University of Edinburgh

Division of Informatics

Generating ACIS Models From SMS Models

4th Year Project Report

Computer Science

David McLennan

May 31, 2000

Abstract: The aim of this project is to design and implement a process that converts model object descriptions written in the SMS modelling language into CAD descriptions in the ACIS modelling language. The main conceptual problems involved are that we must derive ACIS descriptions from unrelated SMS descriptions, and that we must convert the notions of bound three dimensional space in SMS to actual geometry in ACIS. A successful conversion system was developed and tested, with a good degree of reliability and robustness. This dissertation discusses the current design of the conversion system and also suggests further work that could be carried out on this subject.

Contents

Chapter 1

Introduction

1.1 An Introduction

The principle goal of this project is to design and implement a process that converts object model descriptions that have been written in the SMS modelling language into descriptions suitable for inclusion into the ACIS modelling language.

The Suggestive Modelling System (hereby referred to as SMS) is an object representation language created by Robert Fisher and further developed by his research group at the division of Artificial Intelligence. It's main purpose and motivation is for object recognition, rather than object depiction, and thus it is optimised for representing the strongly visible features and relationships of non-polyhedral objects.

ACIS is an object orientated geometric modelling engine, created by Spatial Technologies, designed for use as the geometry foundation within three-dimensional (3D) modelling applications. Its use is common in the Computer Aided Design/Computer Aided Manufacture (CAD/CAM) industries, as well as the milling and aerospace industries. From its background, it is not surprising that ACIS is an object depiction language. This means that it attempts to describe an object exactly in every way, at the expense of description size and simplicity.

We wish to find a method of generating a mapping from SMS to ACIS such that an object modelled by SMS may be easily converted into a object that is in an ACIS representation. This is not necessarily a simply mapping, as there are subtle – yet fundamental deference's about the way in which SMS portrays its geometry and that way in which ACIS does.

One point to make is that we are primarily interested in converting the geometry of the scene depicted by SMS. There are other facts about the scene that SMS can describe, an example of this is the SMS Viewpoint dependent information which stores information that is very useful for an object recognition system, but does not effect the geometry of the scene. Indeed, ACIS may not have a natural method of storing this additional data, and we must consider whether it is relevant to our project goals.

1.2 Synopsis

This document follows the development of a system that converts model object descriptions written in the SMS modelling language into suitable descriptions for inclusion into the ACIS modelling language. It is split up into three main sections:

Background

Chapter 2 gives detailed background information on SMS, which assists the reader in understanding some of the tasks the conversion system must undergo, as well as giving insight into possible sticking points.

Chapter 3 gives detailed background information on ACIS. Unlike chapter 2 however, this chapter is not just purely background information, although the majority of it is so. Whilst introducing ACIS, we are already looking for similarities between ACIS and SMS, with a view about developing an approach that will enable the construction of the conversion system to begin.

The Conversion System

Chapter 4 covers both the development and the operation of the conversion system that is the core of this project.

Testing and Evaluation

Chapter 5 covers the tests that were run on the converter system in order to ascertain both its correctness and robustness.

Chapter 6 is our conclusion, which brings together the project as a whole, as well as suggesting some possible future work.

1.3 Acknowledgements

I would like to take this opportunity to thank the following people for their assistance in the execution of this project.

Bob Fisher – For the excellent advice, endless patience and debugging at 3 in the morning.

Naoufel Werghi – For reminding me what π meant.

Craig Robertson – For compiler advice, and introducing me to Mulg.

This project could not have happened without you all.

David McLennan 30/5/00.

Chapter 2

```
(LINE line20 LENGTH 20)

(Boundary our_surface_boundary

line20 at
TRANSLATION (-10,-10,0) ROTATION VECTOR (0,0,1) INTO (0,1,0)
SCALE1

line20 at
TRANSLATION (-10,10,0) ROTATION VECTOR (0,0,1) INTO (1,0,0)
SCALE1

line20 at
TRANSLATION (10, 10,0) ROTATION VECTOR (0,0,1) INTO (0,-1,0)
SCALE1

line20 at
TRANSLATION (10,-10,0) ROTATION VECTOR (0,0,1) INTO (-1,0,0)
SCALE1
)

(PLANE our_surface
BOUNDARY_LIST (our_surface_boundary AT ORIGIN SCALE 1)
INCLUDED POINT (0,0,0)
)
```

A Detailed View of the Suggestive Modelling System

2.1 An Introduction to the Suggestive Modelling System

The Suggestive Modelling System (SMS) is an object representation system, created by Robert Fisher, that is motivated by the requirements of object recognition. It is this requirement that is the driving force behind both the features that can be found in SMS, and the differences between SMS and more traditional means of describing geometric scenes.

The main difference to be found is that SMS is not an object depiction language. Most languages for describing 3D geometry are geared towards describing the objects in the scene with as high a degree of accuracy as possible, and thus tend to support complicated methods of describing arbitrary shapes. For example, these may include NURBS[1] based surfaces and a scale calibration system. Such languages are called object depiction languages, because their primary goal is accurate object depiction. Examples of object depiction languages include Autodesk's DXF, the open source PLY object file format, and of course ACIS SAT files. However when we look at the requirements of object recognition systems, we find that object depiction languages are not particularly suited for the purposes of object recognition. There are 3 main problem areas: -

- Object recognition is a difficult and computationally expensive process. Thus it is desirable to keep the scene as simple as possible, whilst retaining enough information to allow the recognition system to operate. Object depiction languages tend to have too much detail built into their models – is it really necessary to have that scratch on a Cola bottle modelled? The object recognition system will inform you that it is a Cola bottle – scratch or not. Excessive detail could ruin attempts for an object recognition system to run in real-time. Of course, object depiction languages do not *force* the use of high model detail, but at best they do force the object recognition system to cope with complicated methods of describing simple objects.

- Object depiction languages to not tend to decompose easily. In object recognition systems, scene comparison is a common task. Scene comparison methods require an efficient way for the scene to be broken down into units small and simple enough for easy comparison. Breaking up an arbitrary object in an object depiction system may be computationally expensive.

- Object depiction languages do not in general cover the more specialised aspects of object recognition. These include the ability to attach additional information about each object to the object in the scene, or viewpoint dependent information.

It was essentially these three problems that necessitated the development of the Suggestive Modelling System. It has the following features that separate it from object depiction languages, some of which resolve the problems discussed above.

- SMS is suggestive, rather than literal. Object depiction languages are literal – they attempt to describe a scene in an exact manner. Literal models are suitable for accurate image generation, suggestive models represent observable features without excessive numerical detail. Suggestiveness is of course required for generic model representation, otherwise rough matchability is not possible. However small losses of detail are acceptable,

[1] Non Uniform Rational B Splines

provided that an accurate overall description of an object is maintained. For example surface splines might represent a literal model of a Coca-Cola bottle. A suggestive model would build a Coca-Cola bottle out of cones and cylinders.

- SMS has a powerful subcomponent hierarchy. This allows rapid decomposition of a scene into simple geometric primitives. We shall delve into the structure of the SMS hierarchy in section 2.2

- SMS uses symbolic descriptions in that it allows definition of geometric models in a hierarchical manner, assigning symbolic names to features. Feature shape is separated from pose, in accordance with the output of scene segmentation techniques. In addition to this, SMS allows parameterisation of its models – variables and expressions may replace numerical values. We shall delve into this feature in section 2.7

- Additional object information may be attached to an SMS model or component through the use of properties. Such information could be numerical values calculated offline to assist in object recognition. Properties benefit from the SMS hierarchy by utilising it to calculate the domain of a property in the scene.

- SMS can also hold detailed viewpoint data. Information about feature or subcomponent visibility from several views may be included for inspection tasks or hypothesis verification. Such information may again be computed offline and attached to the model for quick reference.

These features combine to solve the shortcomings of object depiction languages, providing support for many object recognition techniques. It satisfies the needs for model invocation, model matching and reference frame estimation, without the excess baggage and complexity of an object depiction system. These properties are desirable for the use with object recognition systems, since they are interested in understanding the overall object scene, rather than fine object detail. Thus SMS is optimised to represent strongly visible features and relationships of non-polyhedral objects. It does this by integrating curve, surface and volumetric descriptions within a flexible subcomponent hierarchy. As such, should an object recognition system wish to express a scene it perceives in an exact manner, an SMS model would be a good choice.

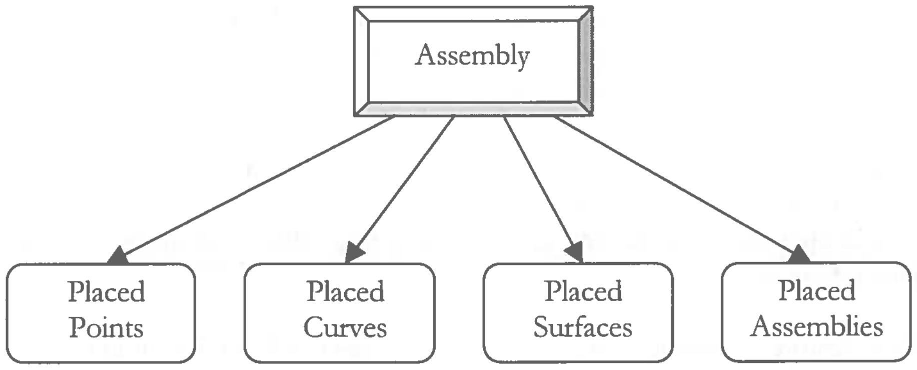
Now that we understand the aims behind SMS and the tasks it supports, let us delve a little further into what specific parts of SMS we are interested in to achieve our goal of conversion, and have a closer look at the hierarchy and syntax of SMS.

2.1.2 What parts of SMS we need for our conversion

Recall from the project introduction that the project aim is the conversion of geometric models from SMS to ACIS. Because it is the geometry that we are interested in converting, there are some features in SMS that are irrelevant when you consider our goal. (For example, viewpoint information is redundant – it is for the purposes of object recognition only, the geometry of the scene is not affected by it.) For this reason, we need not concern ourselves with viewpoint information, properties or volumes. I may mention them for the sake of completeness – but our primary focus is on the methods SMS utilises to describe the geometry of a scene and the hierarchy of SMS. Now let us take a look at the structure of the SMS hierarchy.

2.2 The SMS Model Hierarchy

The SMS hierarchy forms a tree structure, and can have multiple levels. The top level object is called the assembly. In here are all the references to the predefined surfaces, as well as global translation, rotation and scale values. It is in the assembly level that the final SMS model comes together, (hence the name), by translating and rotating the predefined surfaces until the required object is built. One point to note is that a surface can be used by the assembly stage more than once. For example, a cube could be created by referencing the same square plane surface 6 times, but with different transitional and rotational values. An assembly can even reference other assemblies. For example, if there pre-existed SMS models of objects which were required in another scene, a user could easily build a new SMS model by including each SMS object model assembly in the assembly of the new SMS scene, with additional transitional and rotational values for scene layout. To avoid confusion, such assemblies are called subassemblies. A graphical overview of the top level SMS hierarchy can be found below.



There may be zero or more numbers of points, curves, surfaces or assemblies. There are also further subtypes of curves and surfaces. For curves, we can have:-

- Lines
- Circular Arcs
- Elliptical Arcs
- Parabolic Arcs
- Hyberbolic Arcs

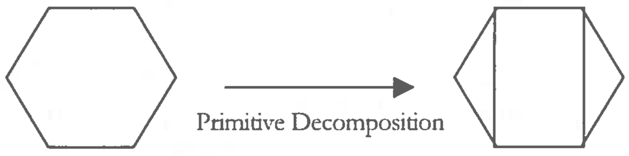
For surfaces, SMS supports:-

- Planes
- Cylinders
- Cylindrical Patches
- Ellipsoids
- Cones
- Tori
- Doubly Curved Patches

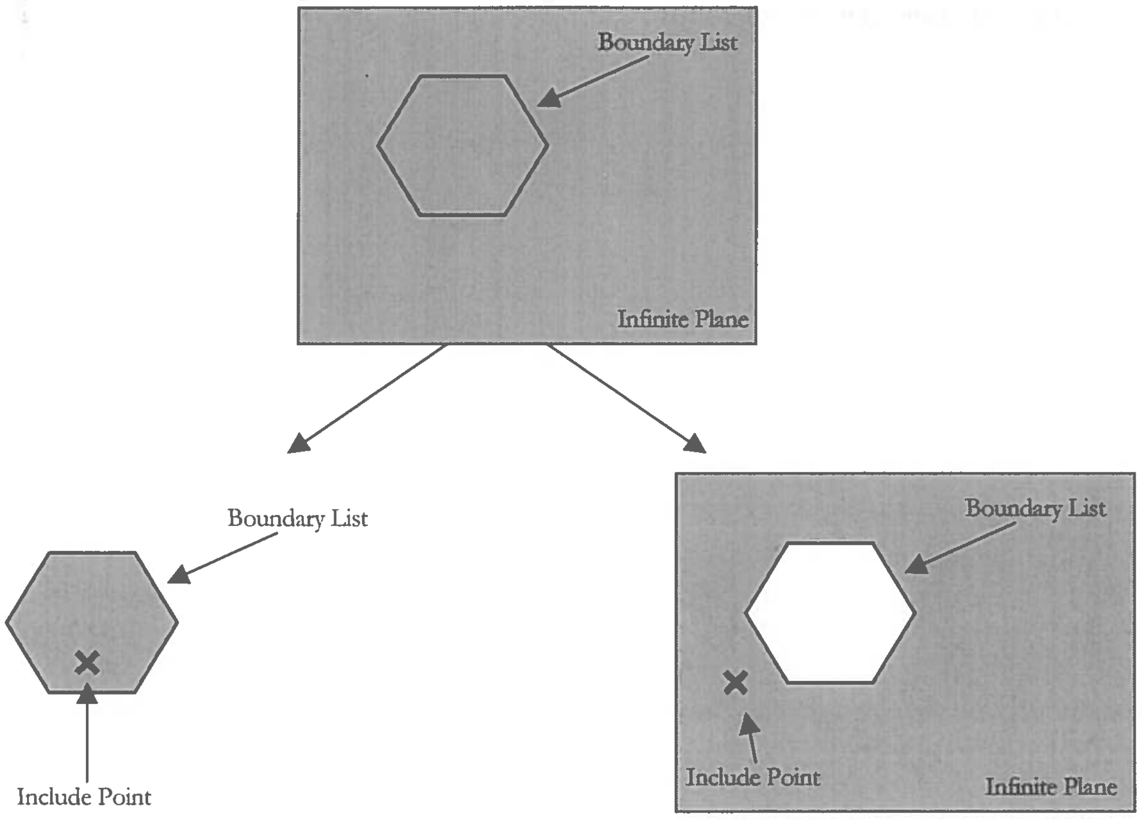
We shall have a closer look at these geometric primitives, including acceptable grammar, in section 2.4. However, before we can proceed to that we need to understand how SMS gives shape to its geometric primitives – it does this through a boundary representation system. This system is the key to complex shapes in SMS, and as we shall find out, the biggest hurdle to the conversion process.

2.3 The SMS Boundary Representation System

Surface primitives are all fine and well, but it can be difficult to create complex shapes with them. For example even a trivial convex shape such as the hexagon below requires multiple primitives for its construction. The result can be a very high number of primitives required to represent even a simple shape, and complex shapes may prove impossible to build.



SMS avoids this problem through the use of *boundaries* to control its surface primitives. For example, say we wanted to construct a 2D plane in the shape of a hexagon. Rather than having to decompose the hexagon into multiple parts (as above), or including a hexagon primitive in SMS, we declare an infinite plane, and then construct a *boundary list* out of curves that declares the point at which the plane ends. (In our example we use straight curves, i.e. lines!) Once a boundary list has been created, an include point is required to specify whether the boundary list represents the outer limit of the plane, or a hole in it. For an illustrated example, consider the diagram below. The grey area represents our plane primitive, which is infinite in X and Y. A boundary list consisting of 6 curves forming a hexagon is declared, and is translated to our desired location. Finally, the position of the include point decides which side of the boundary the plane is now valid in – the inside or the outside. If the include point is inside the boundary list, then the plane now only exists inside the boundary, and we have a plane in the shape of a hexagon. If the include point is on the outside, then the plane is valid everywhere *but* inside the boundary. In effect, we have punched a hexagon shaped hole in the infinite plane.



Although this is a simple example, this is how all the objects in SMS are constructed. Where appropriate, SMS geometric primitives are infinite in nature (Planes, Cylinders, Cylindrical Patches, Cones and Doubly Curved Patches), and those that are naturally bound (Ellipsoids and Touri) can still have holes punched in them via the use of boundaries. For example, consider the construction of a hemisphere of radius R. Here we would declare a spherical ellipsoid of radius R, along with a boundary list consisting of a circular arc with length 2π radians and radius R. The boundary list would be placed so that it cuts the ellipsoid in two, with the include point deciding which half we would retain.

As the term Boundary List implies, a single geometric primitive can have multiple boundaries. Thus highly complex shapes may be built from a single primitive, with multiple holes and segments. One point to note is that in SMS, boundaries may only be curves - you cannot use a pre existing surface to define a boundary. Is this a problem? The answer is no, since SMS is a surface, rather than solid modeller. It is interested only in punching holes in surfaces - what goes on inside or behind a surface is irrelevant. However, should SMS concern itself with volumes in the future (indeed, first and second order volumetric features are two SMS extensions) having surfaces as boundaries would make an interesting extension – akin to Boolean operation tools in a solid modeller.

To conclude, the SMS boundary representation system allows us to construct complex shapes from a relatively limited list of geometric primitives. Whilst area or volume primitives might initially seem to be simpler, they suffer from either forced decomposition, or force us to support primitives for every possible polygon shape. (There are a lot!) The boundary representation method for describing polygons is efficient, effective and flexible.

Now that we understand both the generic structure of SMS, and how boundaries can be utilised to customise our primitives to the desired shape, we can dive into the specific details of the geometric primitives. We shall examine each primitive, looking at its properties, and its lexical and grammatical forms.

2.4 The SMS Geometric Primitives

The SMS Geometric primitives fall into two categories, curves and surfaces. Curves are normally used to form boundaries for the surfaces, although their use by themselves in an SMS scene is valid. Nevertheless, surfaces tend to form the majority of an SMS model.

Just before we delve into the specifics, I should note the fact that SMS has multiple methods of specifying rotations. This is partly for flexibility – some specifications are more convenient to use in certain situations than others, and partly because the requirements on the rotational values can be different from primitive to primitive. For example, one rotation option is to give a *vector* (x_1,x_2,x_3) into (x'_1,x'_2,x'_3), such that x_i transforms onto x'_i. This is quite convenient, except that the rotation is partly unconstrained – it does not allow for rotation about the axis of the vector. This is no problem for curves, but for other situations it may be. These rotation methods are discussed in detail in section 2.6. For now, we shall use the marker <Rotation Spec> to denote an arbitrary rotation method for the geometric primitives. In reality, this will decompose into an actual rotation specification, as detailed in section 2.6.

2.4.1 Curves

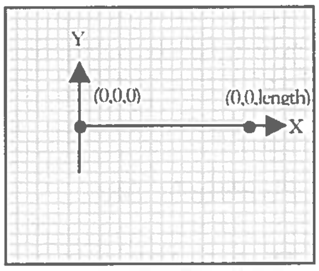
A curve definition requires at least a name and the curve's parameters. There are 5 curve types – each is discussed in detail below.

Line

A LINE is declared by supplying a name and its length:

```
(LINE name LENGTH length)
```

Where name is an arbitrary string, and length is an integer. This declares a line along the positive z axis, with endpoints (0,0,0) and (0,0,length).

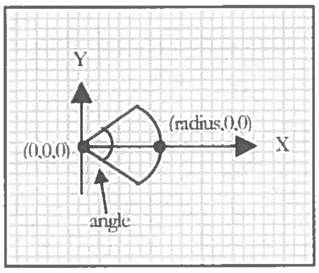


Circular Arc

A circular arc is specified by a name, its radius, and the angle subtended by the ends of the arc.

```
(CIRC_ARC name RADIUS radius ANGLE angle)
```

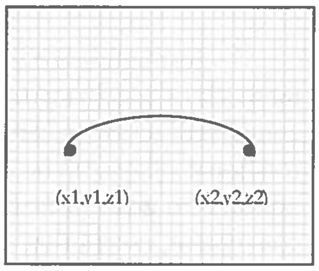
This declares an arc in the X-Y plane whose centre is at the origin and with the curve's midpoint at (radius,0,0). As with most of the remaining features, an arc must be placed with a fully constrained rotation -- any of the forms described in section 2.6 will suffice.



Elliptical Arc

An elliptical arc allows the selection of any section of an ellipse. The syntax expects the **X** radius, **Y** radius, and two endpoints.

```
(ELLIPSE name XRADIUS x YRADIUS y
ENDPOINTS(x1,y1,z1) (x2,y2,z2))
```

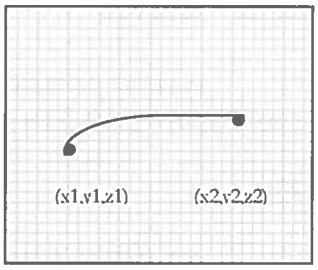


This defines an ellipse in the X-Y plane centred at the model origin, with radii x and y along the two co-ordinate axes. The section of interest is taken clockwise from the first to second endpoints. Making both endpoints equal specifies a complete ellipse. If the complete ellipse is in fact a circle, then, because it is rotationally symmetric, it should be placed using the <vector into> notation.

Parabolic Arc

An arc may be extracted from the parabola x = rate * y^2, by supplying rate and two endpoints.

```
(PARABOLA name RATE rate ENDPOINTS (x1,y1,z1)
(x2,y2,z2))
```



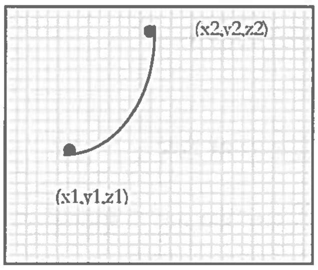
This generates a finite section of the parabola in the x-y plane which falls between the two endpoints.

Hyperbolic Arc

The hyberbolic arc is extracted from the hyperbola

$x = \sqrt{yrate * y + yoffset}$, In the form:

```
(HYBERBOLA name RATE yrate OFFSET offset ENDPOINTS
(x1,y1,z1) (x2,y2,z2))
```



This refers to the finite section of the hyperbola in the x-y plane.

2.4.2 Surfaces

As already discussed, surfaces in SMS have a shape and a boundary list. Aside from the practical advantages of using boundaries, this mirrors the action of vision programs which may first find the generic shape of a surface patch and then later attempt to match the surface's boundaries, so the separation also has object recognition advantages. SMS allows the specification of surfaces with no boundaries for cases where a system matches only on infinite primitives. For each surface, the boundary list follows the following pattern: a boundary name, translation, rotation and scale. Multiple boundaries can be used, but the surface must be simple and connected – although the SMS compiler does not force this. The SMS surface primitives are:

Plane

The plane is the simplest surface patch, requiring no parameters. A plane is defined in the x-y plane, as follows:

```
(PLANE name
       BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
INCLUDED_POINT (x0,y0,z0))
```

The default normal of the plane is pointing down the negative Z axis, so care must be taken when rotating a plane into place in the assembly – otherwise back face culling may result in disappointment!

Cylinder

This creates in infinite elliptical cylinder as defined by supplying radii in the Y and Z directions. (I.e. The cylinder itself lies along the x-axis)

```
(CYLINDER name YRADIUS y_radius ZRADIUS z_radius
 BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
 INCLUDED_POINT (x,y,z))
```

Cylindrical Patch

A cylindrical patch is defined in exactly the same ways as a cylinder – for the exception of the keyword `CYLPATCH` instead of `CYLINDER`. The difference between the two is the position of their axes. A cylinder lies along the X axis, whilst the cylindrical patch's axis is offset by `z_radius`, meaning that its surface touches the X axis.

Ellipsoid

An ellipsoid is specified by X,Y and Z radii, plus the obligatory boundary list. Since the Ellipsoid is naturally bound by the radii parameters, boundary lists are confined for the purposes of subtraction. A sphere would be represented by an ellipsoid with equal X,Y and Z radii.

```
(ELLIPSOID name XRADIUS x_radius YRADIUS y_radius ZRADIUS z_radius
 BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
INCLUDED_POINT (x,y,z))
```

Cone

A cone is specified by giving a radius rate, which is the tangent of the cone angle, and a boundary list.

```
(CONE name RADIUS_RATE tan_alpha
 BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
INCLUDED_POINT (x,y,z))
```

Torus

The torus is specified by a major radii (outer ring) and a minor radii (inner ring).

```
(TORUS name MAJOR_RADIUS major_rad MINOR_RADIUS minor_rad
 BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
 INCLUDED_POINT (x,y,z))
```

The axis of symmetry of the torus is coincident with the X axis of the local frame.

Doubly Curved Patch

Known by its keyword TWOPATCH, this is an infinite doubly-curved surface, with its curvature axes residing at the origin, aligned with the X and Y axes. This representation is designed to be an approximation to real doubly curved patches, whose curvatures will vary everywhere on the patch. A good way of visualising a TWOPATCH is to imagine a horse saddle, it curves front to back as well as left to right.

```
(TWOPATCH name XRADIUS x_radius YRADIUS y_radius
 BOUNDARY_LIST (
       boundary_1 AT TRANSLATION (x,y,z) ROTATION <rotation spec> SCALE scale
       .
       .
       boundary_n AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale)
 INCLUDED_POINT (x,y,z))
```

2.4.3 The Assembly

We have already come across the SMS assembly in the hierarchical section. There, we stated that the assembly was the root of the hierarchy of SMS. Here, we delve into the actual lexical form of the composite feature root of SMS.

