
; Save the results to an output file.
(part:save ”tmptrans1.sat”)
;; #t
```

Some of the Scheme extensions related to transformations on models are:

afig:apply–transform . . . . . Apply a transform to an animation–figure.

afig:get–transform . . . . . Get the current transformation for an animation–figure.

afig:set–transform . . . . . Set the transform for an animation–figure.

curve:transform . . . . . Modifies a curve or edge by applying a transform.

entity:fix–transform . . . . . Applies a body transformation to all underlying geometry.

entity:transform . . . . . Applies a transform to a single entity or list of entities.

gvector:transform . . . . . Applies a transform to a gvector.

position:transform . . . . . Applies a transform to a position.

transform:axes . . . . . Creates a transform that takes an object from model space to the space defined by the new origin and axes.

transform:compose . . . . . Concatenates two transforms.

transform:copy . . . . . Copies a transform.

transform:identity . . . . . Creates an identity transform.

transform:inverse . . . . . Creates an inverse transform.

transform:reflection . . . . . Creates a transform to mirror an object through an axis.

transform:rotation . . . . . Creates a transform to rotate an object about an axis.

transform:scaling . . . . . Creates a scaling transform.

transform:translation . . . . . Creates a translation transform.

transform? . . . . . Determines if a Scheme object is a transform.

wcs:from–transform . . . . . Creates a work coordinate system given a transform.

wcs:to–model–transform . . . . . Gets the transform of the active WCS to model space.

wcs:to–wcs–transform . . . . . Gets the transform from the active WCS to the specified WCS.

Using Booleans

Topic: *Modifying Models, *Booleans

Boolean operations illustrate the true power of ACIS and solid models. Booleans are used to unite or intersect two solid bodies, or to subtract one solid body from another.

The following example creates a block and a cylinder and uses Boolean operations to unite them. Instead of solid:unite, solid:intersect or solid:subtract could have been employed to create other unique solids.

Scheme Example