An assembly consists of zero or more (feature, scale, translation and rotation) quadruplets, which together constitute the actual geometrical model. The quadruplets have the following generic form:

```
Feature AT TRANSLATION (x,y,z) ROTATION <rotation-spec> SCALE scale
```

Where <rotation-spec> is a rotational specification, as defined in section 2.6.

The assembly syntax allows different representations of the object to be included in the model – at most one for each of the basic feature types. Hence a model may have point, curve, surface and volume based descriptions in the same SMS model. Representation types that are not required are simply omitted. The full assembly syntax is as follows:

```
(ASSEMBLY name
          VARS (var-name-1 (DEFAULT_VALUE var-value-1) ... )
           PLACED_POINTS <placed-feature-list>
           PLACED_CURVES <placed-feature-list>
           PLACED_BOUNDARIES <placed-feature-list>
           PLACED_SURFACES <placed-feature-list>
           PLACED_VOLUMES <placed-feature-list>
           PLACED_ASSEMBLIES <placed-feature-list>
           VDFG_LIST NONE
           DEFAULT_POSITION AT TRANSLATION (x,y,z) ROTATION <rotation-spec>
           PROPERTIES NONE) // Potential user properties
```

Remember that each of the above lines is optional – with the exception of the assembly line of course! The <placed-feature-list>s refer to one or more placed features, a bit like a boundary list, but of features instead. Below is an example of a surface placed feature list.

```
PLACED_SURFACES
           feature_1 AT TRANSLATION (0,0,0) ROTATION RST (0,PI,0) SCALE 1
                                 .
                                 .
           feature_n AT TRANSLATION (0,0,0) ROTATION RST (0,PI,0) SCALE 1
```

Also note that you can call further assemblies from here – this can be very useful, for instance clusters of objects can be created by referencing a base assembly multiple times.

2.5 SMS Reference Frame Transformations

You may have noticed that transformations of various forms can exist within the SMS hierarchy. The boundary elements have transformations, the features can have transformations, and even the assembly contains a "master" transformation. (The Default Position) An important point to note is that all these transformations aggregate together, starting with the assembly transformation, to form the "final" translation, rotation and scale values of every feature. This process is known as *concatenation*, and is useful since we may control the translation and rotation of objects at any point in the hierarchy, and the values placed will affect every feature within the domain of our chosen feature.

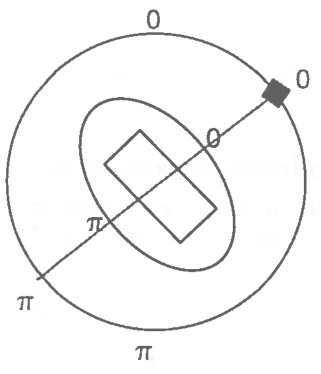
2.6 SMS Rotation Specifications

As was mentioned in the introduction to the SMS geometric primitives, SMS has a variety of ways of expressing rotations. Part of the reason for doing this is for flexibility. SMS is very much a working language, designed to be the "glue" between various systems involved in the task of object recognition. These systems may have different means of expressing rotations since there is no guarantee that they were either designed by the same teams, or that they somehow agree on what form of rotational specification to use. The second reason for the differing rotations is that the requirements of the rotation differ from primitive to primitive. As was mentioned before, it is safe for a line to have an unspecified rotation around its local axis (imagine the line rotating around its direction vector). However "stricter" rotations are required for some curves and most surfaces. Thus SMS allows a number of different rotational styles, placing responsibility on the user to decide which is the best for a particular situation. The specifications are as follows:

RST – Rotation, Slant and Tilt

```
RST (r,s,t)
```

The rotation value here defines planar (X-Y) rotation in radians. Slant and tilt work together to define a Z rotation. Imagine tilt as the "target" and slant as the rotation in the plane of that target. The diagram below should make things a bit clearer.



Rotation
Slant (about π/4 here)
Tilt (about π/3 here)

Quaternion

QUATERNION (q_0, q_1, q_2, q_3)

Standard quaternion notation – rotation by θ about axis W. (Cos θ/2, Sin θ/2 W)

Rotation Matrix

MATRIX (m_{11},m_{12},m_{13})
(m_{21},m_{22},m_{23})
(m_{31},m_{32},m_{33})

Standard Euler transformation matrix.

Axis

```
AXIS (axis₁ θ₁)
     (axis₂ θ₂)
     (axis₃ θ₃)
```

This specifies a rotation about the three coordinate axes. *Axis$_i$* is one of X,Y or Z.

Vector Pair

```
VECTOR_PAIR (x₁,y₁,z₁) INTO (x'₁,y'₁,z'₁), (x₂,y₂,z₂) INTO (x'₂,y'₂,z'₂)
```

This specifies a rotation such that vector V_1 becomes V'_1, and V_2 becomes V'_2.

Vector

```
VECTOR (x,y,z) INTO (x',y',z')
```

This specifies an unconstrained rotation mapping x onto x'.

Unconstrained

```
UNCONSTRAINED
```

Completely unconstrained rotation – used where rotation does not really matter or is irrelevant. For example, a sphere's rotation is unconstrained – it will look the same whatever the actual rotation values placed upon it.

Together, all these rotational specification options provide a very wide choice to the user, allowing them to utilise the specification that matches their particular needs the best.

2.7 Parameterisation and Expressions in SMS

Recall from the chapter introduction that SMS allows the use of expressions. Expressions may be placed as an alternative to any scalar parameter in SMS. This can come in useful – a simple example is that it avoids excess numerical baggage when dealing in radians, since we may use expressions involving π do denote values in radians exactly, rather that decimal approximations. The expressions in SMS are C like in their grammatical and lexical forms, and include all fundamental mathematical calculations, as well as various sine and vector operations. Their details may be found in the table below – note that (factor) is any expression that evaluates to a numerical value, and that nested parenthesises are supported.

Expression	Mathematical Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
PI	Value of π.
ABS (factor)	Absolute Value
COS (factor)	Cosine (Angle in Radians)
SIN (factor)	Sine (Angle in Radians)
TAN (factor)	Tangent (Angle in Radians)
ACOS (factor)	Inverse Cosine (Radians)
ASIN (factor)	Inverse Sine (Radians)
ATAN (factor)	Arctangent (Radians)
SQRT (factor)	Square Root
LOG (factor)	Natural Logarithm
EXP (factor)	e^x
MIN ((factor), (factor))	Scalar Minimum
MAX ((factor), (factor))	Scalar Maximum
DOTPR ((factor), (factor))	Vector Dot Product
CROSSPR ((factor), (factor))	Vector Cross Product

One point to note is that the Vector Product operations obviously require vectors as parameters. Thus (factor) for them represents a parenthesised triple of float expressions.

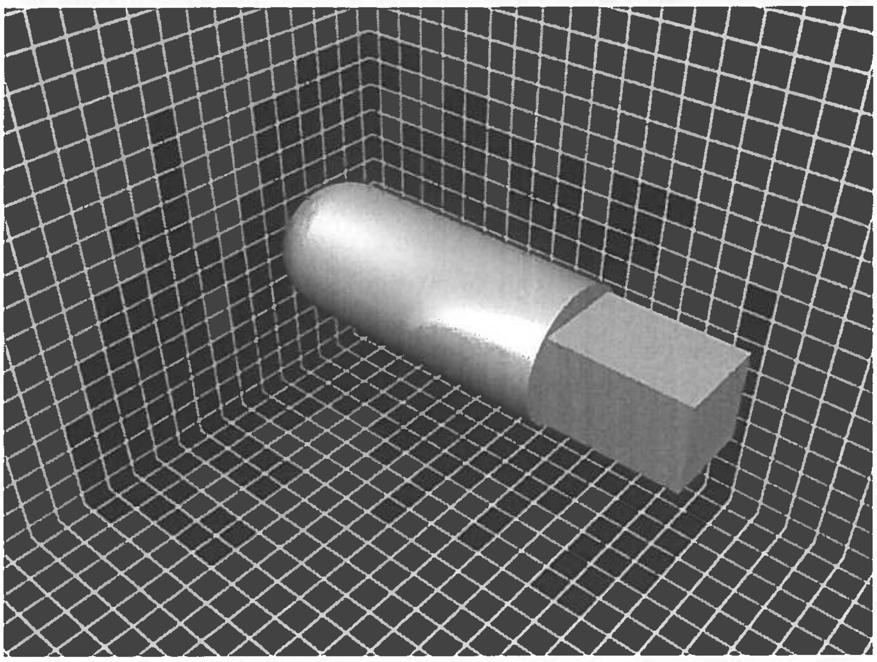
By this point, we have covered all the basic features of SMS that are necessary to understand in order to begin thinking about what is required of the conversion process to ACIS – we shall have a chat about this in the chapter conclusion. We have covered:

- The SMS Hierarchy
- The SMS Boundary System
- The geometric primitives of SMS – curves, surfaces and the assembly
- The rotation and transformation specifications of SMS
- Allowable Expressions in SMS

Let us promote understanding of these features by utilising them all in a real world example. In the next section, we shall build a simple mechanical part in SMS that makes use of all these features.

2.8 Manually Building a Simple Geometric Model in SMS

Here we are going to build an SMS model of an atypical industrial object. The shape itself is relatively simple as you can see from the three dimensional picture of it below.



It consists of a cuboid, attached via a plane to a cylinder, which is then terminated with a hemisphere. How do we go about building a model of this in SMS? We have already taken a step towards our solution – we have decomposed the model into simple parts. The next step is to see if we can decompose it further, which of course we can. Allowing for each part to have its own translation and rotation values we can decompose the model as follows

Original Shape   Parts that can be expressed as primitives in SMS

$$\begin{tikzcd}
 & \text{Hemisphere} \arrow[r] & \text{Half a Sphere} \\
 & & \text{Capped Cylinder} \\
\text{Aircraft Part} \arrow[ruu] \arrow[r] \arrow[rddd] & \text{Cylinder} \arrow[ru] \arrow[rd] & \\
 & & \text{Endcap (surface between the cylinder and the cuboid)} \\
 & & \text{4 x Rectangle} \\
 & \text{Cuboid} \arrow[ru] \arrow[rd] & \\
 & & \text{Square Endcap}
\end{tikzcd}$$

Once we have decomposed the model into parts that can be expressed as primitives in SMS, we can begin to build our model. Our second step is to think about how we need to modify the SMS primitives to fit our requirements, via the use of boundaries. For the half a sphere, we need to cut an SMS ellipse in half, for the cylinder we need to cap its length (recall that the cylinder is infinite in SMS), and for the rectangles we need to form boundaries to limit their size to that required. With these plans in mind, let us proceed to the building of the model.

Building the hemisphere

As mentioned, in order to build our hemisphere we need to cut an ellipsoid in two via the use of a boundary. For the boundary we will require a circular arc that has the same radius as the sphere, and a length of 2π (for a closed circle). We express this in SMS as:

```
(CIRC_ARC circle30 RADIUS 15 ANGLE 2*PI)
```

For the boundary, we just need to include our "chopping" circle with the correct rotation and translation values. For the translation, it will just be (0,0,0) since the ellipsoid will be initially centred at the origin, and for the rotation, we need to rotate its default orientation in the X-Y plane to the Y-Z plane, since we wish to cut it in half along the X axis. In order to do that we need to rotate the circular arc by 90 degrees, or π/2 Radians. Hence:

```
(BOUNDARY boundsphere1
          circle30 AT TRANSLATION (0,0,0) ROTATION RST (0,PI/2,0)
          SCALE 1)
```

Now that we have defined the boundary, we can call the SMS ellipsoid, together with an include point which picks what half of our ellipse we wish to retain.

```
(ELLIPSOID sphere
            XRADIUS 15
            YRADIUS 15
            ZRADIUS 15
            BOUNDARY_LIST (boundsphere1 AT ORIGIN SCALE 1)
            INCLUDED_POINT (15,0,0))
```

That's it!

Building the cylinder

For the cylinder, we require the SMS cylinder primitive, to together with a boundary that will cap its ends where we wish them to end. For the boundary, we can utilise the same Circular Arc that we created for the hemisphere, since the cylinder has the same radius. The boundary we use is as follows:

```
(BOUNDARY boundcylinder1

            circle30 AT TRANSLATION (30,0,0) ROTATION RST (0,PI/2,0)
            SCALE 1

            circle30 AT TRANSLATION (-30,0,0) ROTATION RST (0,PI/2,0)
            SCALE 1)
```

Now all that we need to do is to make as call to the SMS cylinder primitive with our newly defined boundary and include point.

```
(CYLINDER cylinder1
            YRADIUS 15
            ZRADIUS 15
            BOUNDARY_LIST (boundcylinder1 AT ORIGIN SCALE 1)
            INCLUDED_POINT (0,0,-15) )
```

Building the Cuboid

To construct the cuboid, we need to construct 5 planes. One for the end of the cuboid (which is square), and 4 for the sides of the cuboid (which are identical rectangles). However, recall from out discussion about assemblies that we may call a surface more than once, so in reality we only need to create 2 planes, the square and the rectangle. In order to create these we will have to declare two new lines, of different lengths, in order to construct the boundaries. The lines are:

```
(LINE line30 LENGTH 30)
(LINE line20 LENGTH 20)
```

For the square plane, we construct the following boundary:

```
(BOUNDARY bound1
      line20 AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1) INTO (0,1,0)
      SCALE 1

      line20 AT TRANSLATION (0,20,0) ROTATION VECTOR (0,0,1) INTO
      (1,0,0) SCALE 1

      line20 AT TRANSLATION (20,20,0) ROTATION VECTOR (0,0,1) INTO
      (0,-1,0) SCALE 1

      line20 AT TRANSLATION (20,0,0) ROTATION VECTOR (0,0,1) INTO
      (-1,0,0) SCALE 1 )
```

Then utilise this boundary to cut out the square plane that we desire:

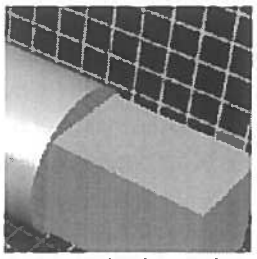
```
(PLANE face1
      BOUNDARY_LIST (bound1 AT ORIGIN SCALE 1)
      INCLUDED_POINT (10,10,0)
)
```

For the rectangle, we perform similar operations, but with slightly different values – eg. We replace line20 with line30 in some instances. Its boundary and declaration may be found below.

```
(BOUNDARY bound2
      line20 AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1) INTO
      (0,1,0) SCALE 1

      line30 AT TRANSLATION (0,20,0) ROTATION VECTOR (0,0,1) INTO
      (1,0,0) SCALE 1

      line20 AT TRANSLATION (30,20,0) ROTATION VECTOR (0,0,1) INTO
      (0,-1,0) SCALE 1

      line30 AT TRANSLATION (30,0,0) ROTATION VECTOR (0,0,1) INTO
      (-1,0,0) SCALE 1)

(PLANE face2
      BOUNDARY_LIST (bound2 AT ORIGIN SCALE 1)
      INCLUDED_POINT (10,10,0) )
```

Building the Cylinder to Cuboid Interface

What are we talking about here? It's the part that is placed between the cylinder and the cuboid in our object. (See picture to the right – it's marked with red lines.) It consists of a circular plane that has the same radius as the cylinder, with a square hole the same size as the cuboid cut out from it that the cuboid fits into. We can represent this through the use of a double boundary, one to bound a plane into a disc that has the same radius as the sphere, another to cut the square hole out of it. For the first boundary, we can again use the circular arc that we declared for the hemisphere. For the second, we can be a little more cunning – we can reuse the boundary we created for the end of the cuboid, since it has the same shape and size. The first boundary that gives us a disc is as follows:



```
(BOUNDARY cuboid_interface
      circle30 AT TRANSLATION (0,0,0) ROTATION RST (0,0,0)
      SCALE 1)
```

The plane declaration has the form:-

```
(PLANE face3
      BOUNDARY_LIST (
      bound1 AT TRANSLATION(-10,-10,0) ROTATION RST (0,0,0) SCALE 1
      cuboid_interface AT ORIGIN SCALE 1)
      INCLUDED_POINT (14,0,0))
```

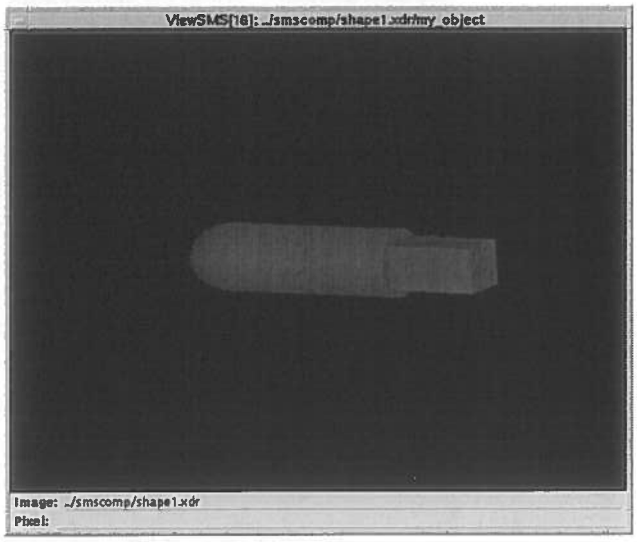
Note the utilisation of the existing square boundary bound1 in the boundary list. The include point is carefully chosen to place it in the section we wish to retain – between the disc and the square cutout.

Bringing it all together – The Assembly

We have now created all the SMS primitives that are necessary for the objects description, all we need to do now is build the model by calling them from the assembly. Recall that all of our primitives were created centred at the origin – thus we will have to transform them in the assembly to ensure their correct position. The assembly can be found below, with the SMS code in blue, and my comments in red.

```
(ASSEMBLY our_object
PLACED_SURFACES    // Start placing our surfaces
// Build our cuboid
face1 AT TRANSLATION (30,-10,-10) ROTATION RST (0,0,0) SCALE 1
face1 AT TRANSLATION (30,-10,10) ROTATION RST (0,0,0) SCALE 1
face1 AT TRANSLATION (30,10,-10) ROTATION RST (0,PI/2,PI/2) SCALE 1
face1 AT TRANSLATION (30,-10,10) ROTATION RST (0,-PI/2,PI/2) SCALE 1
face2 AT TRANSLATION (60,-10,-10) ROTATION RST (0,PI/2,0) SCALE 1

// Place the interface between the cuboid and the cylinder
face3 AT TRANSLATION (30,0,0) ROTATION RST (0,PI/2,0) SCALE 1

// Call the cylinder
cylinder1 AT TRANSLATION (0,0,0) ROTATION RST (0,0,0) SCALE 1

// And finally the hemisphere
sphere AT TRANSLATION (-30,0,0) ROTATION RST (0,0,0) SCALE 1
VDFG_LIST NONE // No VDFG list (Viewpoint info)

// Set a global transformation for the entire scene
DEFAULT_POSITION AT (0,0,0) ROTATION RST (0,0,0)

// No user properties
PROPERTIES NONE)
```

That's it! This may be a very simple model, but it makes use of most of the common features in SMS, and is a typical example of the kind of model we wish to convert. A ViewSMS rendering of the model we have just built can be found to the right. (ViewSMS is the SMS renderer.)


ViewSMS[18]: ../smscomp/shape1.xdr/my_object
Image: ../smscomp/shape1.xdr
Pixel:

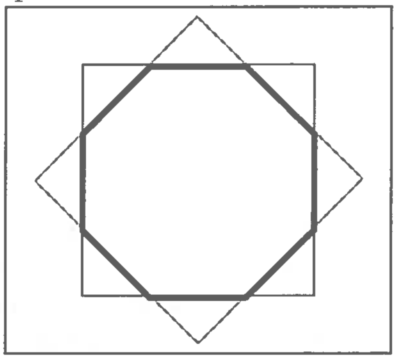
2.9 Some problems with SMS?

SMS was written to be highly flexible – recall the decisions to use boundaries and the number of different rotational specifications. These features are part of the strength of SMS, but they may come at a price.

Potential Problems with Boundaries

SMS places a lot of responsibility on the user for correctness. One of the responsibilities for the user is to ensure that all boundaries are closed. What that means is that all boundaries should form a cycle – a closed region in 2D space. However, objects that do not have closed boundaries will still compile, as the responsibility for correctness is placed solely on the user. Since (as you will find out soon) our conversion effort uses the SMS compiler for error checking, we are forced to take one of two choices. We either keep the responsibility for the models correctness on the user, or we spend effort in providing error checking as part of the conversion process. This decision will be covered in Chapter 4, but my point is that I feel this feature in SMS is undesirable. I feel it is a failure in the semantic checking during compilation and it is something that should be checked at compile time. In reality, the reason this ambiguity exists is twofold. Firstly, SMS is very much a working language – it is still being developed and tuned for specific image recognition tasks to this day. The lack of boundary checking is probably just an oversight – after all the inclusion of boundary semantic checking would be relatively simple at compile time. Secondly this oversight has probably been allowed to continue because the average user of SMS is very experienced – thus it is perhaps forgivable to place more responsibility on them than the average 'C' compiler might on its users. Nevertheless this ambiguity does pose a problem for our conversion task.

Another potential problem is that it is currently perfectly legal for overlapping boundaries to exist within SMS. For example, the "normal" method for constructing an octagonal patch would be the creation of a boundary using 8 lines, and the application of that boundary onto a plane. There is another method however. We could create a square boundary from 4 lines. We can then apply this boundary to a plane – not once but *twice*. The first would obviously create a square, the second would be applied as well – but at an angle of 45 degrees to the first. (See diagram to the right – 1st boundary in blue, 2nd in red, resulting shape – black.) I do not consider this a design flaw in SMS, but it does open up some paths to abuse. If one were to take this to the extreme, one could create N-gons with a very high N, simply by applying the same square boundary N times with an extra rotation of 360/N each time. This feature of overlapping boundaries forces the conversion process to carry out various operations it calculate the final shape of a surface, rather than just naively take the boundaries as the final shape. However, to be fair this is more an issue that we must deal with during our conversion, rather than any fatal flaw in SMS. There is more about this issue in Chapter 4.



Potential Problems with Rotations

Another place where SMS places responsibility on the user is to use the correct rotation specification. Take the following (incorrect) line in an assembly as an example:-

```
face1 AT TRANSLATION (30,-10,-10) ROTATION UNCONSTRAINED SCALE 1
```

This is of course, incorrect if face 1 is anything but a sphere or point. However even if it is a plane it will still compile. Is this wrong? This is a hard question to answer. It is of course undesirable for an object that requires rotational values to have none and still be considered valid by the compiler – but SMS does state that "valid" rotational values must be used. Again I feel that we are witnessing a lack of semantic checking within the SMS compiler – something that is unfortunate for us but not a design flaw in SMS. As we shall see in chapter 4, this issue is resolved by following the compiler's example – which is to assume that all objects have a default rotation of zero, unless otherwise stated. Thus if a feature has an invalid rotation, then it simply has a rotation of zero.

2.95 A Conclusion on SMS

By now we have covered all the foundations of SMS and all the features that are necessary to understand in order to carry out our project aim of conversion of the geometry within SMS. With this knowledge, we can start to think about what might be required of our conversion system. Certainly, an object depiction language such as ACIS should have no problem depiction any geometry that SMS can understand – most even have similar primitives to those used by SMS. However, it is the way that SMS portrays object shape via the use of boundaries that will probably require most thought.

To conclude, the Suggestive Modelling System is a highly optimised object recognition support mechanism, that has a number of unique features for the purposes of object recognition. It offers good compatibility with scene segmentation techniques via its use of a powerful sub-component hierarchy, and has the capability for user extension via the use of surface properties. As such, should an object recognition system wish to express a scene it perceives in an exact manner, an SMS model would be a good choice.

Chapter 3

A Detailed View of the ACIS Modelling Engine

3.1 An Introduction to the ACIS Modelling Engine

The ACIS 3D Toolkit (ACIS) is an object-oriented three-dimensional geometric modelling engine from Spatial Technology Inc. (Spatial). It is designed for use as the geometry foundation within virtually any end user 3D modelling application. Written in C++, ACIS provides an open architecture framework for wireframe, surface, and solid modelling from a common, unified data structure. It does this by providing a set of C++ classes and functions to create an end user 3D application. A good way to image this is to imagine ACIS as a kernel, to which applications can make calls to for geometry processing and rendering. 3rd party or specialised functionality may be incorporated via "husks" that sit between applications and the ACIS kernel. ACIS integrates wireframe, surface, and solid modelling by allowing these alternative representations to coexist in a unified data structure, which is implemented in a hierarchy of C++ classes. ACIS bodies can have any of these forms or combinations of them.

ACIS has become popular in the CAD/CAM industry due to the fact that it allows companies to develop applications that are highly specialised to the tasks at hand, whilst maintaining interoperability with other applications written by other companies. This ensures easy intercommunication between companies and departments, whilst allowing for extensive application modification.