```
; Create a solid block.
(define my_block (solid:block (position –20 –20 –20)
    (position 20 20 20)))
;; my_block
; Create a cylinder.
(define my_cyl (solid:cylinder (position 20 0 –20)
    (position 20 0 20) 20)))
;; my_cyl
; Unite the two bodies into a new body.
(define my_united (solid:unite my_block my_cyl))
;; my_united
; Save the results to an output file.
(part:save ”tmpbool1.sat”)
;; #t
```

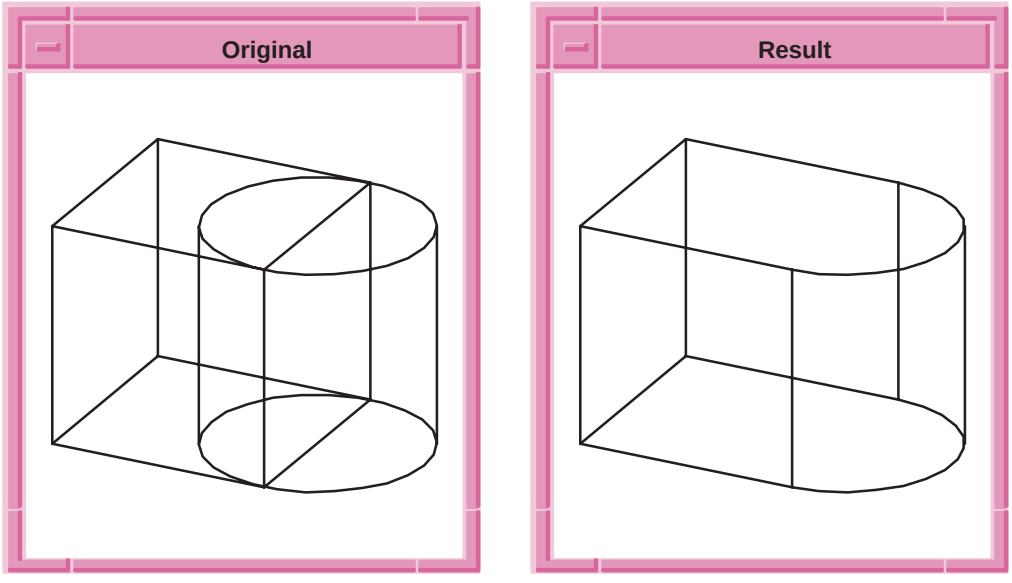


Figure 8-6. Unite Block and Cylinder

Some of the Scheme extensions related to Boolean operations on models are:

body:combine . . . . . Combines a list of solid and/or wire bodies into a single body.

body:separate	Separates a single disjoint body into multiple entities.
bool:intersect	Intersects two or more bodies.
bool:merge	Combines faces and edges of equivalent geometry.
bool:merge–faces	Combines specific faces on a body.
bool:nonreg–intersect	Intersects two or more nonregularized bodies.
bool:nonreg–subtract	Subtracts one or more nonregularized bodies from a body.
bool:nonreg–unite	Unites two or more nonregularized bodies.
bool:regularise	Regularizes an entity.
bool:subtract	Subtracts one or more bodies from a body.
bool:unite	Unites two or more bodies.
curve:intersect	Gets the intersection between two edges or curves.
edge:project–to–plane	Projects an edge onto a plane.
edge:split	Splits an edge into two entities at a specified position.
edge:trim–intersect	Trims both edges to the intersection of the edges.
face:intersect	Gets the intersection curve between two faces.
position:project–to–line	Gets the projection of a position on to a line.
position:project–to–plane	Gets the projection of a position onto a plane.
solid:imprint	Imprints curves of intersection of two bodies onto the faces of bodies.
solid:imprint–stitch	Joins body1 and body2 along the intersection graph.
solid:intersect	Intersects a list of solids.
solid:planar–slice	Slices a solid body with a plane to produce a wire body.

solid:slice . . . . . Gets the intersection graph between two bodies and returns it as a wire body.

solid:split . . . . . Splits all periodic faces of a body along the seams.

solid:stitch . . . . . Joins body1 and body2 along edges or vertices of identical geometry.

solid:subtract . . . . . Subtracts a list of solids from a solid.

solid:unite . . . . . Unites two solids.

Blending a Solid

Topic: *Modifying Models, *Blending

Blending is used frequently in mechanical engineering designs. It permits:

- Chamfering or rounding sharp edges
- Adding a chamfer or fillet between two faces (e.g., adding material)
- Variable blending and chamfering (blend radius or chamfer size are not constant)

The following example picks up where Sweeping example left off. In the sweeping example, a wire body consisting of linear edges was created. This is offset, creating another wire body that has rounded corners. This wire body is then swept into a solid. The blending example takes some of the edges of that solid and applies a constant radius blend to them, as is shown in figure 8-7.

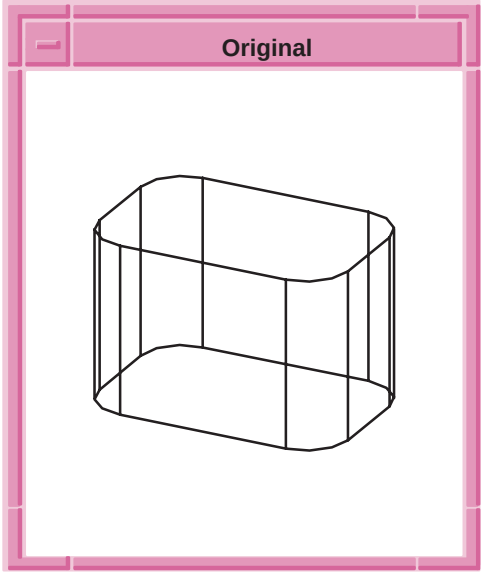
After creating the swept solid, defines the blend radius and assigns this to a list of edges. The list of edges, my_list, is simply the list of edges obtained from the wire offset operation and then later used as the profile in sweeping. Either a single edge or a group of edges can have the blend attribute assigned to it. Once the edges have a blend attribute, the network of blend edges is passed into the blend:network Scheme extension, which completes the blend.

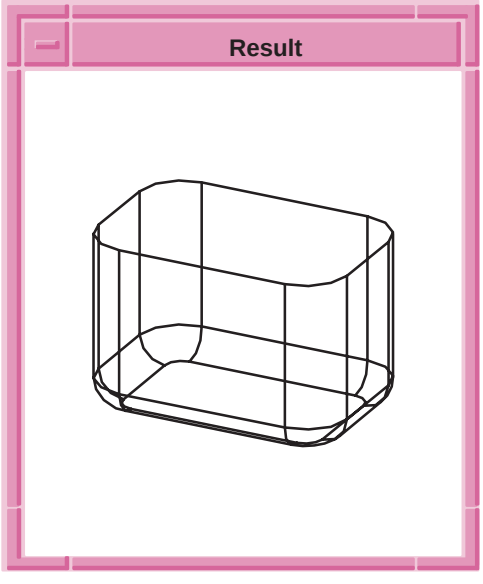
Note *The* blend:get–network *Scheme extension is used to get all edges associated with a blend attribute from just one edge of the list.*

Scheme Example

```
;--------- Copy of the Offsetting a Wire Body Example
;--------- (below this line)
; Define the positions to be used.
(define my_point1 (position 0 0 0))
;; my_point1
(define my_point2 (position 20 0 0))
;; my_point2
(define my_point3 (position 20 10 0))
;; my_point3
(define my_point4 (position 0 10 0))
;; my_point4

; Create edge 1 as a sine edge.
(define my_linear1 (edge:linear my_point1 my_point2))
;; my_linear1
; Create edge 2 as a linear edge.
(define my_linear2 (edge:linear my_point2 my_point3))
;; my_linear2
; Create edge 3 as a spline curve.
(define my_linear3 (edge:linear my_point3 my_point4))
;; my_linear3
; Create edge 4 as a spline curve.
(define my_linear4 (edge:linear my_point4 my_point1))
;; my_linear4

; Create a wire-body from a list of edges.
(define my_wirebody (wire-body
    (list my_linear1 my_linear2 my_linear3 my_linear4)))
;; my_wirebody
(define my_offset (wire-body:offset my_wirebody 5 "r"))
;; my_offset
(entity:delete my_wirebody)
;; ()
(define my_profile my_offset)
;; my_profile
```

```
; Create a list of edges for use later.
(define my_list (entity:edges my_profile))
;; my_list
; Create a vector path for sweeping.
(define my_vec_path (gvector 0 0 20))
;; my_vec_path
; Sweep the profile along this vector. Create it as a solid.
(define my_sweep (sweep:along-vector my_profile #t
    my_vec_path))
;; my_sweep
;--------- Copy of the Offsetting a Wire Body Example
;--------- (above this line)


; Modify the model by blending all of the edges along
; the base.
(blend:const-rad-on-edge my_list 3)
;; (#[entity 19 1] #[entity 23 1])
(blend:network (blend:get-network (car my_list)))
;; #[entity 19 1]
; Save the results to an output file.
(part:save "tmpblend1.sat")
;; #t
```





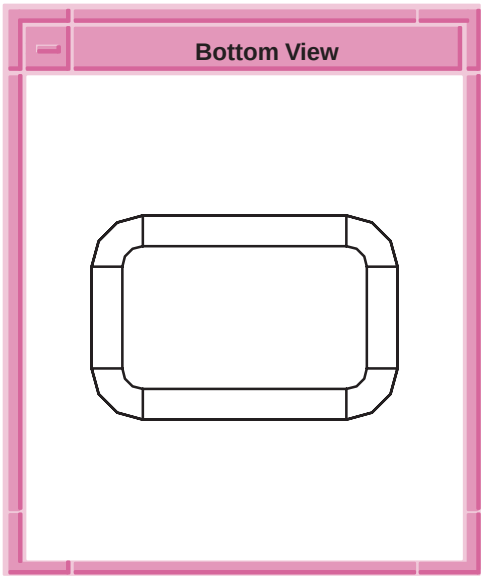


Figure 8-7. Blending Edges

Some of the Scheme extensions related to blending solids are:

blend:chamfer–on–edge . . . . . Attaches a chamfer blend attribute to each edge in the input list.

blend:complete . . . . . Executes the third phase of blending.

blend:const–rad–on–edge . . . . . Attaches a constant radius blend attribute to each edge in the input list.

blend:edge–info . . . . . Lists the edge blend type and internal data values.

blend:get–network . . . . . Gets a list of all edges and vertices that are in the same blend network as the given entity.

blend:get–smooth–edges . . . . . Gets a list of all edges that smoothly connect to a given edge.

blend:local . . . . . Creates a blend in a nonmanifold body by adding faces locally around a given edge, blending the edge, and removing the added faces.

blend:make–sheet . . . . . Executes the first phase of blending which creates a blend sheet.

blend:make–wire	Executes the second phase of blending which creates a wire intersection.
blend:network	Creates blends on a list of edges and vertices that make up a single blend network.
blend:on–vertex	Attaches vertex blend attributes to each vertex in the input list.
blend:remove–from–edge	Removes blend attributes, if any, from the input edge.
blend:remove–from–vertex	Removes blend attributes, if any, from the input vertex.
blend:var–rad–on–edge	Attaches a variable radius blend attribute to each edge in the input list.
blend:vertex–info	Lists the vertex blend type and internal data values.
edge:fillet	Creates a fillet blend on two edge entrays (entities with rays).
solid:blend–edges	Creates a cylindrical blend on a list of edges.
solid:chamfer–edges	Creates a chamfer blend on a list of edges.

Local Operations, Shelling, and Deformable Surfaces

Topic: Ignore

The Local Operations Component, Shelling Component, and ACIS Deformable Modeling Component all provide specialized functionality for modifying models.

Local Operations

Topic: *Modifying Models, Local Operations

The following example shows the taper faces local operation.

Scheme Example