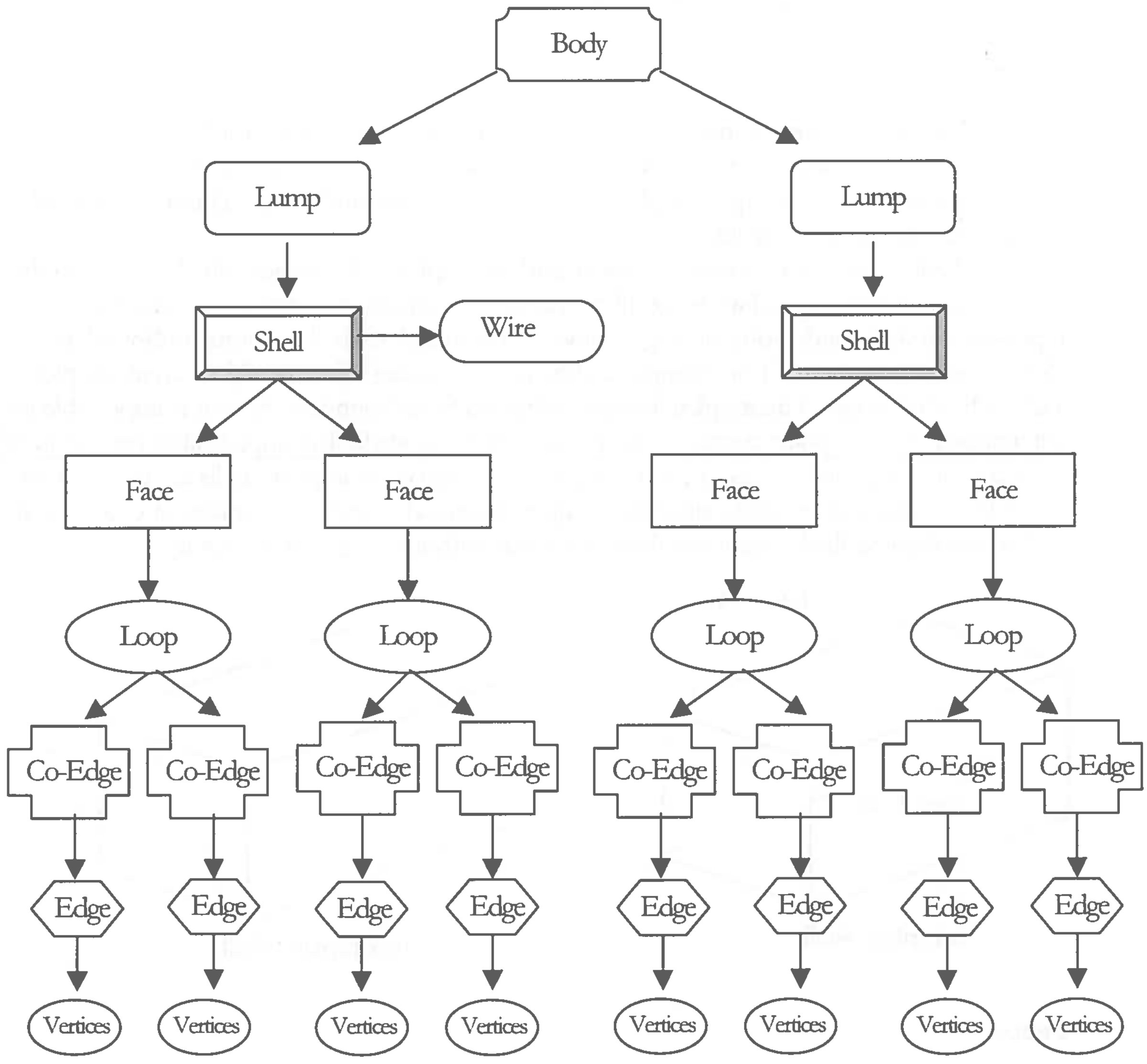
The primary design goal for ACIS is one of flexibility and expandability. For example, ACIS has numerous tools and behaviours depending on what needs to be modelled. Linear and quadric geometry is represented analytically (similar to SMS), whilst free form geometry is modelled by non uniform rational B-splines (NURBS). In addition to manifold geometry, ACIS can also represent non-manifold geometry. Objects can be bounded, unbounded, or semi-bounded, allowing for complete and incomplete bodies. For example, a solid can have a face missing, and a face can have a missing edge. This is unlike SMS, which implies that in order for an object to be valid it must be fully bounded. (Although unbounded objects will still compile.)

Some of ACIS's internal operations (such as Boolean operations) require bounded objects in order to function, but nonetheless unbounded models can be represented. Due to these operational constraints on some of ACIS's functions, ACIS can be classified as a boundary-representation modeller, which implies that a boundary must sit between solid material and empty space in order to gain from the more advanced ACIS tools. A more common name for this is a *solid modeller.*

One final way ACIS can represent complex geometry is through laws. These laws can deform more basic geometry to form new complex geometry that is created by deforming the original geometry by the application of a law. An example a helix could be created by applying a helix law to a torical arc (a curved cylinder). As the arc progressed round its radius, its geometry would by displaced by the helix law to form a helix structure.

3.2 The ACIS Model Hierarchy

The ACIS hierarchy refers to the spatial relationships between the various model entities. (Sometimes known as the model topology). By itself, the object hierarchy defines a "silly putty" model, whose position and indeed precise shapes are not fixed in space. For example a square plane and a skewed rhombus are topologically equivalent, but not geometrically. The hierarchical model precise size and shape are fixed when the object hierarchy is associated with geometric information. Like SMS, the ACIS hierarchy forms a tree structure – however unlike SMS it is more complicated. This is partly due to the additional object complexity that ACIS can offer, thereby requiring more levels in the hierarchy to support it. The other reason is for the purposes of flexibility – as mentioned in the chapter introduction, ACIS can support different types of model as well as different geometries. An example of this would be bounded and unbounded models. The result is a more complicated hierarchy than is seen in SMS, but with the offer of more flexibility in the model topology. A typical object in ACIS would have the following hierarchical decomposition.



3.3.1 The ACIS Model Hierarchy Components in Greater Detail

Let us examine the components of the diagram on the previous page in greater detail.

Bodies

The highest-level model entity in ACIS is a body. Typically, a body is a single solid component, such as a ball bearing or a stripped down engine block. A body can also be several disjoint bodies treated as one. Bodies consist of zero or more lumps.

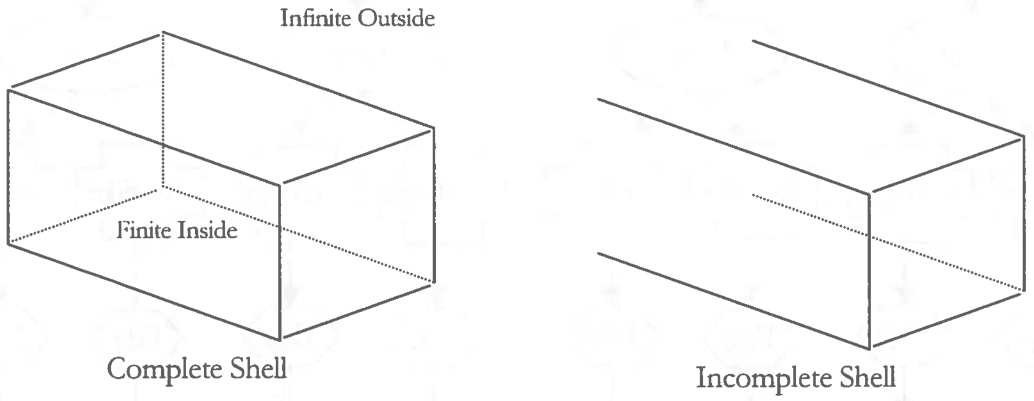
Lumps

A lump represents a bounded, connected region in space. For example, an entire connected set of points would be considered a lump, whether the set is 3D,2D,1D or indeed any combination of dimensions. A solid block with a dangling outside face is one lump, as is a solid block with an internal cavity. If a body consists of two disconnected objects, then each object will become a lump, and thus the body will have two lumps.

Shells

A shell is an entire connected set of faces and/or wires. For example a lump that consists of a cube would just have one shell, while a lump that consists of a block that has a cavity inside it (as in our lump example) would consist of two shells, one defining the outside of the lump, the other the inside.

Shells come in two forms, complete and incomplete. A complete shell adheres to the rules, in that all its faces and wires are all connected to each other in some way and thus represent a fully bounded object. (E.g. a cube). An incomplete shell has some unbounded element as a constituent. For example, a cube that is missing a face would be an incomplete shell. The shell is called incomplete because it has no finite boundary, since it is impossible to tell whether or not a point in space is inside or outside the shell. Incomplete shells are allowed in ACIS for compatibility reasons, but objects consisting of incomplete shells are denied a lot of ACIS'es manipulation tools, since they require bounded objects as a constraint of set theory and an incomplete shell cannot use these functions without danger of ambiguity.



Faces

A face is a portion of a single geometric surface in space, similar to a two dimensional version of a body. A face's boundary constitutes zero or more loops of edges. Faces can be either single or double sided. If a face is single sided, then points on one side of the face are considered to be inside the shell, and points on the other side to be outside the shell. If a face is two sided, then points on either side are considered to be *all* inside or outside the shell. If they are outside, then the face represents an infinitely thin 2D sheet, such as a perfect razor blade. (Might be too easy to cut yourself with this!). If they are all inside, then the face represents an internal partition embedded in the solid.

Loops

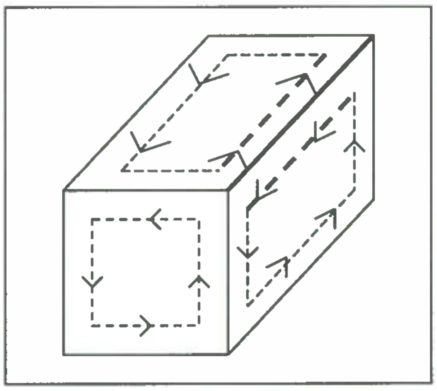
A loop represents a connected portion of the boundary of a face. It consists of a set of edges connected in a linked chain which may be either circular or open ended. A single circular loop can be used to define a face, or multiple open ended loops that achieve the same effect.

Wires

A wire is a connected collection of edges that are not attached to faces and therefore do not enclose any volume. Typically used to define either wireframe models or an infinitesimally thin passageway within bulk material. Wires are "islands" in the hierarchy, in that they are attached directly to shells. We shall utilise them in order to display SMS boundaries that are called from the assembly stage directly, and are thus not associated with any SMS surface.

CoEdges

A coedge records the occurrence of an edge in a loop of a face. The introduction of coedges permits edges to occur in one, two or more faces, and so makes possible the modelling of sheets and solids (manifold or not). A loop refers to one coedge in the loop, from which pointers lead to the other coedges of the loop.
Coedges in a loop are oriented so that looking along the coedge with the outward pointing face normal upwards, the face is on the left.
Coedges in a loop are ordered in a continuous path around the loop and are doubly linked. If a loop is not a circular list, the loop points to the first coedge.
In a manifold solid body shell, each edge is adjacent to exactly two faces; therefore, the edge has two coedges, each associated with a loop in one of the faces (the two faces can be the same, and even the loops can be the same). In this case, the two coedges always go in opposite directions along the edge.



In the figure to the right, an isometric view of a solid shows three faces. Each face is bounded by a loop of coedges. Each edge (corner of the block) has two coedges, one for each face that is adjacent to the edge. Each coedge is coincidental with the edge adjacent and parallel to it.
The coedges are shown as dished lines, with arrows to indicate their direction. Our chosen edge (the only bold solid line) has two coedges (also shown in bold), which have partner pointers to each other, because they are both associated with edge E.

In a sheet body, there may be edges which have only one coedge. These are knows as *free edges*, and they mark the boundary of a sheet. If the face attached to the coedge is single-sided, the inside and outside of the associated shell are not well defined near the edge, and so the shell is necessarily incomplete. (The meaning of shell incompleteness is discussed in the section on shells above.)

Edges

An edge is a line that is bounded by one or more vertices. The edge can be straight, or curved. (To be truthful, all edges in ACIS are curves, the straight ones have a curvature of zero.) One or more edges belong to each loop. Each edge contains a record of its sense (forward or reversed) relative to its underlying curve.

An important feature of ACIS edge representation is the arrangement of the coedges around an edge. If only two faces meet at an edge, the two coedges from those faces point to each other through the coedge partner pointers. (If there is only one coedge, its partner pointer is NULL.) If more than two faces meet at an edge, the coedges are in a circular linked list. The order of the list is important, because it represents the *radial ordering* of the faces about the edge in a counterclockwise direction.

Vertices

Finally, everything gets resolved to vertices. A vertex is a known point in all dimensions relevant to the model. They are used to bound edges. The vertices are the only hierarchical component that is also a piece of the model geometry – the rest of the hierarchy is purely topological.

To conclude, we have a different hierarchy here than we do in SMS. The main differences stem from the different design goals of SMS and ACIS. ACIS has to deal with a far greater number of different situations than SMS has to, as well as being specifically as general as possible – recall that ACIS is designed as an API for developers to build their own products on top of. This places a demand on the SAT format that we have shown here to be flexible to the extreme – and at the same time, have strong controls on the way objects are written in the SAT format in order to prevent any sort of ambiguity.

Now we shall have a quick look at equivalent ACIS geometric primitives to the SMS ones that we have already studied – with the aim of starting to think about possible methods of conversion.

3.3 The ACIS Geometric Primitives

The ACIS geometric primitives are, understandably, large in number and highly configurable. In the interests of validity, we shall limit our focus on the primitives that are comparable to the SMS geometric primitives covered in section 2.3. One fact that we should note is that because of the depth of ACIS, there is often more than one way to achieve the same result. For example, in the curves section to come we will discuss interpolated curves for the purposes of representing parabolic and hyperbolic curves. This same functionality could have been provided by either the application of laws to more simple geometry, or through the use of conics. The reasons why I have chosen a particular method over others is a combination of three main factors:

- The highest degree of similarity to the SMS primitive
- The lowest degree of error
- The highest degree of simplicity

Thus should we have a case where an SMS primitive will not natively translate into an ACIS representation, then our choice of method will be largely based on what offers the lowest degree of error, with simplicity a secondary consideration. Thankfully, most SMS primitives do natively translate into ACIS, so this is rarely an issue, however should there not be a direct representation then we will explore the choice we do make.

3.3.1 Curves

ACIS supports three types of curves:

- Analytic Curves - Simple curves that can be represented by non polynomials.
- Interpolated Curves - Complex curves that are represented internally by B-Splines.
- Composite Curves - Curves that consist of an ordered list of analytic and interpolated curves.

We are particularly interested an analytic curves to provide our lines and elliptical arcs, and interpolated curves for our parabolic and hyperbolic arcs.

Analytic Curves

Analytic curves are represented analytically by an algebraic formula. There are two subtypes of this curve type that interest us – straight and ellipse.

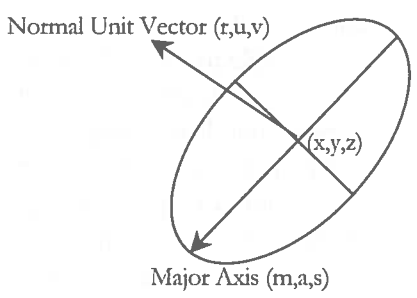
Straight Curves are simply lines – they have the form:

```
straight-curve $-1 x y z  u v w I I #
```

Where $-1 is a null pointer to separate the data from the name type, x, y and z denote the start position, and u v w denote a direction vector. Note the lack of either length information or a pointer to a terminating vertex. The information *is* present in the SAT file, but it is linked at a different point in the hierarchy. (In this case the edge that binds the straight-curve to the vertices – have a look at the hierarchy again ☺.)

The other analytical curve that interests us is the ellipse. It has the following form:

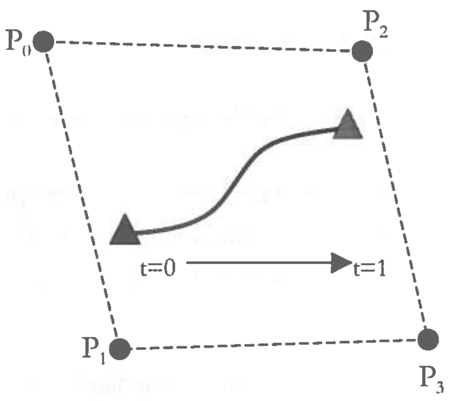
```
ellipse-curve $-1 x y z   r u v   m a s  R I I #
```



Where x,.y and z represent the position of the centre of the ellipse, and r u v represent the unit vector to the normal of the ellipse plane. (See diagram to the right) m, a and s represent the vector of the major axis of the ellipse from the ellipse centre, and R represents the ratio between the major and minor ellipse axis. If R=1.0, then the ellipse will be perfectly circular – which is very useful since we can use this property to represent SMS circular arc's as well as ellipses. Ellipses too have the property of relying on information elsewhere in the ACIS hierarchy (as do most of the ACIS geometric features). For example, the ellipse is bound by one or two vertices that are pointed at by the parent of the ellipse curve in the hierarchy, the edge.

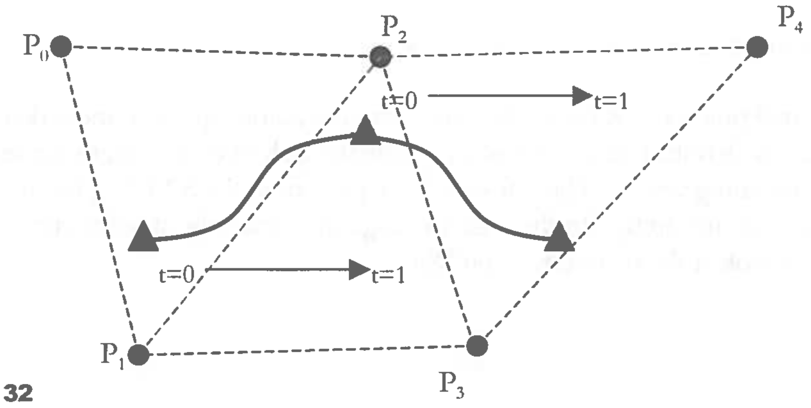
Interpolated Curves

Interpolated curves are used in ACIS to represent complex curves that a single formula could not easily describe. There are multiple classes of interpolated curve available, each with a different interface and parameters - however they all resolve down to B-Splines eventually. The idea behind B-Splines is that by placing 4 control points in 2D/3D space you can create a curve that is a function of those control points. Thus by moving these control points around you can control the shape of the curve.



When a point on the curve line needs to be calculated, the positions of the 4 control points, plus a value representative of how far down the curve your point is - t (t=0 at the start point, t=1 at the end point) are placed into a blending function. The outputs of that blending function are the co-ordinates of that particular point on the curve. An example of a B-Spline segment is to the right.

With a bit of thought and from the diagram, it is obvious that with only 4 control points, only very simple curves can be represented – no fine detail is possible. B-Splines overcome this problem through sub-division. Should a curve become too complex to represent with a single spline, then it is broken down into multiple splines, joined together by knots. To ensure continuity (i.e. No sharp changes in gradient), each curve segment shares 3 of its 4 control points with its neighbours, and the control points roll onwards as the curve passes each knot. The knots themselves play no part in the spline calculation, they simply mark out the domain of the control points. Consult the diagram below for an example of this.



As mentioned, there is more than one type of integrated curve available. The one we are interested in is called exactcur, which takes keypoints and a tolerance metric as parameters. ACIS will then construct a B-Spline that passes through or near the keypoints, depending on the tolerance value. The reason that this class of curve is so useful is that B-Splines themselves do not pass through their control points – ACIS will construct the control points for you that cause the curve passes through the keypoints you set. This method can be prone to error if there are any steep changes in gradient and not enough keypoints - as it is hard to place control points that both satisfy the needs for continuity and the need to pass through the keypoints – hence the tolerance value. This might become a showstopper for some applications, however, recall that the SMS primitives we wish to emulate are parabolic and hyperbolic curves. Due to their mathematical nature, the only time they have steep gradients is when x is very close to zero – the curve quickly flattens out after that – be it vertically or horizontally. Therefore it is possible to keep the B-Spline free of error, provided the regularity of the keypoints is a function of x. i.e. keypoints close together when x is small, further apart when x is large. This function is covered in more detail in chapter 4. Our choice of integrated curve has the following format in the ACIS SAT file:

```
intcurve-curve $-1 [forward/backward] { exactcur full nubs <knots>
[open/closed] <keypoints> {keypoint set - (x,y,z)} <tolerance>
null_surface null_surface nullbs nullbs I I 0 0} I I #
```

Where [forward/backward] represents the direction of the curve, exactcur and full represent our intcurve subtype, <knots> indicates the number of knots in the curve (integer), [open/closed] denotes whether the curve is a cycle (closed) or not (open). <Keypoints> represents the number of keypoints given in the keypoint set, and the keypoint set is a set of triples representing the location of the curve keypoints. Finally we have the tolerance value, which by default is zero, followed by pointers that would link the spline into a NURBS[1] surface if it were part of one. In this case it is not, so they are null. The intcurve needs only one or two vertices to bound it, and they are linked in at the edge stage in the hierarchy – in a similar manner to the two curves discussed previously.

The two types of analytical curve, straight and ellipse, combined with our chosen type of interpolated curve, exactcur, allow us to accurately portray any of the SMS curve types in ACIS. Now that we have a complete line set, let us move onto surfaces in ACIS, followed by a small example of an ACIS SAT file.

[1] Non Uniform Rational B Spines

3.3.2 Surfaces

In a similar theme to curves, the number of types of surface supported by ACIS is large. Therefore we shall limit our investigation into the types of surface that we could use to represent SMS geometry as accurately as possible. Recall that we are looking for surfaces to represent the following SMS primitives: plane, cylinder, cylindrical patch, cone, torus and doubly curved patch.

Plane

The ACIS plane is used to represent planar surfaces from triangles to N-gons. It can be bound by any sort of curve(s), with the provision that the curves in question must lie in a 2D plane with an identical normal to the surface plane and together form a complete cycle. It has the following format:

```
plane-surface $-1 x y z  n r m  u v e [forward_v/backward_v] I I I I #
```

Where x,y,z represent any point that lies on the plane (normally chosen as the centre of the plane – but not necessarily so), n,r,m represent the normal vector of the plane, and u,v,e represent the *u* derivative vector, which orientates the *u,v* parameter space onto the plane. This is by default perpendicular to the normal vector, but it can be changed – mainly for texturing purposes.

Cones and Cylinders

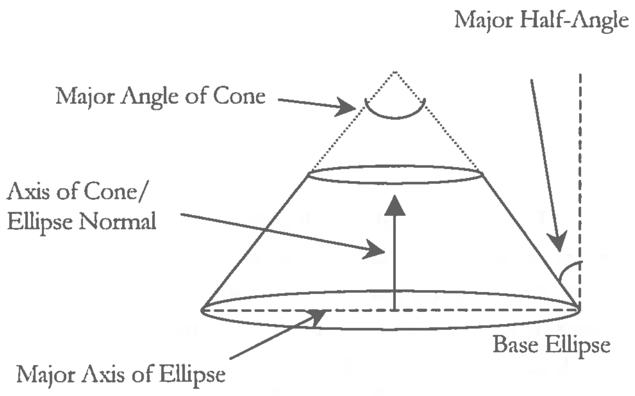
The cone primitive defines an elliptical single cone. It is defined by a base ellipse and the sine and cosine of the major half-angle of the cone. The normal of the base ellipse represents the axis of the cone. Two values, a sine angle and a cosine angle, decide the type of cone generated from the base ellipse. Between them, these two angles decide which of the following 4 possibilities occur:

- The radius of the cone gets smaller as we transverse up the ellipse normal
- The radius of the cone gets larger as we transverse up the ellipse normal
- The radius of the cone does not change as we transverse up the ellipse normal – i.e. we have created a cylinder. This is how cylinders are constructed in ACIS, as a subtype of a cone.
- The cone is planar and has no height at all. (Better represented by a plane)

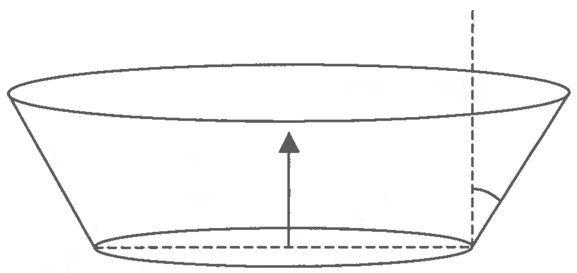
A diagram visualising these 4 possibilities can be found on the next page. With the information above in mind, the cone surface has the following form:

```
cone-surface $-1 x y z  r u v  m a s R I I <cosine angle>  <sine angle>
<u parameter scale> I I I I #
```

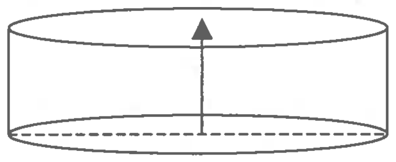
Note that the cone surface is similar in declaration to the ellipse curve – x,.y, z represent the position of the centre of the ellipse, and r u v represent the unit vector to the normal of the ellipse base to the surface. M, a and s represent the vector of the major axis of the ellipse base from the ellipse centre, and R represents the ratio between the major and minor ellipse axis. Finally we have the cosine and sine angles, followed by a u parameter scale for u,v mapping of the surface.