```
(define my_block2 (solid:block
    (position –25 –25 –10)(position 25 25 10)))
;; my_block2
; my_block2 => #[entity 2 1]
(solid:blend–edges (pick:edge (ray
    (position 0 0 0) (gvector 1 –1 0))) 20)
;; (#[entity 2 1])
```

```
(lop:taper-faces (list (pick:face
    (ray (position 0 0 0) (gvector 1 0 0)))
    (pick:face (ray (position 0 0 0)
    (gvector 0 -1 0))) (pick:face
    (ray (position 0 0 0) (gvector 1 -1 0))))
    (position 0 0 -10) (gvector 0 0 1) 45)
;; #[entity 2 1]
; Save the results to an output file.
(part:save "tmplocop1.sat")
;; #t
```

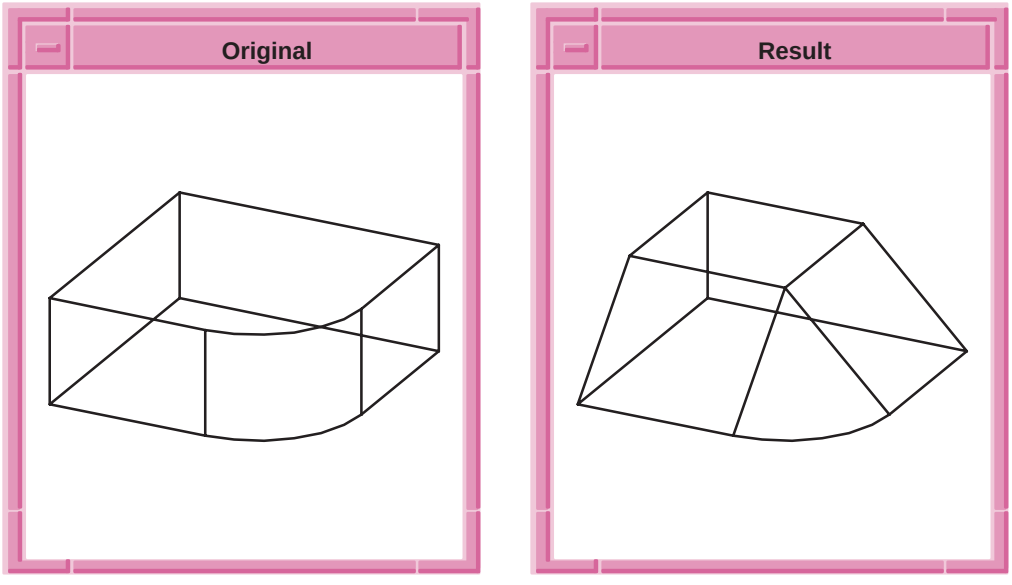


Figure 8-8. Taper Faces Local Operation

Shelling

Topic: *Modifying Models, *Shelling

The following example shows how a shelling operation works by creating a hollow solid body.

Scheme Example

```
; shell:hollow-body
; Create a solid block.
(define my_block (solid:block (position -20 -20 -20)
    (position 20 20 20)))
;; my_block => #[entity 1 1]
(define my_face (car (entity:faces my_block)))
;; my_face => #[entity 2 1]
; Hollow the body
(shell:hollow-body my_face 10)
    (gvector 0 0 1))) 10)
;; #[entity 1 1]
; Save the results to an output file.
(part:save "tmpshell1.sat")
;; #t
```

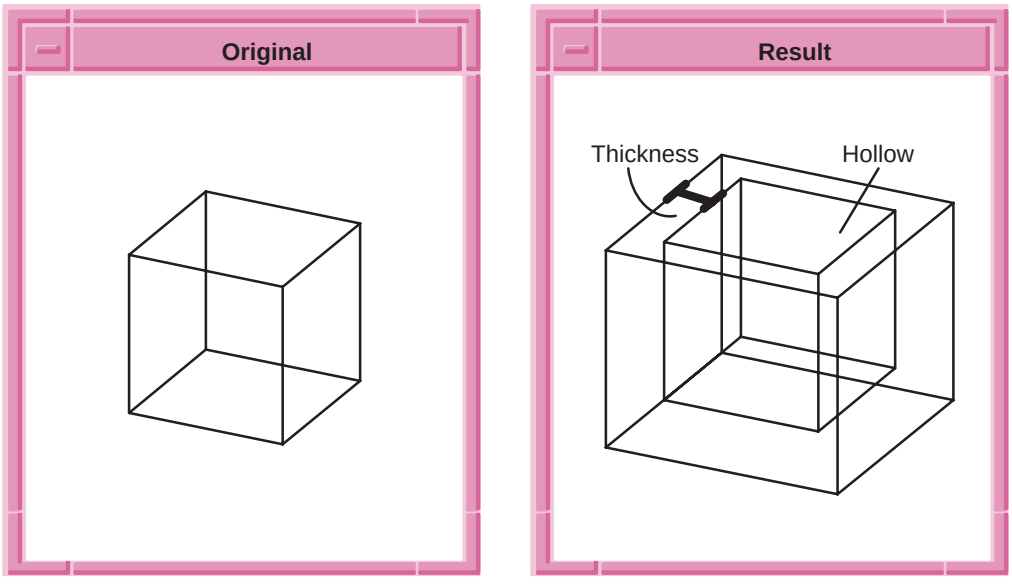


Figure 8-9. Hollow Body

Deformable Modeling

Topic: *Modifying Models, *Deformable Surfaces

The following example shows how to perform free-form deformations on a simple face.

Scheme Example

```
; Make a test face
(define my_dsmodel ( ds:test–face 6 6 36 36 0))
;; my_dsmodel

; Show the load and constraints
( ds:set–draw–state my_dsmodel 1 (+ 4 8))
;; ()

; Make a spring and add it to the test–face
(define my_spring (ds:add–spring my_dsmodel 1
    (par–pos 0.5 0.5) (position 10 10 15) 100))
;; my_spring
; Solve to see deformation effect
(ds:solve my_dsmodel)
;; ()

; Compress the spring and display
(ds:set–load–gain my_dsmodel my_spring 1000 #t)
;; 4
(ds:solve my_dsmodel)
;; ()
; The surface should just barely be deformed
; increasing the spring gain reduces the distance
; between the spring end points.

; Compress the spring and display
(ds:set–load–gain my_dsmodel my_spring 10000 #t)
;; 4
(ds:solve my_dsmodel)
;; ()

; Compress the spring and display
(ds:set–load–gain my_dsmodel my_spring 100000 #t)
;; 4
(ds:solve my_dsmodel)
;; ()
```