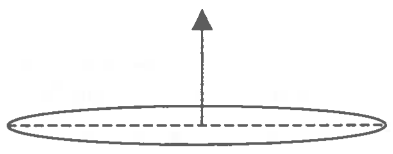
Case 1 – Cosine Angle Negative



Case 2 – Cosine Angle Positive



Case 3 – Sine Angle = 0



Case 3 – Cosine Angle = 0

Sphere

Spheres can be deformed in all 3 axis in ACIS, so they can represent all the shapes an ellipsoid in SMS can. They are not bound by default – just like SMS, but edges are permissible for extra detail (eg. holes or hemispheres) provided the supplied edges lie on the surface of the sphere. The ACIS sphere is defined as follows:
3

```
sphere-surface $-1 x y z  x1 y1 z1 [forward_v/backward_v] I I I I #
```

Where x,y,z represent the position of the centre of the sphere, x1 represents the distance vector to the surface along the x axis, y1 represents the distance vector to the y axis, and z1 represents the distance vector to the z axis. The forward/backward option controls whether the sphere surface points outwards (forward_v) or inwards (backward_v).

Torus

The torus in ACIS is defined by giving a position for its centre, the radius of the torus "spine", and the radius for the torus ring. The torus by default has its vertical axis in the +Z direction. It is defined as:

```
torus-surface $-1 x y z n r m <major-radius> <minor-radius> u v w
[forward_v/backward_v] I I I I #
```

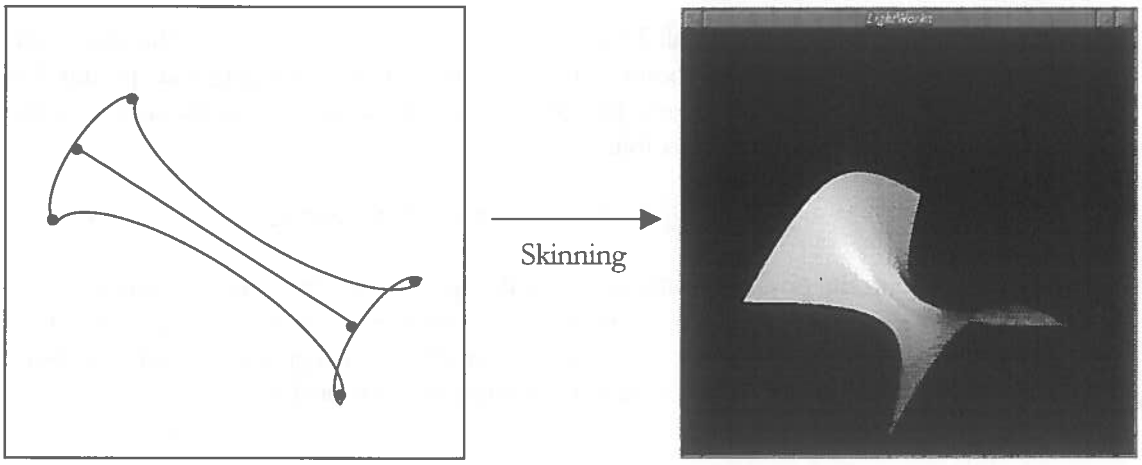
Where x,y,z represent the position of the centre of the torus, n,r,m the torus normal (i.e. direction), `<major-radius> the radius of the torus spine, <minor-radius>` the radius of the torus "tube", u,v,w the uv origin direction for the u,v parameters, and finally forward_v or backward_v for the surface direction.

Spline Surfaces

The representation of a doubly curved patch in ACIS is probably the hardest primitive conversion, since there is no corresponding ACIS primitive that we can base our inputs on. Instead, we are forced to provide our own ACIS implementation of a doubly curved patch using surface splines. The surface is declared through the creation of a spline "cage", ie. The surface is declared at key points using splines, and the surface is interpolated from these splines. This method is similar to the way we represent parabolic and hyperbolic curves using in that it there is the possibility of error – fine detail may be missed out in the interpolation. However, since the doubly curved patch in SMS is a function of two radii, we can be confident that the gradients of the curve will remain steady, and hence there is no risk of error.

For a visual example of what I am talking about, consider the problem of building a horse saddle. First, we would build a "cage" representing the shape of the saddle at key points out of 5 splines. 3 splines define the shape longitudinally, and 2 splines cap the ends. The splines are of the same type used to define the parabolic and hyperbolic curves, exactcur's, but they are essentially B-Splines. You can see a diagram of the spline cage below, to the left.

Once we have declared our splines, we can use them to declare a spline-based surface that interpolates between the splines in exactly the same way that the splines interpolate in-between themselves. The result of this "skinning" can be seen in the picture below to the right.



The spline surface has the following format in the ACIS SAT file:

spline-surface $-1 [forward/backward] { netsur v {splines} u {splines} F 0 F } I I #

Forward/backward represent the direction of the surface and netsur the surface subtype. V represents the number of splines in the v parameter direction, with {splines} representing that set of splines – (see interpolated curves for the fine details). Finally, u represents the number of splines in the u parameter direction, with the set it numbers following right behind ({splines}). The exact methods behind the generation of the spline cage are more of a conversion nature, and are thus detailed in chapter 4.

Summary

Between all the primitives we have covered here, we can describe any SMS primitive in ACIS, even in cases where the primitives do not natively translate, like doubly curved patches. Now let's bring all of our knowledge together in a concise ACIS SAT example.

3.4 A Closer Look at an ACIS SAT File

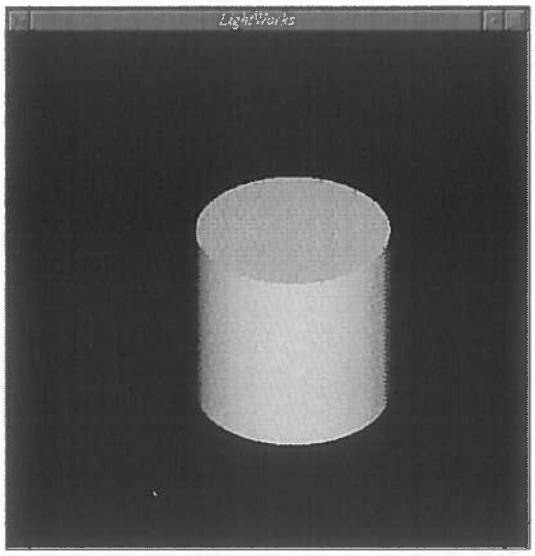
Now that we have a rough understanding of the ACIS hierarchy, and suitable ACIS primitives we can have a look at how ACIS actually saves it geometric data in an SAT file. The way data is structured in ACIS is quite different from SMS. SMS keeps control of its features though its sub-component hierarchy. For example, within a SMS plane primitive you could find everything you need to display that plane, with the exception of the line declarations. ACIS is slightly different. It too has a hierarchy, but it is more separated from the actual geometric constructs. There are entities, such as loops, whose only purpose is a hierarchical one, and to lock features together. For example – you may recall the fact that you cannot find the bounding information for ACIS curves with the curve declaration. The bounding vertices are linked in above the curve primitive at the edge node – there you will find pointers to the curve and its bounding vertices.

The following simple example shows a SAT file with topology and geometry. The first three lines are the header, followed by the entity records, and finally the end marker. Note the way SAT files very closely match the ACIS model hierarchy, although this should come as no surprise! The cryptic text represents the actual SAT file, the text after the # sight represents my comments as to what is going on. See if you can work out what object this SAT file represents! Note that $x is a pointer to line x, and $-1 is a Null pointer. Start with the body at line 0. (denoted by –0 at the beginning of the line).

```
400 0 1 0 Header Information
11 Scheme AIDE 11 ACIS 4.0 Solaris 24 Mon Dec 02 13:59:03 1999 Header
Information
25.4 1e-06 1e-10 Header Information
-0 body $1 $2 $-1 $3 # A body with display atributes at line -1, lump at
line -2, and translation at line -3
-1 display_attribute-st-attrib $-1 $4 $-1 $0 1 # ACIS Renderer Setup for
this object
-2 lump $-1 $-1 $5 $0 # A lump whose shell is at line 5, and whose
parent is at line 0
-3 transform $-1 1 0 0 0 0 -1 0 1 0 0 10 0 1 rotate no_reflect no_shear
# Translational Information
-4 rgb_color-st-attrib $-1 $6 $1 $0 0 1 0 # Colour values of the object
-5 shell $-1 $-1 $-1 $7 $-1 $2 # A shell that consists of a face at line
7
-6 id_attribute-st-attrib $-1 $-1 $4 $0 1 # More renderer information
-7 face $-1 $8 $9 $5 $-1 $10 forward single # A face that consists of
another face at line 8, a loop at line 9 and a surface at line 10
-8 face $-1 $11 $12 $5 $-1 $13 forward single # A face that consists of
another face at line 11, a loop at line 12 and a surface at line 13
-9 loop $-1 $14 $15 $7 # A loop that consists of another loop at line
14, and a coedge at line 15
-10 cone-surface $-1 0 0 0 0 0 1 10 0 0 1 I I 0 1 forward I I I I # A
cone surface with numeric values
-11 face $-1 $-1 $16 $5 $-1 $17 forward single # A face that consists of
a loop at line 16, parent shell at line 5 and a surface at line 17
-12 loop $-1 $-1 $18 $8 # A loop that has a coedge at line 18
-13 plane-surface $-1 0 0 -10 0 0 -1 -1 0 0 forward_v I I I I # A Plane
surface with numeric values
-14 loop $-1 $-1 $19 $7 # A loop with coedge at line 19
-15 coedge $-1 $15 $15 $18 $20 1 $9 $-1 # A coedge, connected to another
coedge at line 18, and with an edge at line 20.
-16 loop $-1 $-1 $21 $11 # A loop with coedge at line 21
-17 plane-surface $-1 0 0 10 0 0 1 1 0 0 forward_v I I I I # A Plane
surface with numeric values
-18 coedge $-1 $18 $18 $15 $20 0 $12 $-1 # A coedge with an edge at line
```

```
20
-19 coedge $-1 $19 $19 $21 $22 1 $14 $-1 # A coedge with another coedge
at line 21 and an edge at line 22
-20 edge $-1 $23 $23 $18 $24 forward # An edge, bounded by vertex at
line 23, edge type at line 24.
-21 coedge $-1 $21 $21 $19 $22 0 $16 $-1 # A coedge that joins the loops
at lines 14 and 16 together.  Has an edge at line 22
-22 edge $-1 $25 $25 $21 $26 forward # An edge, bounded by vertex25 of
edge type at line 26
-23 vertex $-1 $20 $27 # A vertex, bounding the edge at line 20,
numerical values at line 27
-24 ellipse-curve $-1 0 0 -10 0 0 -1 10 0 0 1 I I # The type of edge 20
-25 vertex $-1 $22 $28 # A Vertex, bounding edge 22, numerical values at
line 28
-26 ellipse-curve $-1 0 0 10 0 0 1 10 0 0 1 I I # The type of edge at
line 22
-27 point $-1 10 0 -10 # Position of the vertex at line 23
-28 point $-1 10 0 10 # Position of the vertex at line 25
End-of-ACIS-data
```

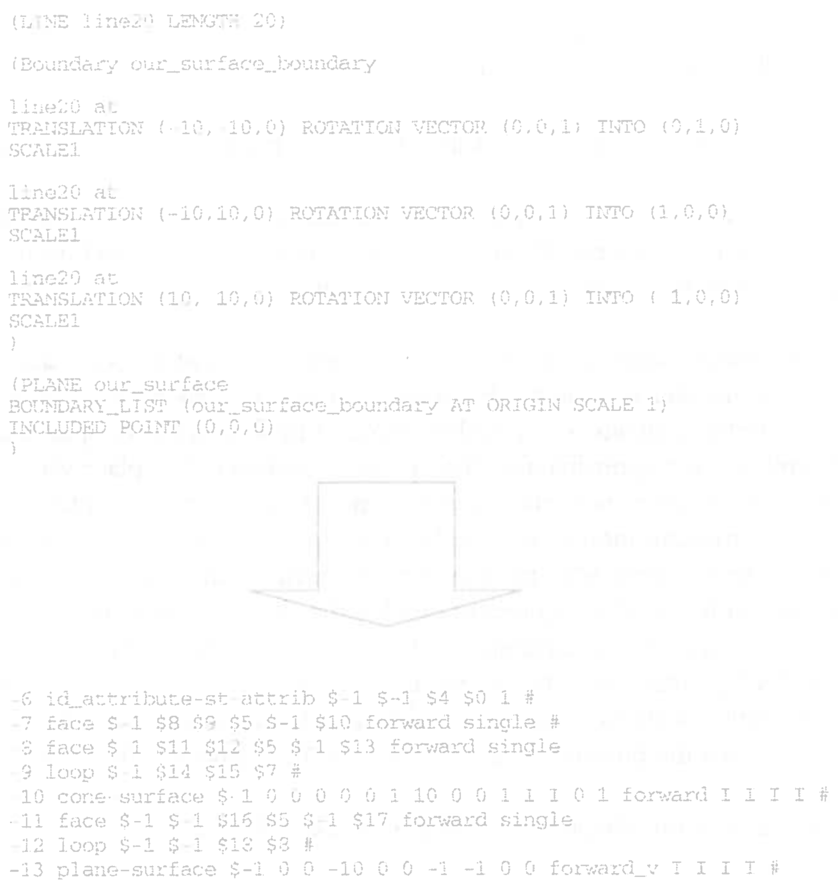
The clues to what shape this file represents are the fact that each edge only has one bounding vertex, and the cone surface call on line 10. The fact that the edges only have one bounding vertex means that the edges must be circles, and the cone-surface is joining them together. Thus our shape is a cylinder! A picture of it may be found below.



3.4 A Conclusion on ACIS

By now we have covered all the features of ACIS that we need to know about in order to carry out our design and implementation of a conversion system. Yet we have barely scratched the surface of what ACIS can do, and the functionality it can provide. Yet now we must turn our attention to the main goal of the project. In the next chapter, our diligence over these last two chapters will (hopefully!) be paid off. There we shall discuss the various requirements of the conversion process, what potential problems it must overcome to be successful, and the algorithms that drive its operation.

Chapter 4



```
(LINE line20 LENGTH 20)

(Boundary our_surface_boundary

line20 at
TRANSLATION (-10, 10,0) ROTATION VECTOR (0,0,1) INTO (0,1,0)
SCALE1

line20 at
TRANSLATION (-10,10,0) ROTATION VECTOR (0,0,1) INTO (1,0,0)
SCALE1

line20 at
TRANSLATION (10, 10,0) ROTATION VECTOR (0,0,1) INTO ( 1,0,0)
SCALE1
)

(PLANE our_surface
BOUNDARY_LIST (our_surface_boundary AT ORIGIN SCALE 1)
INCLUDED POINT (0,0,0)
)
```

```
-6 id_attribute-st-attrib $-1 $-1 $4 $0 1 #
-7 face $-1 $8 $9 $5 $-1 $10 forward single #
-8 face $-1 $11 $12 $5 $-1 $13 forward single
-9 loop $-1 $14 $15 $7 #
-10 cone-surface $-1 0 0 0 0 0 1 10 0 0 1 1 I 0 1 forward I I I I #
-11 face $-1 $-1 $16 $5 $-1 $17 forward single
-12 loop $-1 $-1 $13 $8 #
-13 plane-surface $-1 0 0 -10 0 0 -1 -1 0 0 forward_v I I I I #
```

The SMS to ACIS Conversion Process

4.1 An Introduction to the Conversion Process

The central goal of this project is the construction of a process that supports the conversion of model descriptions written in the SMS modelling language into descriptions suitable for building an ACIS SAT file. In this chapter we will examine the process that was developed, and some of the reasoning behind the features that it displays.

One of the major points about the conversion system is that its foundation is based on the existing SMS compiler and its supporting libraries. The reason for this is that by utilising the SMS compiler, we could save ourselves time and effort from:

- Eliminating the need to construct an SMS Parser.
- Eliminating the need for complex syntax/grammar checking the SMS files, since the SMS compiler would check the incoming files for validity.
- Utilising some of the existing functionality of the SMS libraries to carry out required tasks.

In addition to all of this, the SMS compiler places the data contained in the SMS file into ordered and logical data structures. By accessing these structures, we would have a good way of capturing all the data necessary for the conversion process.

The conversion process takes this data, and assigns it to instances of various classes, dependent on what the data represents. Each major primitive in SMS has its own class – this provides us with useful flexibility, since the behaviour of each class can be specifically tailored to the SMS primitive it is responsible for. The actual conversion takes place via intercommunication between these classes, with some taking a more authoritative role than others. This intercommunication is described in detail in section 4.x. The net result is that each of the classes representing SMS primitives in the scene return a segment of an ACIS SAT file that represents the ACIS equivalent of what the class represents in SMS. These messages then get passed to an amalgamation class, which binds all the messages together and alters the ACIS SAT pointers such that all the segments interconnect in a valid way according to the ACIS hierarchy. With the addition of header and footer information, this amalgamation becomes the finished ACIS SAT file, and the conversion process is complete.

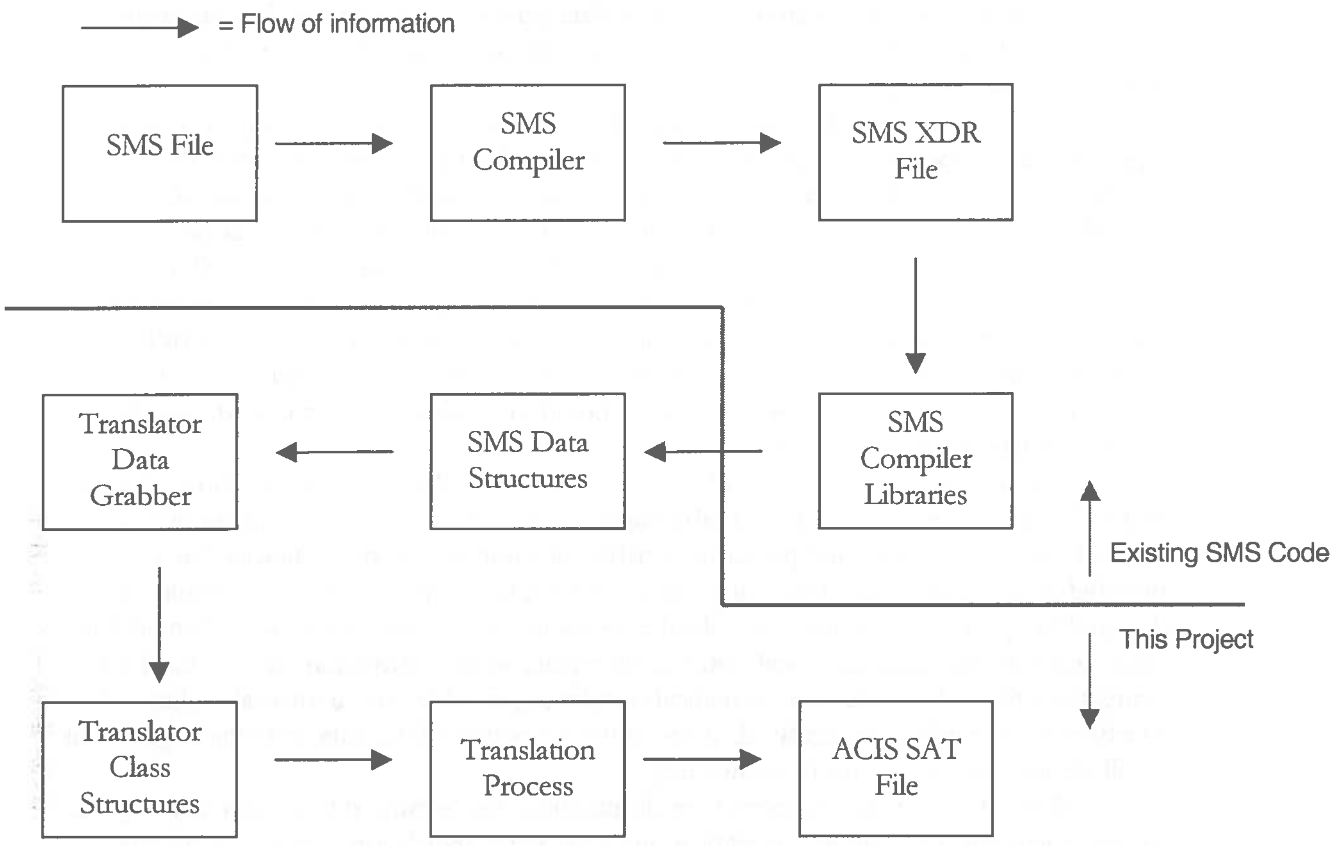
In the course of this chapter, we shall discuss the following:

- The overall architecture of the conversion system
- The functions of the conversion system classes and their responsibilities
- The boundary conversion system
- The top level translation process
- The feature level translation process
- The final amalgamation stage
- A simple example of the system in operation

Now let us begin by developing an overall view of the conversion system by examining its large scale structure and design.

4.2 Architectural Overview

A top-level filter and pipes style diagram of the system process architecture can be found below. Green shading means the box represents a data set, beige means a data processing activity. Note the boundary between the pre-existing SMS compiler and this project.



The SMS compiler takes an SMS file as input, compiles it, and then saves the output as an XDR file. The XDR file is part of Sun's Remote Procedure Call mechanism, and is essentially the class instances generated by the SMS compiler saved to disk. We can utilise functionality from the SMS compiler libraries to restore these class instances – represented in the diagram above by the SMS Data Structures box. At this point we can begin the process of acquiring the data necessary for our translation to occur from the SMS data structures. This is thankfully made fairly easy due to the organised way the SMS data structures are set up. A variety of SMS classes store all the information necessary for SMS to build the models. An instance of the SMS Object Manager class keeps track of all these classes through a table of pointers. Thus all we have to do to get the information we require is to parse through all the class instances pointed to by the master Object Manager class.

What we do once we get to a particular instance depends on that type of feature it is. For example, if it is a line then we copy its name, type (e.g. straight or curved?), length and endpoints to an instance of our own line class (more on that later). We also copy information about the object translation and rotation. For a more complex object like a boundary we copy the boundary name, the name of each boundary member, plus rotational and transitional information for each member of the boundary. A similar process occurs for the rest of the SMS primitives, leaving us with a complete representation of the SMS scene ready for translation. While it might seem that the translator data grabber stage is simply responsible for the construction of the translator data structures, it does have two important responsibilities beyond that. Let's take a quick look at them.

4.3 Unified Rotational Specifications and Expression Evaluation

So far we have only seen the translator data grabber stage perform data management operations. However it does have two tasks of translation that it carries out whilst transversing the SMS data structures.

The first is to do with rotations. Recall from Chapter 2 that SMS supports multiple types of rotation specification, including Rotate Slant Tilt, Quaternion and Vector Pair notation. When SMS is compiled, these different rotation specifications are preserved within the SMS data structures. Since we are only interested in a single method of rotation specification for our purposes of conversion to ACIS, it makes sense to convert all the different forms of rotation to a single unified form when we place the data into our own class instances. Each rotation specification in SMS has its own conversion procedure, which takes the rotation and converts it into a 3 by 3 rotation matrix, which is the standard method of describing rotation in the conversion system. Should the rotation be unspecified, then the feature will have a rotation of zero.

The second is to do with variables and expressions. Recall once more form Chapter 2 that SMS supports expression and variable substitution. The SMS compiler does collapse most of these expressions, and performs variable substitution in some instances, but it provides no guarantees that it will carry out all of them to completeness. Thus should our data grabber process detect any unresolved expressions or variables, it will solve them at this stage, and place the final numerical values in their place in the conversion class. It resolves expressions through standard mathematical calculation, variables via an internal lookup system to where the variables were declared. (One of the properties of the data grabbing stage is that it will always visit variable declarations first!)

Both of these tasks are carried out during this stage because this is a convenient place to carry them out, and it allows simplification of the actual translation system. By solving these problems here, we create a unified environment for all the conversion system classes, one in which we only deal with comparable numerical representations and values.

By the end of this stage, we have everything in place to actually begin the conversion process that will produce our ACIS SAT representation of the scene. Let us take a closer look at the conversion stage by examining its components, architecture and how it solves key problems.

4.4 The Translator Classes and Class Diagram

As mentioned previously, each SMS primitive and hierarchical structure has its own class. They are:

- Sms2acis_Line – This class is responsible for the data storage of all SMS line data, be it line, elliptical or parabolic. It is responsible for the conversion of the SMS lines into their ACIS equivalent, as requested by the sms2acis_assembly class. It can also output its contents in their original SMS form – this is used by the boundary conversion class, sms2acis_boundary.

- Sms2acis_Boundary – The boundary class has one of the biggest responsibilities of an Sms2Acis class. Not only must it contain information on each boundary instance, but it is also responsible for the process of boundary integration, which is explained in section 4.4. It is frequently queried by other sms2acis classes in regards to their boundaries, and it returns solved ACIS line geometry in response to these. (Again, section 4.x explains this process!)

- Sms2acis_Plane – Responsible both for holding data about each plane and for the translation of the plane into an ACIS specification.

- Sms2acis_Cylinder – The cylinder class holds information about each cylinder primitive declared in the SMS scene, and is also responsible for the translation of the cylinder via co-operation with the appropriate boundary class.

- Sms2acis_Cylinder_P – As above, but for cylindrical patches.

- Sms2acis_Ellipsoid – Instances of this class hold information about any ellipsoids declared in the SMS scene. Like the other SMS primitive classes, it is also responsible for the conversion of its primitive into ACIS.