```
; The spring displaces the surface by larger
; ang larger amounts.
; Commit the changes back to the model so it can be saved.
(ds:commit my_dsmodel)
;; ()
; Save the results to an output file.
(part:save "tmpdmod1.sat")
;; #t
; When loading this part, the deformed surface can be
; seen by rendering it or turning on silhouette lines.
```

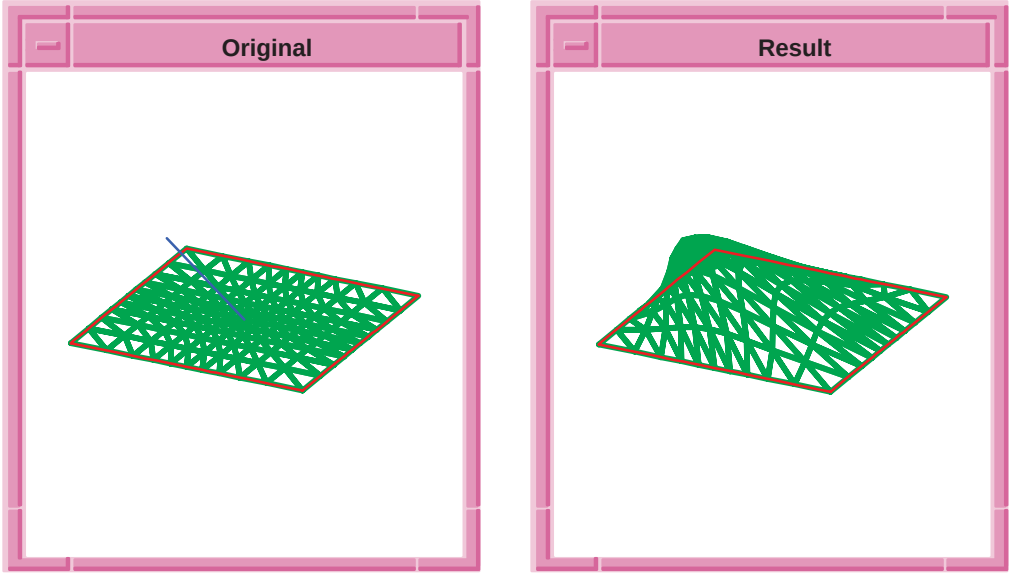


Figure 8-10. Free–Form Deformations on a Face

Analyzing Models

Topic: *Analyzing Models, *Geometric Analysis, *Physical Properties

Creation and modification can be considered the major tasks associated with solid modeling. However, analysis is also important because it can help define geometry and can verify model geometry.

An earlier section discussed how transformations are used to orient and position entities, such as solid primitives, after creation. Suppose that a unique surface *A* was created using some surface creation technique, such as net surfaces or skinning, and that a newly created entity *B* is now to be positioned and oriented with respect to some geometric property of *A*. While transformations may provide a means of moving and orienting entity *B*, analysis helps determine where to move entity *B* and how to orient it.

ACIS has four main areas of analysis:

Physical Properties . . . . . Provides surface area, volume, and center of gravity information about a model.

Object Relationships . . . . . Provides information regarding proximity on a plane, on a line, parallelism, perpendicularity, etc.

Geometric Analysis . . . . . Uses ACIS laws to find out specific mathematical answers about geometric elements, such as the *n*th derivative of a curve at a given point, numerical maxima and minima, etc.

Cellular Topology . . . . . Doesn't perform any analysis in its own right, but is a data structure for storing information about the model that custom applications can employ for their own analysis operations.

All geometric entities provide methods for querying properties. However, the data structures do not explicitly store information such as the point of the curve where the radius of curvature is minimized, the tangent vector from a given point on an edge, the normal vector from a given point on a surface, or the numerical *n*th derivatives of points along a curve to determine continuity. There are many geometric properties that you might want to know when analyzing your model.

ACIS laws provide powerful analysis features. Not only can independent calculations be carried out on complex mathematical functions, but calculations can also be carried out using geometric entities as input. ACIS laws expand significantly the type of analysis that can be performed in creating and manipulating models. The complexity of the analysis is determined by the complexity of the defining laws and geometric elements.

In the following example, illustrates determination of the maximum radius of curvature of a point on a spline edge. After the spline edge is created, it is passed into a curvature law function and parameterized from 0 to 1. Then, the numerical maximum of curvature is determined for that edge. This is illustrated in figure 8-11.

Scheme Example