- Sms2acis_Cone – Holds data about SMS cones, and the functionality to convert them to ACIS primitives.

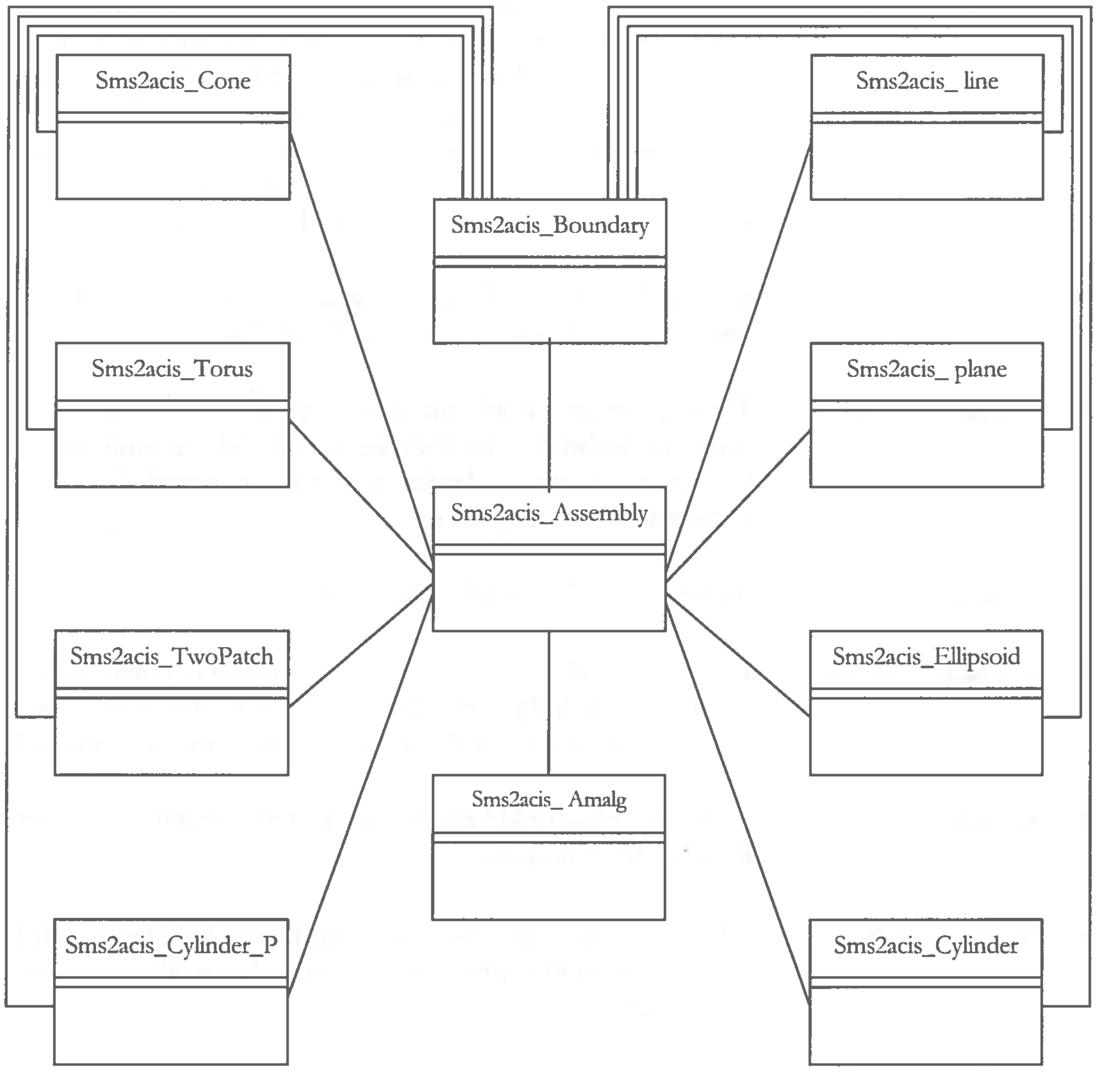
- Sms2acis_Torus – The Sms2acis_Torus class is responsible for holding information on any SMS touri in the scene, as well as the usual conversion functionality.

- Sms2acis_TwoPatch – This class is responsible for the holding and conversion of SMS doubly curved patches. It has a slightly more difficult job that the rest of the primitive classes (you may recall why from Chapter

- Sms2acis_Assembly – The assembly class holds all the information given by the SMS assemblies. It deals with sub assemblies by carefully merging them with the main assembly at load time – whilst maintaining the subassemblies transformational and rotational information. During the translation stage, SMS primitive classes query the assembly class to find out their positional and rotational information within the assembly, so that they may accumulate that with their own internal values to reach correct final values.

- Sms2acis_ amalg – This class is not responsible for holding any information about the SMS geometry. Instead it is responsible for the amalgamation of all the ACIS sections that are passed to it by the SMS primitive classes when they undergo their translation operations. It is responsible for the re-mapping of the ACIS pointers from all the fragments and to bring them all together into a single ACIS scene.

Class Diagram

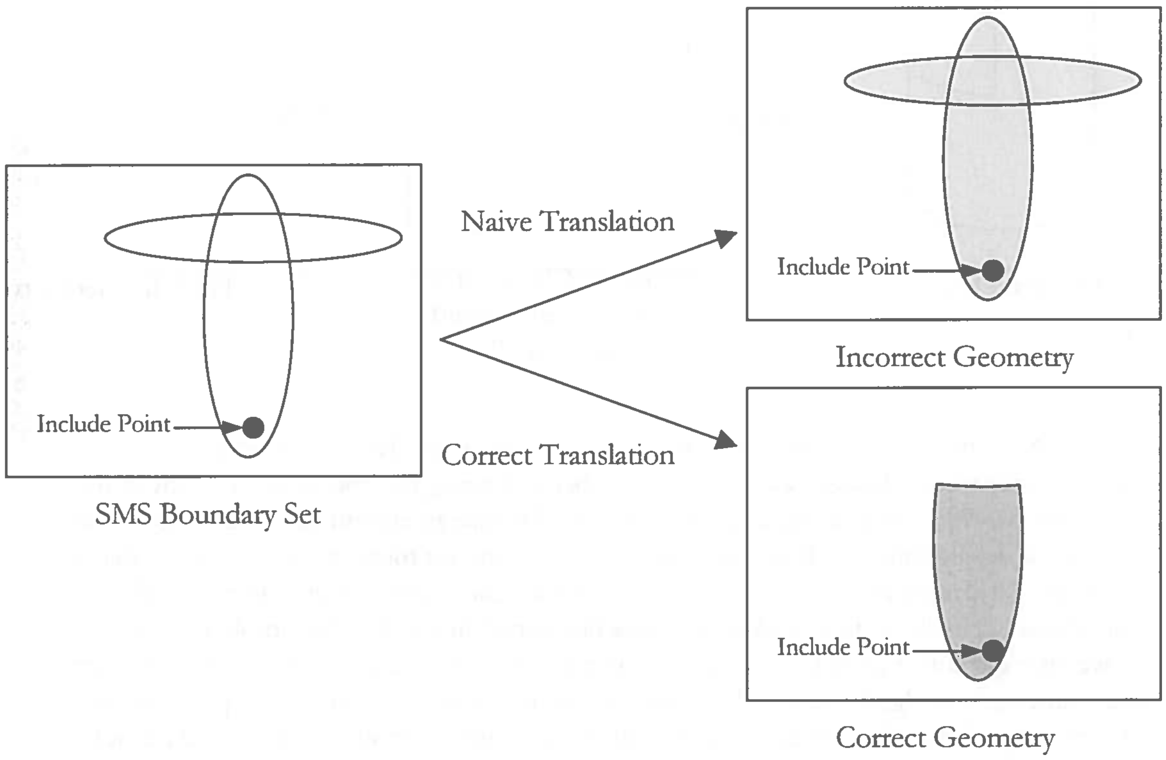


The class diagram above shows the static structure of the conversion process – a line in-between two classes means that the two classes can communicate with each other during the conversion process. Now we have examined the overall structure of the system, let us delve into the specifics of each class, what functionality they lend to the conversion system and how the overall conversion takes place. First of all, let us examine how the sms2acis_boundary class converts SMS boundaries into ACIS representations.

4.5 The Boundary Translation System

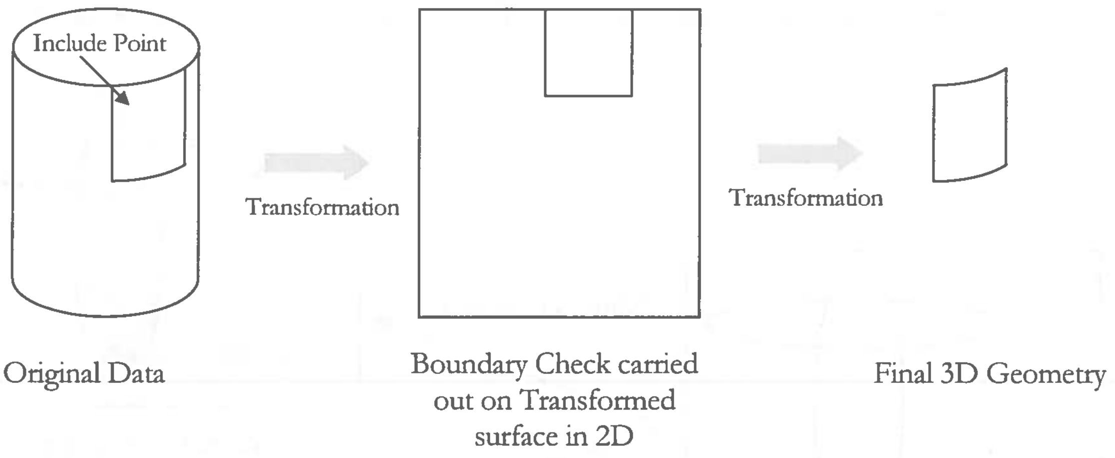
The biggest single problem to be solved in the conversion process is the translation of the SMS boundaries. Not only do we need to convert the boundaries described in SMS, but we must also validate them when constructing geometry from them. This issue was briefly discussed in Chapter 2, when we noted that ambiguities were possible, as well as boundary overlap.

Ambiguity's can exist because the SMS compiler does not check that the SMS boundaries are closed. A boundary in space is closed when it forms a closed region in 2D space. Furthermore, the checking for this loop is not necessarily naïve – a closed region in Space may be formed through the intersection of the boundary lines, rather than a loop of lines that share vertices. Boundary Overlap can exist where two boundaries coexist in the same region of 2D space. An example of boundary overlap was given in section 2.8, where we examined a process that could create an octagonal patch via the use of a square boundary applied twice. The reason we need to validate the boundaries is simple – if we do not, then we risk constructing geometry that is incorrect. For an example of this, please consult the diagram below.



It was decided that the behaviour of the boundary conversion should be that if a closed region exists around the include point – even if such a region is not depicted by a single set of submitted boundaries, but the submitted set as a whole, then the class should attempt a conversion. If however, there exists no closed region around the include point, *and the calling primitive is infinite* then we should raise an exception and exit, since there is now way to convert an infinite feature. The reason for the calling primitive check is that some SMS primitives are naturally bound, e.g. ellipses, and therefore can safely exist with no boundaries.

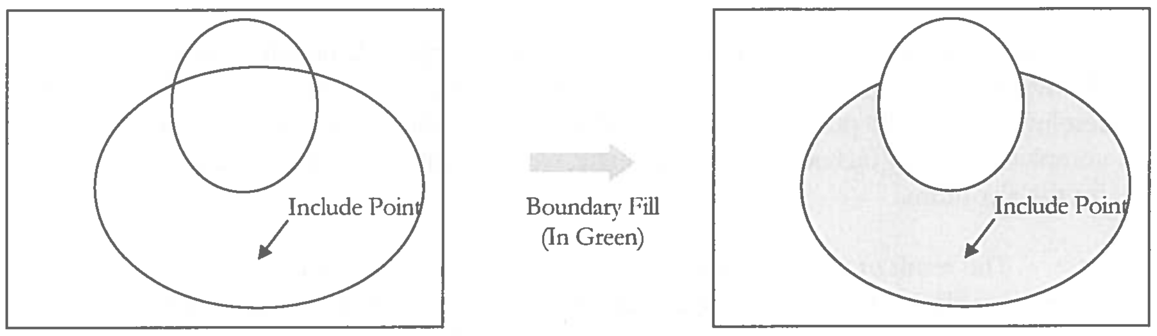
Now that we realise that a process to check the boundaries is necessary for the construction of correct geometry from them, how may we go about designing such a process? One important point to realise is that because boundaries may only exist on a surface, we can with care use transform functions to convert boundaries described in 3D onto a 2D plane. A good example of this the well-known cylindrical map of the Earth. Here a 3D spherical surface undergoes a transformation and is projected onto a 2D plane – the map. There are problems with this however – for example this projection does not conserve shape or area – particularly close to the poles. However since we will transform the information back to its original 3D form this is not a problem for us, since the transform function to return the 2D plane back into 3D will faithfully undo any projection errors. Therefore we may vastly simplify any algorithm we build by carrying out our entire boundary checking in 2D. The figure below gives a visual example of 3D boundaries undergoing this transformation->boundary check->transformation process.



Now that we can carry out our boundary checking in 2D, what processes can we utilise to construct validated boundaries from them? During the course of my study in this area, two possible solutions were developed – the *Boundary fill* algorithm and the *Boolean curve-curve intersection* algorithm. These essential difference between these two algorithms is that the boundary fill algorithm is discrete, whilst the Boolean curve-curve intersection algorithm is continuous. The boundary fill algorithm was developed first and is the simplest in nature, however it did suffer from the problems of being a discrete solution, and the Boolean curve-curve intersection algorithm was developed as a more elegant solution to the problem. It is also the algorithm utilised in the final version of the actual solution. Let us have a quick look at the boundary fill algorithm, why it was replaced, and a more detailed look at the Boolean curve-curve intersection algorithm.

The Boundary Fill Algorithm

The boundary fill algorithm is well known in computer graphics, and is used for rasterisation. It works by giving the algorithm a starting seed point in a 2D scene. It then proceeds to recursively colour the scene until it reaches a different colour than the background one, at which point it stops and returns. The recursion continues until there are no more pixels available with the original background colour. A common use for this algorithm is for the "paint bucket" tool seen in many drawing programs. A small graphic of this algorithm in action can be seen on the next page.



We can use this algorithm to calculate correct solutions to our boundary by performing the following steps.

1. Declare a large array of pixels with a default background value.

2. Obtain the set of boundaries for this feature.

3. Draw the boundaries (in a different colour) onto the array of pixels via a transformation function suitable for the SMS feature that is undergoing conversion.

4. Perform a boundary fill with the include point as the seed point for the algorithm.

5. Turn all of the pixels that were drawn in the boundary colour back into the background colour. This leaves us with just the fill colour and the background colour.

6. Trace around the fill colour with the boundary colour. This step gives us our final boundary representation. If tracing is impossible because the whole array has become the fill colour, them we know that the boundary set is not closed and we raise an exception.

7. Construct the boundary utilising an ACIS B-Spine curve, feeding every pixel that has the boundary colour in as a keypoint of the curve – with the keypoint undergoing transformation back into 3D space. The resulting curve is our boundary.

The advantages of this solution are that it is simple, fast and robust. It is capable of detecting when the boundary set is open rather than closed, handle boundary cases of boundary overlap, and will always produce a generally correct geometrical description of the solved feature boundary.

The disadvantages are that it can generate a lot of keypoints for the B-Spine, and that since the B-Spline is an estimation of the boundary, errors can manifest themselves in two ways – *Aliasing* and *Overshoot*. Aliasing is created because during this algorithm, we move the boundary from a continuous solution (line equations) to a discrete solution (pixels – specific values at specific points, with no data in-between). The result is that if there are boundary features small enough not to be represented accurately during the boundary drawing phase, then the resulting boundary will not depict them accurately either. The other problem, overshoot, is caused by a property of the ACIS B-Splines. Recall that the version of the ACIS B-Splines we use here, exact_cur, can suffer from error should the tangent of the line they represent vary wildly. However, such a thing can occur here – highly acute angles can occur close to boundary intersection points. The result is known as overshoot – whereby the Spline

"Balloons" out between the two keypoints at the acute angle. Although both these disadvantages can be reduced in seriousness by increasing the sampling rate (ie. Increase the resolution of the 2D pixel plane when we draw the boundaries) they never go away completely, and by increasing the sample rate, we can vastly increase the memory requirement for the algorithm.

The result of these problems was a desire to create a better algorithm, one that did not resort to a solution based on discrete values, but instead solve our problem whilst preserving continuity. The result of this was the Boolean Curve-Curve Intersection Algorithm.

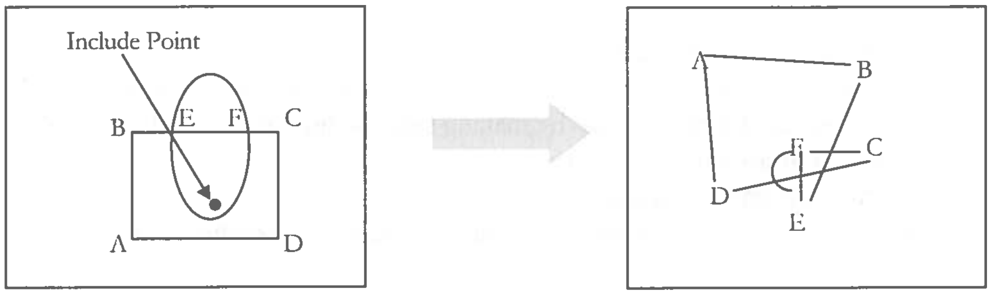
The Boolean Curve-Curve Intersection Algorithm

The first step of the Boolean Curve-Curve Intersection Algorithm is to identify where all the intersection points of our boundary set take place. How can we calculate this? The first step to a solution is to note that all of the curves (including the straight ones) in SMS can be represented as a quadratic equation, that is some form of $x^2 + y + c = 0$, once that curve has been transformed from 3D to 2D. When two curves intersect, there exists a point (or indeed multiple points) where the two equations that represent the points are equal to each other. That is, p(u) – q(t) = 0, with p(u) representing the first equation, and q(t) representing the second. We can find out where these points are by solving for the roots of that equation – that is solving two simultaneous equations for parameters u and t:

$$p_x(u) - q_x(t) = 0$$
$$p_y(u) - q_y(t) = 0$$

In reality, these equations are solved via parameterised gaussian elimination. This method will allow us to calculate the co-ordinates of any intersections that occur between any two lines that SMS can represent – with the addition of sanity checks to exclude any theoretical intersections that do not exist because that line has been capped by its SMS endpoints. To calculate all of the intersection points in the boundary set for an SMS surface, we simply solve the roots for each line with every other line, recording the co-ordinates of each intersection and what lines they belong to. With this data in hand, we press on.

The next step of the algorithm is to construct a graph. The nodes of this graph represent either the endpoints of SMS lines as declared in the SMS code, or our newly calculated intersection points. An edge is drawn between two nodes in the graph if a line exists between the two points in the boundary set. (Note that this includes nodes that connect to themselves – e.g. closed ellipses.) This can be easily calculated from an amalgamation of who the intersections and endpoints belong to. The result is a graph full of various interconnections, and normally numerous cycles, with each cycle representing a closed region in 2D space. If there are no cycles, then we know that there are no closed spaces in the set of boundary elements provided, and we may raise an exception. A simple graphical example of this stage may be found on the next page. If there are any cycles to be found in the graph, then we continue onto the next stage.



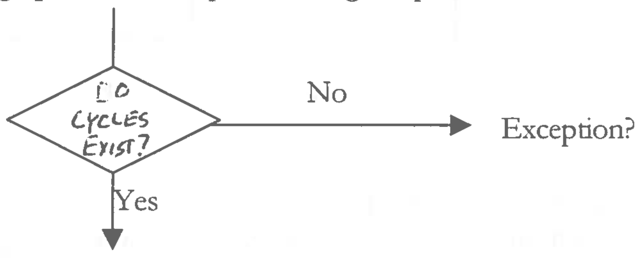
The SMS include point for the feature we are bounding is now brought into the equation. For every cycle in the graph,(*including* one vertex cycles) we apply a test to see whether or not the include point resides within the cycle in its geometric form. Multiple cycles between two vertices count as different cycles overall. In addition, care must be taken here, as we cannot rely on the vertexes alone, since one of the sides may be a curve and thus the polygon may not by convex. However, with knowledge of the type of each line, we can utilise properties of each type to calculate which side of the line the include point lies on. From the results of these investigations, we split up the cycles into two groups – those that have the include point within their boundary, and those that have the include point outwith. Those that do not have the include point inside them are laid to one side, and we continue.

The correct boundary representation lies within our group of remaining possible boundaries. How do we select the correct one? The correct boundary will be the one that provides a boundary that is closest to the include point at *all possible times*. This does not necessarily mean that the correct boundary is the one that has the shortest boundary length, (in fact there are simple cases to prove this), but it *does* mean that the correct boundary is the one with the least volume! So we calculate the volume of all the remaining possible boundaries. This itself incurs an extra step, which is to rationalise the boundaries we do have. This involves constructing new lines that fit the intersection points that the graph cycles identify. For example, an ellipse that is part of a graph cycle would be checked to see if its endpoints are the same as the points depicted in the graph. If one or both of the points in the graph are not the ellipse endpoints, then we know that the point mentioned in the graph must be an intersect point and we may replace one of the ellipse endpoints with that point. Similar actions are carried out for all the other curves. The result is a set of rationalised boundaries ready to have their volumes calculated. We then calculate the volumes of every boundary that is still in our selection set and the boundary that has the least volume is our final correct boundary.

Once we have our "winner" we undergo one final check – to see if any of our boundaries that did not have the include point within them exist totally within our chosen boundary, and thereby create a hole inside it. Once this check is made, can then apply a transform function to each of the boundary members and their associative vertices (if applicable) to give us 3D co-ordinates of the vertices in correct space. The transform function will also give us corrected quadratic equations that represent the equations of our correct boundary member curves in 3D space. Once we have those equations, they are converted to ACIS primitives directly if possible – via functions discussed in section 3.3.1, or to 3D B-Splines if not.

Algorithm Summary

1. Find quadratic equations for all lines.
2. Solve parameter roots for all lines, place intersection co-ordinates in lookup table.
3. Sanity check intersection points by making sure the lines they relied on were not capped by an SMS endpoint.
4. Add any endpoints to lookup table
5. Construct graph from lookup table – edges represent interconnection.



6. For every cycle, test to see if the include point lies within.
7. If so, calculate volume of boundary.
8. Smallest volume boundary wins.
9. Check that no boundaries from 6 lie within winning boundary.

The main advantage of this algorithm is that it produces far fewer errors than the boundary fill algorithm, due to the fact that the Boolean curve-curve intersection algorithm solves its problems algebraically, and never resolves to discrete methods – ie. It remains continuous at all times. It also does not suffer from overshoot, since even if it does resort to B-Splines to describe the quadratic equations that are the output of a boundary translation, each curve is representative of an SMS primitive. Recall from section 3.3.1 that SMS primitive conversions are not susceptible to potential B-Spline errors, due to the fact that they make no drastic changes in tangent. This implies that for an SMS boundary to make a highly rapid change in tangent, it requires the transversal between two line segments. This is the case, and since there is no spline continuity between line segments the risk of overshoot is removed.

The main disadvantage of the Boolean curve-curve intersect algorithm is that it is bulkier than the boundary fill algorithm, and the way in which it calculates the volumes of the potential boundary list can be expensive computationally. In addition to this – it is still not guarantee absolute accuracy, although any errors will be very small. The reason for this is that we are still forced into some measure of discreteness by selecting keypoints for the B-Spines, however the possible error will be very small.

The Generic Operation of the sms2acis_boundary Class

Now that we have examined how the sms2acis_boundary class actually performs the task of converting SMS boundary data into valid ACIS representations, let us have a look at how the sms2acis_boundary class behaves.

The sms2acis_boundary class has two modes of operation, *assembly call* mode and *feature call* mode. Its assembly call mode is by far the most simple, and it is designed for the case where the class is called by the sms2acis_assembly class in order to generate lines without a feature attached to them than are incorporated directly into the scene. In this mode, no boundary validation occurs – simply the conversion of the SMS curves to their equivalent ACIS primitives. The class requires that the sms2acis_assembly provides the names of the boundary lists involved and the sms2acis_boundary class will call each instance of sms2acis_boundary that stores the boundary list to be converted. The sms2acis_boundary class instance(s) then converse with the sms2acis_line classes in order to obtain ACIS chunks that represent ACIS representations of the SMS curves. The boundary class then amalgamates these whilst amending them if necessary with ACIS translation lines and returns a single chunk of ACIS data that consists of all the boundary members aggregated together. The "master" sms2acis_boundary class then performs a final aggregation, and returns the ACIS SAT segment to the sms2acis_assembly that called it.

When in feature call mode, the boundary[1] class is in full swing. This mode is designed for when the sms2acis_boundary class is called by a sms2acis feature class, such as sms2acis_plane. Here full boundary validation and calculation is enabled, as per the processes we have just discussed. The sms2acis_boundary class receives information about the calling primitive and the names of the boundary lists. As in feature call mode the initial sms2acis_boundary instance calls the other instances that are pointed at by the names of the boundary lists provided by the calling sms2acis primitive class. Once all of this information is obtained, the Boolean curve-curve intersection algorithm is run, which results in an ACIS SAT chunk which describes the correct boundary being returned to the calling sms2acis primitive.

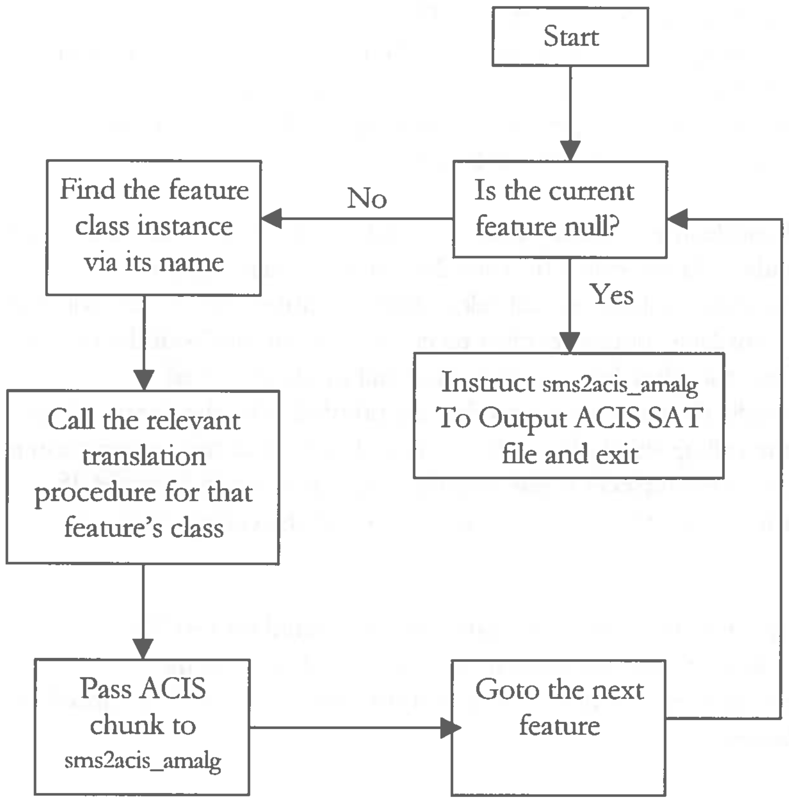
Now that we have covered how SMS boundaries are converted into ACIS representations, let us have a look at how the translation process is driven via the sms2acis_assembly class, and then onto a brief summary of the conversion actions carried out by the sms2acis primitive classes.

[1] Note the phrases boundary class and sms2acis_boundary are interchangeable!

4.6 The Top Level Translation Process

The SMS to ACIS translation process is driven by the sms2acis_assembly class. Recall that the SMS assembly is equivalent to an ACIS body, in that it is the top of the model topology tree. The SMS assembly essentially names the features that the assembly contains, plus their rotational and translational information.

The first translation action the sms2acis_assembly carries out is to check for any SMS subassemblies that are represented by child instances of sms2acis_assembly created by the translator data grabber stage. If any are found, then the second task of the "master" sms2acis_assembly class is to merge the contents of the subassemblies into its own feature listing – paying close attention to the correct accumulation of the subassembly feature transformations and rotations. Once this "master list" has been produced, then the translation process can begin in earnest. The master task loop within the sms2acis_assembly works like this:

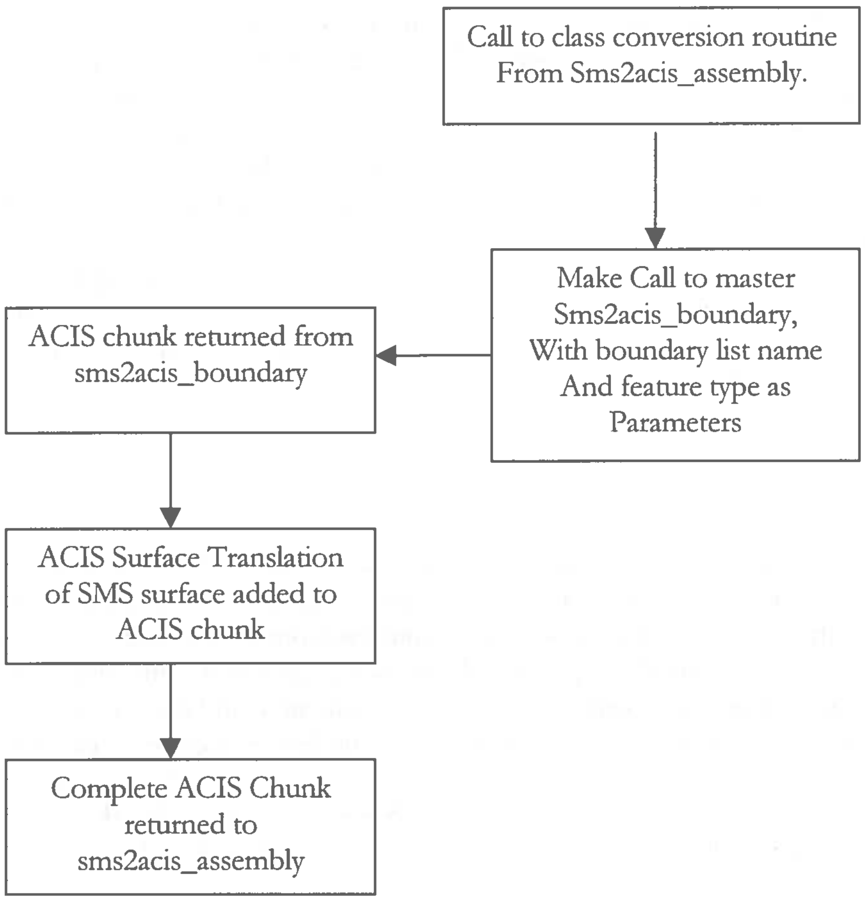

Start
Is the current feature null?
No
Find the feature class instance via its name
Yes
Instruct sms2acis_amalg To Output ACIS SAT file and exit
Call the relevant translation procedure for that feature's class
Pass ACIS chunk to sms2acis_amalg
Goto the next feature

It should be noted that when the sms2acis_assembly class calls the translation procedure in the feature's class, any valid rotational and translational information contained in the assembly is passed to the feature class as well. This ensures that all the embedded translations and rotations within the SMS hierarchy all accumulate to the final transformation value of the features in the scene correctly.

When the current feature class returns from our call, it returns along with it a chunk of an ACIS SAT file representing the ACIS equivalent of what that feature represents. For an atypical SMS surface, the ACIS return text will normally have an ACIS lump as a root feature, and will be complete for all features below that. The last stage for that feature is to aggregate it into the whole scene. This is done by passing the ACIS chunk to the sms2acis_amalg, which is responsible for the construction of the final SAT file and pointer correction for intercommunication between the chunks.

4.7 The Feature Level Translation Process

As mentioned previously in this chapter, each SMS primitive is represented by an instance of a class designed to represent an SMS primitive of that type. The main reason the system was built this way is that it allows us to optimise the translation function for each class, in order to perfect the translation result. There are 8 sms2acis primitive classes – as in the class diagram and brief class explanation list in section 4.4. The typical feature level conversion loop can be demonstrated in the following diagram:


Call to class conversion routine From Sms2acis_assembly.
Make Call to master Sms2acis_boundary, With boundary list name And feature type as Parameters
ACIS chunk returned from sms2acis_boundary
ACIS Surface Translation of SMS surface added to ACIS chunk
Complete ACIS Chunk returned to sms2acis_assembly

For more information about how specific SMS primitives may be translated into comparable ACIS forms, please see chapter 3, A Detailed View of the ACIS Modelling Engine.

Please note that the above process this is a general example of feature level translation. In particular, there are 3 main exceptions to the process detailed above. They are:

sms2acis_line

If this class is called directly from the master sms2acis_assembly, then we know that this request is part of the SMS PLACED_CURVES section in the assembly. As such, the boundary translate request is omitted and the surface translation stage is replaced by a curve translation.

sms2acis_boundary

As detailed in section 4.5 – if called from sms2acis_assembly as part of the PLACED_BOUNDRIES section of the SMS assembly, sms2acis_boundary will just return the members of the boundary – no surfaces are involved.

sms2acis_twopatch

This incorporates an extra stage in the conversion process, due to the fact that its surface will not map easily onto any ACIS primitive, and therefore we are forced to create an ACIS spline based surface to emulate its shape. The extra stage in the conversion process is the creation of a spine cage (overview in section 3.3.2), based on the x radius and y radius values from an origin point. The number of splines in the spline cage is a relation on the values of the respective radii. The number of splines in the u parameter direction is controlled by the function $numsplines = {}^{xrad}\!/_{2} \,\mathrm{mod}\, ulo2$, with the number of splines in v parameter direction utilising the same function, but with yrad as the numerator for the fraction. The result of this function is that the number of splines representing a path increases as its curvature increases – thus helping to keep the errors as low as possible.

We now know how the conversion process works – by organising the SMS primitives into different types of class, solving the assembly stage to unify all the sub assemblies, then by parsing the master assembly list and requesting each primitive instance to convert itself and return an ACIS chunk that represents the converted geometry. Sms2acis primitives may also call the boundary stage to initiate boundary normalisation and unification to assist in their own conversions. There is now one final stage to be undergone before we can output an ACIS SAT file, and that is one of amalgamation of all the ACIS SAT chunks returned to the sms2acis_assembly class into a coherent ACIS SAT file. This is this the responsibility of the sms2acis_amalg class. Lets take a closer look at the final stage of the conversion process.

4.8 ACIS Output and Pointer Amalgamation

The final problem is essentially a "cut and paste" one. Recall from previous sections that when the sms2acis_assembly receives an ACIS chunk from one of the classes it has asked to undergo a conversion operation that it does not retain it itself. Instead it passes the chunk along to the sms2acis_amalg class. This classes responsibility is to keep track of all the incoming chunks – in particular the pointers.

Recall that ACIS utilises pointers in its SAT files to link two or more items to each other. It is the way it controls its hierarchy – a child of an edge for example will always have a pointer going from parent to child – for example every edge will point at each vertex that lies on that edge, and every vertex will have a pointer to the point declaration – the actual geometry.

Since the conversion systems have no knowledge of each other or what they are operating on, when the ACIS SAT chunks come back with the root item pointer set to zero. Typically for a surface description, the root item will be an ACIS face, but for curve descriptions called directly from the assembly class, they can be as low as Edges.

At some arbitrary point, the conversion of all the items within the SMS assembly will be complete, at which point the sms2acis_assembly class will send a message to our sms2acis_amalg class instructing it to bind all the separate ACIS SAT chunks together to create a valid ACIS SAT file. Our sms2acis_amalg class will begin by constructing a three line ACIS header, which contains information which portrays a variety of useful points, such as the creation program, the write time and numbers for calculating the numerical accuracy of the SAT file, as well as the byte orders of the machine it was built on and the operating system it ran. Once it has done that, it will then start constructing some of the ACIS hierarchy that it requires, such as a body, and then go about trying to attach the ACIS SAT chunks it has to the hierarchy as it builds. When it attaches an ACIS SAT chunk to the main file – it has to perform pointer recalculation – all the pointers in the chunk are realigned to represent the lines new position in the overall file.

The above process is completed when there are no more SAT chunks to add, at which point the file has a final end mark added and is exported to disk – ready to be loaded into the ACIS modeller as needed!

[illegible] and P[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Chapter 5

Testing and Validation

5.1 Testing Introduction

Once we have developed a potential solution for our problem, we need to examine methods for measuring how successful our system is at achieving its goals. Recall that our the project goal is to establish a system that will convert geometric scenes written in the SMS language into descriptions that are suitable for inclusion into ACIS. How can we demonstrate that these goals have been achieved? It was decided that three separate approaches were needed to demonstrate not only that the conversion system that has been developed is feature complete, but that the implementation of the project is robust.

The first series of tests were unit tests that were utilised during development. The unit tests checked how the class would behave to:

- Open Boundaries
- Closed Boundaries
- Multiple Internal Boundaries

This essentially checked that the graphics primitive class was conversing to the boundary calculation code in sms2acis_booundary correctly – and alerted the developer should one of the standard tests fail.

The second series of tests test for functionality and stress test the boundary checking procedures under a variety of situations. A test schema was developed with the idea that we could try and exercise as much of the functionality that the conversion system could provide as possible. This is done by attempting to convert every sort of SMS primitive in a variety of boundary situations. For more details on this series of tests, please consult section 5.2.

The third series of tests would be increasingly complex scenes built by hand in SMS – 4 scenes in total. When combined together they could demonstrate that the conversion system was feature complete – in that it was capable of handling complex SMS scenes. The evaluation of these tests would be by eye – the scene would be rendered in both the default SMS renderer (called viewsms) and in the ACIS renderer. If the images visually matched up, then the test would be considered a success. Consult section 5.3 for more details.

The act of running all these tests together – and passing them allows us to state that the program performs correctly in all possible grammatical cases, however it is impossible to show correctness for all semantic cases – the domain of all possible semantic cases is simply too large. However we can use these test results to increase our confidence in the solutions to our project goals.

5.2 Functionality Tests

These were a succession of tests designed to try and exercise the system as much as possible. The essentially take the form of the same set of boundary tests, but run for all SMS primitives. In addition to this, the way in which the assembly is translated is checked via a small set of tests near the end of the test schema. The tests were:

Test	30/5/00 Result (Pass/Fail)
A single SMS plane with a single external boundary	PASS
A single SMS plane with a single external boundary and a single internal boundary	PASS
A singe SMS plane with multiple internal boundaries	PASS
A single SMS cylinder with a single external boundary	PASS
A single SMS cylinder with a single external boundary and a single internal boundary	PASS
A singe SMS cylinder with multiple internal boundaries	PASS
A single SMS cylindrical patch with a single external boundary	PASS
A single SMS cylindrical patch with a single external boundary and a single internal boundary	PASS
A singe SMS cylindrical patch with multiple internal boundaries	PASS
A single SMS cone with a single external boundary	PASS
A single SMS cone with a single external boundary and a single internal boundary	PASS
A singe SMS cone with multiple internal boundaries	PASS
A single SMS ellipsoid with a single external boundary	PASS
A single SMS ellipsoid with a single external boundary and a single internal boundary	PASS
A singe SMS ellipsoid with multiple internal boundaries	PASS
A single SMS torus with a single external boundary	PASS
A single SMS torus with a single external boundary and a single internal boundary	PASS
A singe SMS torus with multiple internal boundaries	PASS
A single SMS twopatch[1] with a single external boundary	PASS
A single SMS twopatch with a single external boundary and a single internal boundary	PASS
A singe SMS twopatch with multiple internal boundaries	PASS
Assembly Testing	
Multiple SMS Primitives, each with multiple boundaries	PASS
Multiple SMS Primitives, with a single SMS primitive included as a subassembly	PASS
Multiple SMS Primitives, with multiple SMS primitives included as a subassembly	PASS
Multiple SMS Primitives, with multiple SMS primitives included in multiple Subassemblies	PASS

[1] Twopatch == Doubly Curved Patch

5.3 Integration Tests

These 4 tests check for overall system functionality, and each one increases in terms of complexity, and in the number of SMS features they make use of.

The Block

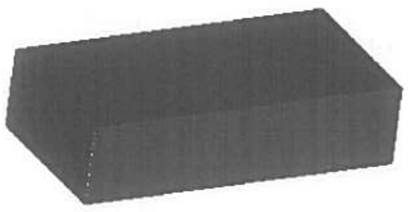
SMS Geometric Primitives Used

LINE
BOUNDARY
ROTATION – RST
ROTATION – VECTOR PAIR
SCALE
INCLUDED_POINT
PLANE
BOUNDARY_LIST
ASSEMBLY

Other SMS Features Used

PROPERTIES
UNARYPROP
SUPERTYPE
CONNECTED
BINARY PROP
VDFG

The file consists of a rectangular block that has its faces declared in subtlety different ways.


ViewSMS Rendering

ACIS Rendering

Result:- PASS

Trashcan

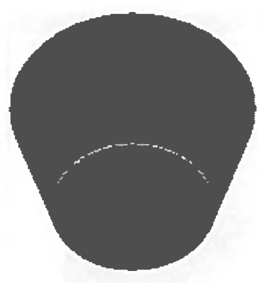
SMS Geometric Primitives Used

ELLIPSE
BOUNDARY
PLANE
BOUNDARY_LIST
TRNASLATION
ROTATION VECTOR PAIR
INCLUDED_POINT
CONE
ASSEMBLY
DEFAULT_POSITION

Other SMS Features Used

VARIABLES
EXPRESSIONS
VDFG
VARIABLE CONSTRAINTS

The file consists of a SMS model of a waste paper basket


ViewSMS Rendering

ACIS Rendering

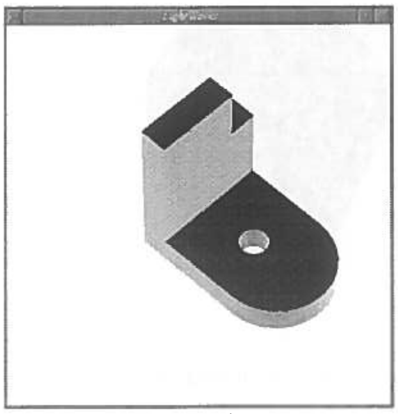
Result:- PASS

Aluminum Bracket

SMS Geometric Primitives Used	*Other SMS Features Used*
LINE	EXPRESSIONS
CIRC_ARC	VARIABLES
BOUNDARY	PARAMETERISATION
TRANSLATION	VDFG_LIST
ROTATION VECTOR PAIR	PROPERTIES
PLANE	CONNECTED
BOUDARY_LIST	BINARYPROP
INCLUDED_POINT	VIS_GROUP
ROTATION RST	TAN_GROUP
CYLINDER	
ASSEMBLY	

This file consists of an aluminum bracket with a notch at one end and a drill home in the other.

ViewSMS Rendering


ACIS Rendering

Result:- PASS

Pencil Holder

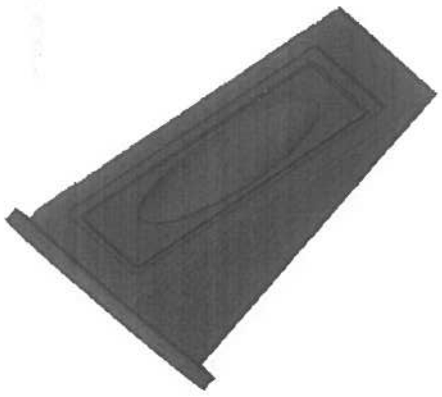
SMS Geometric Primitives Used

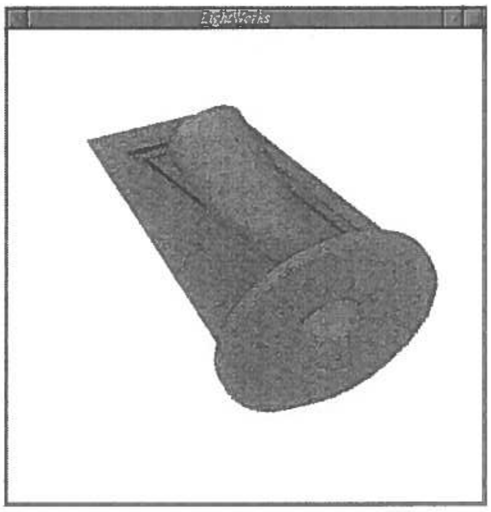
GENERICOBJECT
CIRC_ARC
ELLIPSE
LINE
BOUNDARY
TRNASLATION
ROTATION VECTOR PAIR
PLANE
BOUNDARY_LIST
INCLUDE_POINT
CYLINDER
ROTATION RST
DEFAULT_POSITION

Other SMS Features Used

EXPRESSIONS
VARIABLES
PARAMETERISATION
PROPERTIES
UNARYPROP
VALUPROP
CENTEROID
VDFG
VIS_GROUP
TAN_GROUP
CONNECT_CONSTRAINTS

This complicated file represents a detailed pencil holder. One interesting point to note is that although this object does translate – there are several features that initially look like the products of bugs. However, on closer inspection it turns out that the SMS file is compensating for bugs in ViewSMS! It is these "compensations" that cause mild display differences .


ViewSMS Rendering



ACIS Rendering

Result:- PASS

Chapter 6

Conclusion

6.1 Conclusion

The aim of this project was the design and construction of a process that would take model descriptions written in the SMS modelling language and convert them into a representation suitable for the inclusion into the ACIS SAT modelling language.

This paper describes a process that achieves the aim set out at the start of this project. The process has shown itself through various testing phases to be robust in nature, and it is believed – although not proven – that it is a complete solution for converting SMS curves and surfaces into ACIS descriptions.

This paper has also demonstrated that relatively simple mathematical techniques exist that allow us to carry out powerful transformations on 3D space – whilst maintaining continuousness in the problem domain. I believe that this paper gives a good example of why trying to solve problems algebraically is a good approach whenever possible. It may seem very easy to evaluate problems in discrete forms, such as a pixel array, but can often come at the heavy price of accuracy later – as our earlier boundary conversion algorithm in this document shows. You always run the risk of loosing too much detail – even detail that was not previously appreciated – should the decision be made to solve a problem using discrete methods. One should note that discrete methods tend to be a lot faster – this paper lends its support to that thought. However in translational systems, speed does not tend to be our primary concern, *correctness* is. It is for that reason that the Boolean Curve-Curve Intersection algorithm was chosen preferentially over the far more simple boundary fill algorithm.

6.2 Critical Points and Possible Future Work

Although the project goals have been achieved, there are two aspects of the system now that I believe could be improved. The main problem the current system suffers from is the fact that very often the ACIS models it produces are not very useful. They *are* correct, and the *are* valid, but very few ACIS operations will work on them without generating an error. The reason for this is that ACIS is a *solid modeller*. Many of its internal functions require the model to enclose a region in 3D space completely. Whilst there are a lot of SMS models that are also closed in 3D space, the sms2acis_amalg class that does the final build currently makes no guarantees that it will generate a compatible hierarchy for solid modelling – although it will always produce a model that is valid. It would be a fairly easy modification to carry out, as essentially all it would require is a refactoring of the sms2acis_amalg class.

One final modification would be support for more SMS features. Right now the conversion process still ignores a great deal of SMS functionality – namely because for this project we are only interested in the geometry. Yet ACIS has a large amount of support for 3rd party customisations and additions of the SAT language, and the very heart of ACIS itself, the API kernel can have optional components installed. It would be interesting to see just how far these modifications could go –

perhaps to the point of allowing ACIS to support of of SMS's functionality. I believe it is possible.

6.3 Bibliography

[1] Fisher, R. B., "SMS: A Suggestive Modeling System for Object Recognition", Department of Artificial Intelligence, University of Edinburgh, Research Report 298.

[2] Fisher, R. B., "Representation, Extraction and Recognition with Second-Order Topographic Surface Features", Image and Vision Computing, Vol 10 No 3, pp 156-169, April 1992.

[3] Fisher, R., Trucco, E., Fitzgibbon, A., Waite, M., Orr, M., "IMAGINE - A 3-D Vision System", Sensor Review, Vol 13, No. 2, pp 23-26, April 1993.

[4] AW. Fitzgibbon, E. Bispo, R. B. Fisher, E. Trucco "Automatic acquisition of CAD models from multiple range views", Department of Artificial Intelligence, University of Edinburgh, Research Report 689.