```
; Define a list of points for a spline edge.
(define my_plist (list
    (position 0 0 0)(position 20 20 0)
    (position 40 0 0)(position 60 25 0)
    (position 40 50 0)(position 20 30 0)
    (position 0 50 0)))
;; my_plist
(define my_start (gvector 1 1 0))
;; my_start
(define my_end (gvector –1 1 0))
;; my_end
(define my_testcur
    (edge:spline my_plist my_start my_end))
;; my_testcur
(define my_law
    (law ”map(Curc(edge1),edge1)” my_testcur))
;; my_law
(define my_maxpoint (law:nmax my_law 0 1))
;; my_maxpoint
; All curves in Scheme are parameterized from 0 to 1.
(define my_maxposition (curve:eval–pos
    my_testcur my_maxpoint))
;; my_maxposition
; my_maxposition =>
; #[position 19.9659145308554 29.9949779948174 0]
(define my_point (dl–item:point my_maxposition))
;; my_point
; Create a display list item so the point can be
; viewed. This point won’t be saved to a SAT file.
(dl–item:display my_point)
;; ()
; To no longer display this point, use:
; (dl–item:erase my_point)
```

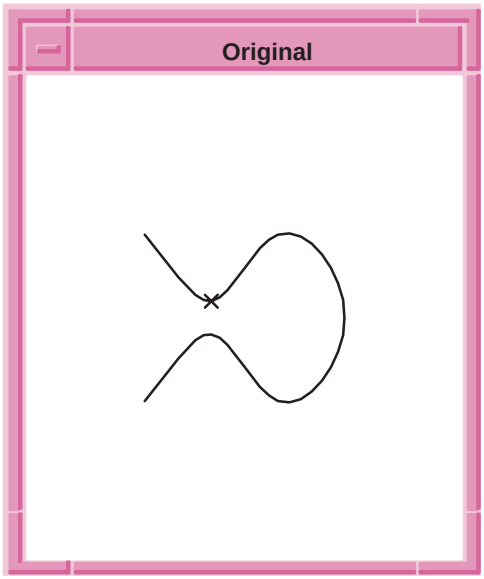


Figure 8-11. First Maximum Point on a Test Curve

The following example demonstrates how analysis is used to determine physical properties of a model. This example also illustrates transforms and Booleans.

The example creates and copies a cylinder. The first instance of the cylinder has a transform applied to it which rotates it 90 degrees. Then, the two cylinders are intersected leaving only the portion of each cylinder that is common to both. This new solid model can be queried for its physical properties, such as its surface area, volume, and center of gravity.

Scheme Example

```
; Purpose: to show that the volume of the intersection of
; two cylinders of unit radius, whose axes intersect at
; an angle of 90 degrees is 5 1/3.
(define my_cyl (solid:cylinder (position 0 0 –5)
    (position 0 0 5) 1))
;; my_cyl
; my_cyl => #[entity 2 1]
(define my_cyl_copy (entity:copy my_cyl))
;; my_cyl_copy
; my_cyl_copy => #[entity 3 1]
(define tr_90 (transform:rotation
    (position 0 0 0) (gvector 0 1 0) 90))
;; tr_90
(define my_tr_cyl (entity:transform my_cyl_copy tr_90))
;; my_tr_cyl
; my_tr_cyl => #[entity 3 1]

(view:refresh)
;; #[view 1076168656]
(define my_intersect (solid:intersect my_cyl my_cyl_copy))
;; my_intersect
; my_intersect => #[entity 2 1]
(solid:massprop my_intersect 0 0.00001)
;; (("volume" . 5.33333332728429)
;; ("accuracy achieved" . 1.21590583275609e–09))
(part:save "example1.sat")
;; ()
```

Some of the Scheme extensions related to analysis are:

gvector:length . . . . . Gets the length of a gvector.

gvector:parallel? . . . . . Determines if two gvectors are parallel.

gvector:perpendicular? . . . . . Determines if two gvectors are perpendicular.

law:derivative . . . . . Creates a law object that is the derivative of the given law with respect to the given variable.

law:eval . . . . . Evaluates a given law or law expression with the specified input and returns a real or list of reals.

law:eval–position . . . . . Evaluates a given law or law expression with the specified input and returns a position or a par–pos.

law:eval–vector . . . . . Evaluates a given law or law expression with the specified input and returns a gvector.

law:nderivative . . . . . Computes the numerical derivative of a law function with respect to a given variable, a given number of times.

law:nintegrate . . . . . Computes the numerical integral of the given law function over the specified range.

law:nmax . . . . . Computes where a law is the maximum over a given interval.

law:nmin . . . . . Computes where a law is the minimum over a given interval.

law:nroot . . . . . Computes all of the roots of a law function over a given interval.

law:nsolve . . . . . Computes all the locations where two laws are equal to one another over a given interval.

law:simplify . . . . . Creates a law which is the algebraic simplification of the given law.

par–pos:distance . . . . . Gets the 2D distance between two par–pos.

position:closest . . . . . Gets the position from a list of positions that is closest to a given position.

position:distance . . . . . Gets the distance between two positions.

position:project–to–line . . . . . Gets the projection of a position on to a line.

position:project–to–plane . . . . . Gets the projection of a position onto a plane.

solid:classify–position . . . . . Classifies a point with respect to a solid.