[5] Graphics Gems II – Numerous 1990

[6] Graphics Gems III – Numerous 1992

APPENDIX – A

SMS AND SAT DATA FROM INTEGRATION TESTS

BLOCK.SMS -> BLOCK.SAT

TRCAN.SMS -> TRCAN.SAT

BED1.SMS -> BED1.SAT

PENCIL.SMS -> PENCIL.SAT

```
/***************************************************************/
/*                         block model                        */
/*                         (block.mdl)                        */
/***************************************************************/

/*******************/
/*  define curves  */
/*******************/
#ifndef LARGEMODEL
#include "/hame/imagine2/models/generics/surface_shapes.sms"
#endif


(LINE   block_length_edge
        LENGTH 100.0
        PROPERTIES NONE)



(LINE   block_width_edge
        LENGTH 50.0
        PROPERTIES NONE)



(LINE   block_height_edge
        LENGTH 25.0
        PROPERTIES NONE)

/*******************/
/* define surfaces */
/*******************/




(BOUNDARY block_boundary_0
                block_length_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
 INTO (1,0,0) SCALE 1.0
PROPERTIES NONE         )




(BOUNDARY block_boundary_1
                block_length_edge AT TRANSLATION (0,50.0,0) ROTATION VECTOR (0,0
,1) INTO (1,0,0) SCALE 1.0
PROPERTIES NONE)




(BOUNDARY block_boundary_2
                block_width_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)




(BOUNDARY block_boundary_3
                block_width_edge AT TRANSLATION (100.0,0,0) ROTATION VECTOR (0,0
,1) INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)


(PLANE block_top_face
        BOUNDARY_LIST (block_boundary_0 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_1 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_2 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_3 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0)
        INCLUDED_POINT (25.0,25.0,0)
        PROPERTIES (
        (UNARYPROP 4000.0 < SIZE < 6000.0 PEAK 5000.0 WEIGHT 1.0)
        (SUPERTYPE plane)))


(BOUNDARY block_boundary_4
                block_length_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
 INTO (1,0,0) SCALE 1.0
PROPERTIES NONE)


(BOUNDARY block_boundary_5
                block_length_edge AT TRANSLATION (0,25,0) ROTATION VECTOR (0,0,1
) INTO (1,0,0) SCALE 1.0
PROPERTIES NONE)


(BOUNDARY block_boundary_6
                block_height_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
 INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)


(BOUNDARY block_boundary_7
                block_height_edge AT TRANSLATION (100.0,0,0) ROTATION VECTOR (0,
0,1) INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)


(PLANE block_side_face
        BOUNDARY_LIST (block_boundary_4 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_5 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_6 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_7 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0)
```

```
        INCLUDED_POINT (25.0,15.0,0)
        PROPERTIES (
        (UNARYPROP 2000.0 < SIZE < 3000.0 PEAK 2500.0 WEIGHT 1.0)
        (SUPERTYPE plane)))




(BOUNDARY block_boundary_8
                block_width_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
INTO (1,0,0) SCALE 1.0
PROPERTIES NONE)



(BOUNDARY block_boundary_9
                block_width_edge AT TRANSLATION (0,25,0) ROTATION VECTOR (0,0,1)
 INTO (1,0,0) SCALE 1.0
PROPERTIES NONE)



(BOUNDARY block_boundary_10
                block_height_edge AT TRANSLATION (0,0,0) ROTATION VECTOR (0,0,1)
 INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)



(BOUNDARY block_boundary_11
                block_height_edge AT TRANSLATION (50,0,0) ROTATION VECTOR (0,0,1
) INTO (0,1,0) SCALE 1.0
PROPERTIES NONE)


(PLANE block_end_face
        BOUNDARY_LIST (block_boundary_8 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_9 AT TRANSLATION (0,0,0) ROTATION RST (0,0
,0) SCALE 1.0
                       block_boundary_10 AT TRANSLATION (0,0,0) ROTATION RST (0,
0,0) SCALE 1.0
                       block_boundary_11 AT TRANSLATION (0,0,0) ROTATION RST (0,
0,0) SCALE 1.0)
        INCLUDED_POINT (25.0,15.0,0)
        PROPERTIES (
        (UNARYPROP 1000.0 < SIZE < 1500.0 PEAK 1250.0 WEIGHT 1.0)
        (SUPERTYPE plane)))

/**********************/
/* primary assemblies */
/**********************/



/* surfaces */
```

```
(ASSEMBLY block
        VARS NONE
        PLACED_POINTS NONE
        PLACED_CURVES NONE
        PLACED_BOUNDARIES NONE
        PLACED_SURFACES
                block_top_face AT TRANSLATION (0, 25.0, 0)
                         ROTATION RST (0,1.57079,1.57079)
                         SCALE 1.0
                block_side_face AT TRANSLATION (0,0,0) ROTATION RST (0,0,0) SCAL
E 1.0
                block_end_face AT TRANSLATION (100.0,0,0) ROTATION RST (0,1.5707
9,0) SCALE 1.0
                block_end_face AT TRANSLATION (0,0,50.0) ROTATION RST (0,1.57079
,3.14159) SCALE 1.0
        PLACED_VOLUMES NONE
        PLACED_ASSEMBLIES NONE
        VDFG_LIST (above_left above_right)
        DEFAULT_POSITION AT TRANSLATION (0,0,300) ROTATION RST (0.0,0.7,5.5)
        PROPERTIES (
        (CONNECTED block_top_face block_side_face#1)
        (CONNECTED block_top_face block_end_face#2)
        (CONNECTED block_top_face block_end_face#2)
        (CONNECTED block_end_face#1 block_side_face#1)
        (CONNECTED block_end_face#2 block_side_face#1)
        (BINARYPROP(block_top_face,block_end_face) 0.5 < ADJACENT < 1.5 PEAK 1 W
EIGHT 1.0)
        (BINARYPROP(block_top_face,block_end_face) 0.65 < RELSIZE < 0.950 PEAK 0
.8 WEIGHT 1.0)
        (BINARYPROP(block_top_face,block_end_face) -0.2 < RELORT < 0.2 PEAK 0.0
WEIGHT 1.0)
        (BINARYPROP(block_top_face,block_side_face) 0.5 < ADJACENT < 1.5 PEAK 1
WEIGHT 1.0)
        (BINARYPROP(block_top_face,block_side_face) 0.56 < RELSIZE < 0.76 PEAK 0
.66 WEIGHT 1.0)
        (BINARYPROP(block_top_face,block_side_face) -0.2 < RELORT < 0.2 PEAK 0.0
 WEIGHT 1.0)
        (BINARYPROP(block_side_face,block_end_face) 0.5 < ADJACENT < 1.5 PEAK 1
WEIGHT 1.0)
        (BINARYPROP(block_side_face,block_end_face) 0.56 < RELSIZE < 0.76 PEAK 0
.66 WEIGHT 1.0)
        (BINARYPROP(block_side_face,block_end_face) -0.2 < RELORT < 0.2 PEAK 0.0
 WEIGHT 1.0)))




/************************************/
/*  viewpoint dependent features    */

/************************************/

/* above left */
(VDFG above_left
      ASSEMBLY block
      VIS_GROUP (block_top_face block_side_face block_end_face)
      TAN_GROUP (NONE)
      PART_OBSCURED_GROUP (NONE)
      CONNECT_CONSTRAINTS (NONE)
      NEW_FEAT_CONSTRAINTS (NONE)
      POSITION_CONSTRAINTS ((VIEWER DOTPR MAP ((0,1,0)) < 0) (VIEWER DOTPR MAP (
(-1,0,0)) < 0) (VIEWER DOTPR MAP ((0,0,-1)) < 0)))
```

```
/* above right */
(VDFG above_right
      ASSEMBLY block
      VIS_GROUP (block_top_face block_side_face block_end_face)
      TAN_GROUP (NONE)
      PART_OBSCURED_GROUP (NONE)
      CONNECT_CONSTRAINTS (NONE)
      NEW_FEAT_CONSTRAINTS (NONE)
      POSITION_CONSTRAINTS ((VIEWER DOTPR MAP ((0,1,0)) < 0) (VIEWER DOTPR MAP (
(1,0,0)) < 0) (VIEWER DOTPR MAP ((0,0,-1)) < 0)))
```

```
500 0 1 0
25 SMS2ACISConverter - 5.0.3 18 ACIS 5.0.3 Solaris 24 Wed May 31 16:11:18 2000
-1 9.99999999999999547e-07 1.00000000000000036e-10
body $-1 $1 $-1 $-1 #
lump $-1 $-1 $2 $0 #
shell $-1 $-1 $-1 $3 $-1 $1 #
face $-1 $4 $5 $2 $-1 $6 forward single #
face $-1 $7 $8 $2 $-1 $9 reversed single #
loop $-1 $-1 $10 $3 #
plane-surface $-1 0 0 12.5 0 0 1 1 0 0 forward_v I I I I #
face $-1 $11 $12 $2 $-1 $13 reversed single #
loop $-1 $-1 $14 $4 #
plane-surface $-1 0 0 -12.5 0 0 1 1 0 0 forward_v I I I I #
coedge $-1 $15 $16 $17 $18 forward $5 $-1 #
face $-1 $19 $20 $2 $-1 $21 reversed single #
loop $-1 $-1 $22 $7 #
plane-surface $-1 0 -50 0 0 1 -0 -0 0 1 forward_v I I I I #
coedge $-1 $23 $24 $25 $26 forward $8 $-1 #
coedge $-1 $27 $10 $28 $29 forward $5 $-1 #
coedge $-1 $10 $27 $30 $31 forward $5 $-1 #
coedge $-1 $32 $33 $10 $18 reversed $34 $-1 #
edge $-1 $35 -50 $36 50 $17 $37 forward 7 unknown #
face $-1 $38 $39 $2 $-1 $40 reversed single #
loop $-1 $-1 $41 $11 #
plane-surface $-1 -25 0 0 1 0 0 0 0 -1 forward_v I I I I #
coedge $-1 $42 $30 $43 $44 forward $12 $-1 #
coedge $-1 $45 $14 $42 $46 forward $8 $-1 #
coedge $-1 $14 $45 $47 $48 forward $8 $-1 #
coedge $-1 $33 $32 $14 $26 reversed $34 $-1 #
edge $-1 $49 -50 $50 50 $25 $51 forward 7 unknown #
coedge $-1 $16 $15 $52 $53 forward $5 $-1 #
coedge $-1 $54 $55 $15 $29 reversed $39 $-1 #
edge $-1 $36 -25 $56 25 $28 $57 forward 7 unknown #
coedge $-1 $22 $58 $16 $31 reversed $12 $-1 #
edge $-1 $59 -25 $35 25 $30 $60 forward 7 unknown #
coedge $-1 $25 $17 $58 $61 forward $34 $-1 #
coedge $-1 $17 $25 $54 $62 reversed $34 $-1 #
loop $-1 $-1 $32 $38 #
vertex $-1 $18 $63 #
vertex $-1 $18 $64 #
straight-curve $-1 25 0 12.5 0 1 0 I I #
face $-1 $-1 $34 $2 $-1 $65 reversed single #
loop $-1 $-1 $54 $19 #
plane-surface $-1 0 50 0 0 -1 0 0 0 -1 forward_v I I I I #
coedge $-1 $66 $52 $55 $67 forward $20 $-1 #
coedge $-1 $58 $22 $23 $46 reversed $12 $-1 #
coedge $-1 $52 $66 $22 $44 reversed $20 $-1 #
edge $-1 $59 -12.5 $68 12.5 $43 $69 forward 7 unknown #
coedge $-1 $24 $23 $66 $70 forward $8 $-1 #
edge $-1 $50 -25 $68 25 $42 $71 forward 7 unknown #
coedge $-1 $55 $54 $24 $48 reversed $39 $-1 #
edge $-1 $72 -25 $49 25 $47 $73 forward 7 unknown #
vertex $-1 $26 $74 #
vertex $-1 $61 $75 #
straight-curve $-1 25 0 -12.5 0 -1 0 I I #
coedge $-1 $41 $43 $27 $53 reversed $20 $-1 #
edge $-1 $56 -50 $59 50 $52 $76 forward 7 unknown #
coedge $-1 $47 $28 $33 $62 forward $39 $-1 #
coedge $-1 $28 $47 $41 $67 reversed $39 $-1 #
vertex $-1 $29 $77 #
straight-curve $-1 0 50 12.5 -1 0 0 I I #
coedge $-1 $30 $42 $32 $61 reversed $12 $-1 #
vertex $-1 $53 $78 #
straight-curve $-1 0 -50 12.5 1 0 0 I I #
edge $-1 $35 -12.5 $50 12.5 $32 $79 forward 7 unknown #
edge $-1 $36 -12.5 $49 12.5 $33 $80 forward 7 unknown #
point $-1 25 -50 12.5 #
point $-1 25 50 12.5 #
plane-surface $-1 25 0 0 -1 0 0 0 -0 1 forward_v I I I I #
coedge $-1 $43 $41 $45 $70 reversed $20 $-1 #
```

```
edge $-1 $56 -12.5 $72 12.5 $55 $81 forward 7 unknown #
vertex $-1 $70 $82 #
straight-curve $-1 -25 -50 0 0 0 -1 I I #
edge $-1 $68 -50 $72 50 $66 $83 forward 7 unknown #
straight-curve $-1 0 -50 -12.5 -1 0 0 I I #
vertex $-1 $48 $84 #
straight-curve $-1 0 50 -12.5 1 0 0 I I #
point $-1 25 50 -12.5 #
point $-1 25 -50 -12.5 #
straight-curve $-1 -25 0 12.5 0 -1 0 I I #
point $-1 -25 50 12.5 #
point $-1 -25 -50 12.5 #
straight-curve $-1 25 -50 0 0 0 -1 I I #
straight-curve $-1 25 50 0 0 0 -1 I I #
straight-curve $-1 -25 50 0 0 0 -1 I I #
point $-1 -25 -50 -12.5 #
straight-curve $-1 -25 0 -12.5 0 1 0 I I #
point $-1 -25 50 -12.5 #
End-of-ACIS-data
```

```
/**********************************************************/
/*  assembly          :  trcan                            */
/**********************************************************/
/*                                                        */
/*  for model         :  robot                            */
/*  variables         :  h (=height),                     */
/*                       br (=big radius),                */
/*                       sr (=small radius)               */
/*                                                        */
/**********************************************************/
/*                 by SEY , Version 17.7.87               */
/**********************************************************/

/*******************/
/* define curves   */
/*******************/
#define sr 20
#define br 30
#define h 40
(ELLIPSE trcan_circ1 XRADIUS sr YRADIUS sr ENDPOINTS (0,sr,0) (0,sr,0))
(ELLIPSE trcan_circ2 XRADIUS br YRADIUS br ENDPOINTS (0,br,0) (0,br,0))

/*******************/
/* define surfaces */
/*******************/

(BOUNDARY trcan_bottom_boundary trcan_circ1 AT ORIGIN)

(PLANE trcan_bot
       BOUNDARY_LIST (trcan_bottom_boundary AT ORIGIN)
       INCLUDED_POINT (0, 0, 0))

(BOUNDARY trcan_cone_sheet1
          trcan_circ1 AT TRANSLATION  (((h * sr) / (br - sr)), 0, 0)
                         ROTATION VECTOR (0,0,1) INTO (1,0,0))
(BOUNDARY trcan_cone_sheet2
          trcan_circ2  AT TRANSLATION  ((((h * sr) / (br - sr)) + h), 0, 0)
                         ROTATION VECTOR (0,0,1) INTO (1,0,0))
(CONE trcan_surf
      RADIUS_RATE((br - sr) * h) / ((h * h) - (((br - sr) * (br - sr)) / 4))
      BOUNDARY_LIST (trcan_cone_sheet1 AT ORIGIN
                     trcan_cone_sheet2 AT ORIGIN)
      INCLUDED_POINT ((h * sr) / (br - sr) + h/2, (sr+br)/2, 0))

/*******************/
/* define volumes  */
/*******************/
(STICK trcan_vol
       LENGTH       h
       CROSS_RADIUS (br + sr) / 2
       BEND_RADIUS  0)

(DENT trcan_dent
      DEPTH  h - (0.1 * h)
      MAJOR_RADIUS (((br + sr) / 2) - 0.1 * h)
      MINOR_RADIUS (((br + sr) / 2) - 0.1 * h))

/**************************/
/*  primary assembly      */
/**************************/
(ASSEMBLY trcan
//        VARS (h   (DEFAULT_VALUE 20.0)
//              br  (DEFAULT_VALUE 10.0)
//              sr  (DEFAULT_VALUE 7.5))
          PLACED_CURVES
          trcan_circ1 AT TRANSLATION  (0,-10, 0) ROTATION VECTOR (0,0,1) INTO (0
,1,0)
          trcan_circ2 AT TRANSLATION  (0, h - 10, 0) ROTATION VECTOR (0,0,1) INT
O (0,1,0)
```

```
          PLACED_SURFACES
          trcan_bot AT TRANSLATION  (0, 0 - 10, 0) ROTATION VECTOR (0,0,1) INTO
(0,1,0)
          trcan_bot AT TRANSLATION  (0, 0.1 - 10, 0) ROTATION VECTOR (0,0,1) INT
O (0,-1,0)
          trcan_surf AT TRANSLATION  (0, (0 - 10 - ((h * sr) / (br - sr))), 0)
                        ROTATION VECTOR (1,0,0) INTO (0,1,0)
          trcan_surf  AT TRANSLATION  (0, (0.1 - 10 - ((h * sr) / (br - sr))), 0
)
                         ROTATION VECTOR (1,0,0) INTO (0,1,0)

          PLACED_VOLUMES
          trcan_vol AT TRANSLATION  (0, 0 - 10, 0) ROTATION VECTOR (1,0,0) INTO
(0,1,0)
          trcan_dent AT TRANSLATION  (0, h - 10, 0) ROTATION VECTOR (1,0,0) INTO
 (0,-1,0)

          VDFG_LIST (trcan_vg_side_below trcan_vg_side trcan_vg_side_above trcan
_vg_top)
          DEFAULT_POSITION AT TRANSLATION  (0, 0, 200) ROTATION RST (0, -0.5, HA
LFPI))


/**********************************/
/* viewpoint dependent features */
/**********************************/
(VDFG trcan_vg_side_below
     ASSEMBLY trcan
     VIS_GROUP (trcan_vol trcan_surf#1 trcan_bot#1 trcan_circ1 trcan_circ2)
     TAN_GROUP ( NONE )
     PART_OBSCURED_GROUP (trcan_circ2)
     CONNECT_CONSTRAINTS ( NONE )
     NEW_FEAT_CONSTRAINTS (VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan1
                                      CONSTRAINTS((trcan_tan1 LEFT PROJECT(0,1,
0)/*???*/))
                                      SURFACE trcan_surf#1
                           VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan2
                                      CONSTRAINTS((trcan_tan2 RIGHT PROJECT(0,1
,0)/*???*/))
                                      SURFACE trcan_surf#1)
     POSITION_CONSTRAINTS ((VIEWER DOTPR MAP (0, 1, 0) > 0     )
                           (VIEWER DOTPR MAP (0, 1, 0) < 0.81 )))

(VDFG trcan_vg_side
     ASSEMBLY trcan
     VIS_GROUP (trcan_vol trcan_surf#1 trcan_surf#2 trcan_circ1 trcan_circ2)
     TAN_GROUP ( NONE )
     PART_OBSCURED_GROUP (trcan_circ2 trcan_surf#2)
     CONNECT_CONSTRAINTS ( NONE )
     NEW_FEAT_CONSTRAINTS (VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan1
                                      CONSTRAINTS ((trcan_tan1 LEFT PROJECT(0,1
,0)/*???*/))
                                      SURFACE trcan_surf#1
                           VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan2
                                      CONSTRAINTS ((trcan_tan2 RIGHT PROJECT(0,
1,0)/*???*/))
                                      SURFACE trcan_surf#1
                           VPD_OCCLBND trcan_circ2  BACKGROUND (trcan_surf#2))
     POSITION_CONSTRAINTS ( (VIEWER DOTPR MAP((0, -1,0)) > 0     )
                            (VIEWER DOTPR MAP((0, -1,0)) < 0.81 )))


(VDFG trcan_vg_side_above
     ASSEMBLY trcan
     VIS_GROUP (trcan_vol trcan_surf#1 trcan_surf#2 trcan_bot#2
                        trcan_circ1 trcan_circ2)
     TAN_GROUP ( NONE )
     PART_OBSCURED_GROUP (trcan_circ1 trcan_surf#2 trcan_bot#2)
     CONNECT_CONSTRAINTS (NONE)
     NEW_FEAT_CONSTRAINTS (VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan1
                                            CONSTRAINTS ((trcan_tan1 LEFT PROJECT(0,1
,0)/*???*/))
                                            SURFACE trcan_surf#1
                                VPD_TANBND BOUNDARY trcan_surf#1->trcan_tan2
                                            CONSTRAINTS ((trcan_tan2 RIGHT PROJECT(0,
1,0)/*???*/))
                                            SURFACE trcan_surf#1
                                VPD_OCCLBND trcan_circ2  BACKGROUND (trcan_surf#2 tr
can_bot#2))
      POSITION_CONSTRAINTS ((VIEWER DOTPR MAP((0, -1,0)) > 0.81 )
                             (VIEWER DOTPR MAP((0, -1,0)) < 0.99 )))

(VDFG trcan_vg_top
      ASSEMBLY trcan
      VIS_GROUP (trcan_vol trcan_dent trcan_surf#2 trcan_bot#2 trcan_circ1 trcan
_circ2 )
      TAN_GROUP ( NONE )
      PART_OBSCURED_GROUP (NONE)
      CONNECT_CONSTRAINTS ( NONE )
      NEW_FEAT_CONSTRAINTS ( NONE )
      POSITION_CONSTRAINTS ((VIEWER DOTPR MAP((0,-1,0)) > 0.99)))


/*********************************/
/*   variable constraints       */
/*********************************/

(CONSTRAINT  ((     15  < h ) & (h  < 30))               ASSEMBLY trcan)
(CONSTRAINT  (((h / 3)  < br) & (br < (h * 3 / 4)))     ASSEMBLY trcan)
(CONSTRAINT  (((br * 2 / 3) < sr) & (sr < (0.9 * br)))  ASSEMBLY trcan)
```

```
500 0 1 0
25 SMS2ACISConverter - 5.0.3 18 ACIS 5.0.3 Solaris 24 Wed May 31 16:16:10 2000
-1 9.9999999999999547e-07 1.0000000000000036e-10
body $-1 $1 $-1 $-1 #
lump $-1 $-1 $2 $0 #
shell $-1 $-1 $-1 $3 $-1 $1 #
face $-1 $4 $5 $2 $-1 $6 reversed single #
face $-1 $7 $8 $2 $-1 $9 reversed single #
loop $-1 $10 $11 $3 #
cone-surface $-1 0 0 0.10000000000000056 0 0 1 25 0 0 1 I I 0.2425356250363329
691 0.9701425001453318764 25 forward I I I I #
face $-1 $12 $13 $2 $-1 $14 forward single #
loop $-1 $-1 $15 $4 #
plane-surface $-1 0 0 -19.899999999999858 0 0 -1 -1 0 0 forward_v I I I I #
loop $-1 $-1 $16 $3 #
coedge $-1 $11 $11 $17 $18 reversed $5 $-1 #
face $-1 $19 $20 $2 $-1 $21 forward single #
loop $-1 $22 $23 $7 #
cone-surface $-1 0 0 0 0 0 1 25 0 0 1 I I 0.2425356250363329691 0.97014250014533
18764 25 forward I I I I #
coedge $-1 $15 $15 $16 $24 reversed $8 $-1 #
coedge $-1 $16 $16 $15 $24 forward $10 $-1 #
coedge $-1 $17 $17 $11 $18 forward $25 $-1 #
edge $-1 $26 -3.141592653589793116 $26 3.141592653589793116 $17 $27 forward 7 un
known #
face $-1 $-1 $25 $2 $-1 $28 forward single #
loop $-1 $-1 $29 $12 #
plane-surface $-1 0 0 -20 0 0 -1 -1 0 0 forward_v I I I I #
loop $-1 $-1 $30 $7 #
coedge $-1 $23 $23 $29 $31 reversed $13 $-1 #
edge $-1 $32 0 $32 6.283185307179586232 $15 $33 forward 7 unknown #
loop $-1 $34 $17 $19 #
vertex $-1 $18 $35 #
ellipse-curve $-1 0 0 20 -0 -0 -1 29.975000000000142 0 0 1 I I #
plane-surface $-1 0 0 20 0 0 1 1 0 0 forward_v I I I I #
coedge $-1 $29 $29 $23 $31 forward $20 $-1 #
coedge $-1 $30 $30 $36 $37 reversed $22 $-1 #
edge $-1 $38 0 $38 6.283185307179586232 $29 $39 forward 7 unknown #
vertex $-1 $24 $40 #
ellipse-curve $-1 0 0 -19.899999999999858 0 0 -1 20 0 0 1 I I #
loop $-1 $-1 $36 $19 #
point $-1 -29.975000000000142 3.670878780444419143e-15 20 #
coedge $-1 $36 $36 $30 $37 forward $34 $-1 #
edge $-1 $41 0 $41 6.283185307179586232 $36 $42 forward 7 unknown #
vertex $-1 $31 $43 #
ellipse-curve $-1 0 0 -20 0 0 -1 20 0 0 1 I I #
point $-1 20 0 -19.899999999999858 #
vertex $-1 $37 $44 #
ellipse-curve $-1 0 0 20 0 0 1 30 0 0 1 I I #
point $-1 20 0 -20 #
point $-1 30 0 20 #
End-of-ACIS-data
```

```
 //  object.sms
//  sms model of an object


//_________ defenition of Transformations________________

//__________defenition of a rotation_______________

#define angle_axis(a,x,y,z) QUATERNION  (COS((a)/2),   \
                                         SIN((a)/2)*x,  \
                                         SIN((a)/2)*y,  \
                                         SIN((a)/2)*z)



//__________ defenition of features______________________

//__________ defenition of lines___________


(LINE  line10  LENGTH    10)
(LINE  line20  LENGTH    20)
(LINE  line30  LENGTH    30)
(LINE  line40  LENGTH    40)
(LINE  line50  LENGTH    50)
(LINE  line60  LENGTH    60)


//__________ defenition of circular arcs___________


(CIRC_ARC   circle10     RADIUS  5  ANGLE  2*PI)
(CIRC_ARC   circle_arc20 RADIUS  20  ANGLE  PI)

//_________ defenition of the plane P1_____________

(BOUNDARY  boundface1

line60  AT


TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (-1,0,0)
SCALE 1


line40  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO  (0,1,0)
SCALE 1


line40  AT
TRANSLATION  (0,40,0)  ROTATION VECTOR  (0,0,1) INTO  (-1,0,0)
SCALE 1

line30  AT
TRANSLATION  (-60,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1


line20  AT
TRANSLATION  (-40,30,0)  ROTATION VECTOR  (0,0,1) INTO (-1,0,0)
SCALE 1

line10 AT
TRANSLATION  (-40,30,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1)


(PLANE    face1
          BOUNDARY_LIST ( boundface1  AT
          ORIGIN
          SCALE  1)
          INCLUDED_POINT (-10,10,0) )



//_________ defenition of the plane P2 _____________


(BOUNDARY  boundface2

line50  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line40  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

line30  AT
TRANSLATION  (0,40,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line30  AT
TRANSLATION  (50,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1


line20  AT
TRANSLATION  (30,30,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line10 AT
TRANSLATION  (30,30,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1)


(PLANE    face2
          BOUNDARY_LIST ( boundface2  AT ORIGIN
          SCALE  1)
          INCLUDED_POINT (10,10,0) )


//_________ defenition of the plane P3_____________


(BOUNDARY  boundface3


line30  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

line30  AT
TRANSLATION  (-10,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

line10  AT
TRANSLATION  (-10,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (-10,30,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1)
```

```
(PLANE   face3
         BOUNDARY_LIST ( boundface3  AT
         ORIGIN
         SCALE  1)
         INCLUDED_POINT (-5,5,0) )



//_________ defenition of the plane P4_____________


(BOUNDARY  boundface4


line10  AT
TRANSLATION  (-10,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (-10,10,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1


line10  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

line10  AT
TRANSLATION  (-10,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1)

(PLANE   face4
         BOUNDARY_LIST ( boundface4  AT   ORIGIN
         SCALE  1)
         INCLUDED_POINT (-5,5,0) )


//_________ defenition of the plane P4bis_____________


(BOUNDARY  boundface4bis


line10  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR   (0,0,1) INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (0,20,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line20  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR   (0,0,1) INTO (0,1,0)
SCALE 1

line20  AT
TRANSLATION  (10,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1)

(PLANE   face4bis
         BOUNDARY_LIST ( boundface4bis  AT ORIGIN SCALE  1)
         INCLUDED_POINT (5,5,0) )


//_________ defenition of the plane P5_____________
```

```
(BOUNDARY  boundface5

line30  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line30  AT
TRANSLATION  (0,40,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line40  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

circle_arc20 AT
TRANSLATION   (30,20,0) ROTATION RST (0,0,0)
SCALE 1


circle10 AT
TRANSLATION   (30,20,0) ROTATION angle_axis(0,0,0,1)
SCALE 1)

(PLANE   face5
         BOUNDARY_LIST ( boundface5  AT ORIGIN SCALE  1)
         INCLUDED_POINT (10,10,0) )


//_________ defenition of the plane P6_____________

(BOUNDARY  boundface6

line20  AT
TRANSLATION  (-20,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line20  AT
TRANSLATION  (-20,40,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line40  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

circle_arc20 AT
TRANSLATION   (-20,20,0) ROTATION angle_axis(PI,0,0,1)
SCALE 1

circle10 AT
TRANSLATION   (-20,20,0) ROTATION angle_axis(0,0,0,1)
SCALE 1)

(PLANE   face6
         BOUNDARY_LIST ( boundface6  AT
         ORIGIN
         SCALE  1)
         INCLUDED_POINT (-10,10,0) )


//_________ defenition of the plane P7_____________

(BOUNDARY  boundface7

line60  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,-1,0)
SCALE 1

line30  AT
```

```
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line50  AT
TRANSLATION  (10,-10,0)  ROTATION VECTOR  (0,0,1) INTO (0,-1,0)
SCALE 1

line20  AT
TRANSLATION  (10,-10,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (0,-60,0)  ROTATION VECTOR  (0,0,1)  INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (30,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,-1,0)
SCALE 1)


(PLANE    face7
          BOUNDARY_LIST ( boundface7  AT
          ORIGIN
          SCALE  1)
          INCLUDED_POINT (5,-5,0) )



//_________ defenition of the plane P8_____________

(BOUNDARY  boundface8

line40  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (0,1,0)
SCALE 1

line30  AT
TRANSLATION  (0,0,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line30  AT
TRANSLATION  (10,10,0)  ROTATION VECTOR  (0,0,1)  INTO (0,1,0)
SCALE 1

line20  AT
TRANSLATION  (10,10,0)  ROTATION VECTOR  (0,0,1)  INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (0,40,0)  ROTATION VECTOR  (0,0,1)  INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION  (30,0,0)  ROTATION VECTOR  (0,0,1)  INTO (0,1,0)
SCALE 1)


(PLANE    face8
          BOUNDARY_LIST ( boundface8  AT ORIGIN SCALE  1)
          INCLUDED_POINT (5,5,0) )


//_________ defenition of the cylinder C1_____________

(BOUNDARY  boundcyl1

circle10 AT
TRANSLATION   (0,0,0) ROTATION   angle_axis(PI/2,0,1,0)
```

```
SCALE 1

circle10 AT
TRANSLATION   (10,0,0) ROTATION   angle_axis(PI/2,0,1,0)
SCALE 1 )


(CYLINDER cyl1
           YRADIUS  10
           ZRADIUS  10
           BOUNDARY_LIST ( boundcyl1  AT ORIGIN SCALE  1)
           INCLUDED_POINT (5,0,10) )


//_________ defenition of the cylinder C2_____________

(BOUNDARY  boundcyl2

circle_arc20  AT
TRANSLATION  (0,0,0) ROTATION   angle_axis(-PI/2,0,1,0)
SCALE 1

circle_arc20  AT
TRANSLATION   (10,0,0) ROTATION   angle_axis(-PI/2,0,1,0)
SCALE 1

line10  AT
TRANSLATION    (0,20,0)  ROTATION VECTOR  (0,0,1) INTO (1,0,0)
SCALE 1

line10  AT
TRANSLATION   (0,-20,0)  ROTATION VECTOR  (0,0,1) INTO  (1,0,0)
SCALE 1)

(CYLINDER cyl2
           YRADIUS  -20
           ZRADIUS  -20
           BOUNDARY_LIST ( boundcyl2  AT ORIGIN SCALE  1)
           INCLUDED_POINT (5,0,20) )


(ASSEMBLY protobject

  PLACED_SURFACES

   face1 AT
   TRANSLATION  (0,0,0) ROTATION angle_axis(PI/2,0,1,0)
   SCALE 1

   face2 AT
   TRANSLATION  (10,0,10) ROTATION angle_axis(PI/2,0,-1,0)
   SCALE 1

   face3 AT
   TRANSLATION  (0,0,60) ROTATION angle_axis(PI,0,1,0)
   SCALE 1

   face4 AT
   TRANSLATION  (0,30,40) ROTATION   angle_axis(PI,0,1,0)
   SCALE 1

   face4bis  AT
   TRANSLATION  (0,30,40) ROTATION  angle_axis(PI/2,1,0,0)
   SCALE 1

   face5 AT
   TRANSLATION  (0,0,0) ROTATION   angle_axis(0,1,0,0)
   SCALE 1

   face6 AT
```

```
TRANSLATION  (10,0,10) ROTATION  angle_axis(PI,0,1,0)
SCALE 1

face7 AT
TRANSLATION  (0,0,0) ROTATION  angle_axis(PI/2,-1,0,0)
SCALE 1

face8 AT
TRANSLATION  (0,40,0) ROTATION  angle_axis(PI/2,1,0,0)
SCALE 1

cyl1 AT
TRANSLATION  (30,20,0) ROTATION   angle_axis(-PI/2,0,1,0)
SCALE 1

cyl2 AT
TRANSLATION  (30,20,10) ROTATION   angle_axis(PI/2,0,1,0)
SCALE 1


 VDFG_LIST (obj_top)


  PROPERTIES(

  /*connectivity between faces */

 (CONNECTED face1  face3)
 (CONNECTED face1  face7)
 (CONNECTED face1  face8)
 (CONNECTED face1  face4)
 (CONNECTED face1  face4bis)
 (CONNECTED face1  face5)


 (CONNECTED face2  face3)
 (CONNECTED face2  face7)
 (CONNECTED face2  face8)
 (CONNECTED face2  face4)
 (CONNECTED face2  face4bis)
 (CONNECTED face2  face6)

 (CONNECTED face5  face7)
 (CONNECTED face5  face8)
 (CONNECTED face5  cyl1)
 (CONNECTED face5  cyl2)

 (CONNECTED face6  face7)
 (CONNECTED face6  face8)
 (CONNECTED face6  cyl1)
 (CONNECTED face6  cyl2)

 (CONNECTED face7  cyl2)
 (CONNECTED face8  cyl2)


 /* pairwise properties */
 (BINARYPROP(face1,face3)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face1,face4)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face1,face4bis)  0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face1,face7)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face1,face8)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face1,face5)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)

 (BINARYPROP(face2,face3)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face2,face7)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face2,face8)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face2,face4)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face2,face4bis)  0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
 (BINARYPROP(face2,face6)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
```

```
    (BINARYPROP(face5,face7)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face5,face8)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face5,cyl1)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face5,cyl2)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face6,face7)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face6,face8)     0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face6,cyl1)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face6,cyl2)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face7,cyl2)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face8,cyl1)      0.5 < ADJACENT < 1.5 PEAK 1 WEIGHT 1.0)


   /* Parallel surfaces */

   (BINARYPROP(face1,face2)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)

   (BINARYPROP(face3,face5)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)
   (BINARYPROP(face3,face4)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)
   (BINARYPROP(face3,face6)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)

   (BINARYPROP(face5,face5)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)

   (BINARYPROP(face4bis,face7) 0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)
   (BINARYPROP(face4bis,face8) 0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)

   (BINARYPROP(face7,face8)     0.5 < PARALLEL < 1.5 PEAK 1 WEIGHT 1.0)


  /* Orthogonal surfaces */

    (BINARYPROP(face1,face3)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face1,face4)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face1,face5)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face1,face6)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face1,face7)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face1,face8)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face2,face3)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face3,face4)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face4,face5)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face5,face6)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face6,face7)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face7,face8)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face3,face7)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face3,face8)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face5,face7)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face5,face8)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)

    (BINARYPROP(face6,face7)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)
    (BINARYPROP(face6,face8)    0.5 < PERPENDICULAR  < 1.5 PEAK 1 WEIGHT 1.0)))


(VDFG obj_top
 ASSEMBLY  protobject
 VIS_GROUP( face3 face4 face6)
 TAN_GROUP( face1 face2 face4bis face7 face8  cyl2)
 PART_OBSCURED_GROUP (NONE)
 CONNECT_CONSTRAINTS (NONE)
 NEW_FEAT_CONSTRAINTS (NONE)
 POSITION_CONSTRAINTS (NONE)  )
```

```
500 0 1 0
25 SMS2ACISConverter - 5.0.3 18 ACIS 5.0.3 Solaris 24 Wed May 31 17:31:21 2000
-1 9.9999999999999547e-07 1.000000000000000036e-10
body $-1 $1 $-1 $2 #
lump $-1 $-1 $3 $0 #
transform $-1 1 0 0 0 1 0 0 0 1 35 0 5 1 no_rotate no_reflect no_shear #
shell $-1 $-1 $-1 $4 $-1 $1 #
face $-1 $5 $6 $3 $-1 $7 reversed single #
face $-1 $8 $9 $3 $-1 $10 reversed single #
loop $-1 $11 $12 $4 #
cone-surface $-1 -10 0 -5 0 0 1 5 0 0 1 I I 0 1 5 forward I I I I #
face $-1 $13 $14 $3 $-1 $15 forward single #
loop $-1 $-1 $16 $5 #
plane-surface $-1 -30 0 25 -1 0 0 0 0 1 forward_v I I I I #
loop $-1 $-1 $17 $4 #
coedge $-1 $12 $12 $18 $19 reversed $6 $-1 #
face $-1 $20 $21 $3 $-1 $22 forward single #
loop $-1 $-1 $23 $8 #
plane-surface $-1 -35 10 45 0 1 0 0 0 1 forward_v I I I I #
coedge $-1 $24 $25 $26 $27 reversed $9 $-1 #
coedge $-1 $17 $17 $28 $29 reversed $11 $-1 #
coedge $-1 $18 $18 $12 $19 forward $30 $-1 #
edge $-1 $31 -3.141592653589793116 $31 3.141592653589793116 $18 $32 forward 7 un
known #
face $-1 $33 $34 $3 $-1 $35 forward single #
loop $-1 $-1 $36 $13 #
plane-surface $-1 -35 15 35 0 0 1 1 0 0 forward_v I I I I #
coedge $-1 $37 $38 $39 $40 reversed $14 $-1 #
coedge $-1 $41 $16 $42 $43 reversed $9 $-1 #
coedge $-1 $16 $44 $45 $46 forward $9 $-1 #
coedge $-1 $47 $48 $16 $27 forward $49 $-1 #
edge $-1 $50 -20 $51 20 $26 $52 forward 7 unknown #
coedge $-1 $28 $28 $17 $29 forward $53 $-1 #
edge $-1 $54 -3.141592653589793116 $54 3.141592653589793116 $28 $55 forward 7 un
known #
loop $-1 $49 $18 $56 #
vertex $-1 $19 $57 #
ellipse-curve $-1 -10 0 5 -0 -0 -1 5 0 0 1 I I #
face $-1 $58 $59 $3 $-1 $60 reversed single #
loop $-1 $-1 $61 $20 #
plane-surface $-1 -35 0 55 0 0 1 1 0 0 forward_v I I I I #
coedge $-1 $62 $63 $41 $64 reversed $21 $-1 #
coedge $-1 $65 $23 $63 $66 forward $14 $-1 #
coedge $-1 $23 $65 $61 $67 reversed $14 $-1 #
coedge $-1 $44 $41 $23 $40 forward $9 $-1 #
edge $-1 $68 0 $69 20 $39 $70 forward 7 unknown #
coedge $-1 $39 $24 $36 $64 forward $9 $-1 #
coedge $-1 $71 $72 $24 $43 forward $73 $-1 #
edge $-1 $74 -10 $50 20 $24 $75 forward 7 unknown #
coedge $-1 $25 $39 $76 $77 reversed $9 $-1 #
coedge $-1 $78 $79 $25 $46 reversed $80 $-1 #
edge $-1 $81 -30 $51 20 $25 $82 forward 7 unknown #
coedge $-1 $83 $26 $79 $84 forward $49 $-1 #
coedge $-1 $26 $83 $71 $85 forward $49 $-1 #
loop $-1 $-1 $83 $56 #
vertex $-1 $85 $86 #
vertex $-1 $27 $87 #
straight-curve $-1 -30 0 5 0 -1 0 I I #
loop $-1 $88 $28 $89 #
vertex $-1 $29 $90 #
ellipse-curve $-1 -10 0 -5 -0 -0 1 5 0 0 1 I I #
face $-1 $91 $30 $3 $-1 $92 forward single #
point $-1 -15 6.123233995736766282e-16 5 #
face $-1 $93 $80 $3 $-1 $94 reversed single #
loop $-1 $-1 $95 $33 #
plane-surface $-1 -40 0 25 1 0 0 0 0 -1 forward_v I I I I #
coedge $-1 $96 $76 $38 $67 forward $34 $-1 #
coedge $-1 $97 $36 $72 $98 reversed $21 $-1 #
coedge $-1 $36 $97 $37 $66 reversed $21 $-1 #
edge $-1 $74 -12.5 $68 -2.5 $41 $99 forward 7 unknown #
```

```
coedge $-1 $38 $37 $95 $100 reversed $14 $-1 #
edge $-1 $68 -5 $101 5 $37 $102 forward 7 unknown #
edge $-1 $69 -5 $103 5 $61 $104 forward 7 unknown #
vertex $-1 $64 $105 #
vertex $-1 $67 $106 #
straight-curve $-1 -30 10 35 0 0 1 I I #
coedge $-1 $107 $42 $48 $85 reversed $73 $-1 #
coedge $-1 $42 $108 $62 $98 forward $73 $-1 #
loop $-1 $-1 $107 $93 #
vertex $-1 $43 $109 #
straight-curve $-1 -30 20 25 0 0 -1 I I #
coedge $-1 $61 $110 $44 $77 forward $34 $-1 #
edge $-1 $81 -20 $69 10 $44 $111 forward 7 unknown #
coedge $-1 $112 $45 $110 $113 reversed $80 $-1 #
coedge $-1 $45 $114 $47 $84 reversed $80 $-1 #
loop $-1 $-1 $115 $58 #
vertex $-1 $77 $116 #
straight-curve $-1 -30 -20 25 0 0 -1 I I #
coedge $-1 $48 $47 $117 $118 forward $49 $-1 #
edge $-1 $51 -15 $119 15 $79 $120 forward 7 unknown #
edge $-1 $121 -15 $50 15 $71 $122 forward 7 unknown #
point $-1 -30 20 5 #
point $-1 -30 -20 5 #
loop $-1 $-1 $123 $89 #
face $-1 $-1 $53 $3 $-1 $124 forward single #
point $-1 -15 -6.1232339957367662 82e-16 -5 #
face $-1 $89 $125 $3 $-1 $126 forward single #
plane-surface $-1 0 0 5 0 0 1 1 0 0 forward_v I I I I #
face $-1 $56 $73 $3 $-1 $127 reversed single #
plane-surface $-1 -15 -20 0 0 1 0 0 0 1 forward_v I I I I #
coedge $-1 $128 $129 $65 $100 forward $59 $-1 #
coedge $-1 $110 $61 $129 $130 forward $34 $-1 #
coedge $-1 $63 $62 $128 $131 reversed $21 $-1 #
edge $-1 $132 -5 $74 5 $72 $133 forward 7 unknown #
straight-curve $-1 -30 7.5 35 0 -1 0 I I #
edge $-1 $103 -20 $101 0 $95 $134 forward 7 unknown #
vertex $-1 $131 $135 #
straight-curve $-1 -35 10 35 -1 0 0 I I #
vertex $-1 $130 $136 #
straight-curve $-1 -35 10 55 -1 0 0 I I #
point $-1 -30 10 35 #
point $-1 -30 10 55 #
coedge $-1 $137 $71 $138 $139 reversed $73 $-1 #
coedge $-1 $72 $137 $140 $141 reversed $73 $-1 #
point $-1 -30 20 35 #
coedge $-1 $76 $96 $78 $113 forward $34 $-1 #
straight-curve $-1 -30 0 55 0 1 0 I I #
coedge $-1 $115 $78 $142 $143 forward $80 $-1 #
edge $-1 $144 -5 $81 5 $78 $145 forward 7 unknown #
coedge $-1 $79 $115 $146 $147 forward $80 $-1 #
coedge $-1 $114 $112 $148 $149 reversed $80 $-1 #
point $-1 -30 -20 55 #
coedge $-1 $146 $138 $83 $118 reversed $125 $-1 #
edge $-1 $119 -1.5707963267948965 58 $121 1.5707963267948965 58 $83 $150 forward 7
 unknown #
vertex $-1 $118 $151 #
straight-curve $-1 -15 -20 5 1 0 0 I I #
vertex $-1 $85 $152 #
straight-curve $-1 -15 20 5 -1 0 0 I I #
coedge $-1 $148 $153 $154 $155 forward $88 $-1 #
plane-surface $-1 0 0 -5 0 0 -1 -1 0 0 forward_v I I I I #
loop $-1 $-1 $138 $91 #
cone-surface $-1 0 0 0 0 0 1 20 0 0 1 I I 0 1 20 forward I I I I #
plane-surface $-1 -15 20 0 0 -1 0 0 0 -1 forward_v I I I I #
coedge $-1 $140 $95 $97 $131 forward $59 $-1 #
coedge $-1 $95 $142 $96 $130 reversed $59 $-1 #
edge $-1 $103 -10 $144 20 $129 $156 forward 7 unknown #
edge $-1 $101 2.5 $132 12.5 $128 $157 forward 7 unknown #
vertex $-1 $141 $158 #
straight-curve $-1 -35 20 35 1 0 0 I I #
```

```
straight-curve $-1 -40 10 35 0 0 -1 I I #
point $-1 -40 10 35 #
point $-1 -40 10 55 #
coedge $-1 $108 $107 $153 $159 reversed $73 $-1 #
coedge $-1 $117 $154 $107 $139 forward $125 $-1 #
edge $-1 $160 -5 $121 5 $138 $161 forward 7 unknown #
coedge $-1 $162 $128 $108 $141 forward $59 $-1 #
edge $-1 $132 -10 $163 30 $108 $164 forward 7 unknown #
coedge $-1 $129 $162 $112 $143 reversed $59 $-1 #
edge $-1 $144 -30 $165 30 $142 $166 forward 7 unknown #
vertex $-1 $130 $167 #
straight-curve $-1 -35 -20 55 1 0 0 I I #
coedge $-1 $154 $117 $114 $147 reversed $125 $-1 #
edge $-1 $168 -5 $119 5 $146 $169 forward 7 unknown #
coedge $-1 $170 $123 $115 $149 forward $88 $-1 #
edge $-1 $168 -35 $165 5 $115 $171 forward 7 unknown #
ellipse-curve $-1 0 0 5 0 0 1 20 0 0 1 I I #
point $-1 0 -20 5 #
point $-1 0 20 5 #
coedge $-1 $123 $170 $137 $159 forward $88 $-1 #
coedge $-1 $138 $146 $123 $155 reversed $125 $-1 #
edge $-1 $160 -1.570796326794896558 $168 1.570796326794896558 $123 $172 forward
7 unknown #
straight-curve $-1 -40 0 55 0 -1 0 I I #
straight-curve $-1 -40 7.5 35 0 1 0 I I #
point $-1 -40 20 35 #
edge $-1 $163 -25 $160 15 $137 $173 forward 7 unknown #
vertex $-1 $155 $174 #
straight-curve $-1 0 20 0 0 0 1 I I #
coedge $-1 $142 $140 $170 $175 reversed $59 $-1 #
vertex $-1 $159 $176 #
straight-curve $-1 -40 20 25 0 0 -1 I I #
vertex $-1 $175 $177 #
straight-curve $-1 -40 -20 25 0 0 -1 I I #
point $-1 -40 -20 55 #
vertex $-1 $149 $178 #
straight-curve $-1 0 -20 0 0 0 1 I I #
coedge $-1 $153 $148 $162 $175 forward $88 $-1 #
straight-curve $-1 -35 -20 -5 -1 0 0 I I #
ellipse-curve $-1 0 0 -5 0 0 -1 20 0 0 1 I I #
straight-curve $-1 -15 20 -5 1 0 0 I I #
point $-1 0 20 -5 #
edge $-1 $165 -20 $163 20 $162 $179 forward 7 unknown #
point $-1 -40 20 -5 #
point $-1 -40 -20 -5 #
point $-1 0 -20 -5 #
straight-curve $-1 -40 0 -5 0 1 0 I I #
End-of-ACIS-data
```

```
(GENERICOBJECT positive_cylinder SURFACECLASS PROPERTIES (
                (UNARYPROP 0 < MAXSURFCURV PEAK 0.025 WEIGHT 1)
                (UNARYPROP -0.0248 < MINSURFCURV < 0.025 PEAK 0 WEIGHT 1)))


(GENERICOBJECT plane SURFACECLASS PROPERTIES (
                (UNARYPROP -0.0248 < MAXSURFCURV < 0.025 PEAK 0 WEIGHT 1)
                (UNARYPROP -0.0248 < MINSURFCURV < 0.025 PEAK 0 WEIGHT 1)))


(GENERICOBJECT negative_cylinder SURFACECLASS PROPERTIES (
                (UNARYPROP MAXSURFCURV < 0 PEAK -0.0248 WEIGHT 1)
                (UNARYPROP -0.0248 < MINSURFCURV < 0.025 PEAK 0 WEIGHT 1)))

////////////////////////////////////////////////////////////////////////////////
//
// Preprocessor macros
//
#define name2(x,y) x ## y
#define name3(x,y,z) x ## y ## z
#define name4(x,y,z,w) x ## y ## z ## w
#define makedec(x,y) name3(x,.,y)

#define DeclareCircle(rad) (CIRC_ARC name2(circle_,rad) RADIUS rad/2 ANGLE 2*PI)
#define DeclareCircleDecimal(rad,dec) (CIRC_ARC name4(circle_,rad,_,dec) RADIUS
makedec(rad,dec)/2 ANGLE 2*PI)

#define DeclareLine(len) (LINE name2(line_,len) LENGTH len)
#define DeclareLineDecimal(len,dec) (LINE name4(line_,len,_,dec) LENGTH makedec(
len,dec))

#define DeclareQuadrant(rad)\
(ELLIPSE name2(quadrant_,rad) XRADIUS rad YRADIUS rad ENDPOINTS (rad,0,0) (0,rad
,0))

#define DeclareQuadrantDecimal(rad,dec)\
(ELLIPSE name4(quadrant_,rad,_,dec) XRADIUS makedec(rad,dec) YRADIUS makedec(rad
,dec) ENDPOINTS (makedec(rad,dec),0,0) (0,makedec(rad,dec),0))

////////////////////////////////////////////////////////////////////////////////
//
// Base
//
DeclareCircle(90)
DeclareCircle(30)
DeclareCircleDecimal(4,2)
//**NEW
DeclareCircle(15)

(BOUNDARY base_underside_sheet1
          circle_30 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_underside_sheet2
          circle_90 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_underside_sheet3
          circle_4_2 AT TRANSLATION (40,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_underside_sheet4
          circle_4_2 AT TRANSLATION (-40,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_underside_sheet5
          circle_4_2 AT TRANSLATION (0,40,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_underside_sheet6
          circle_4_2 AT TRANSLATION (0,-40,0) ROTATION VECTOR Z INTO Z)

(PLANE base_underside
       BOUNDARY_LIST (base_underside_sheet1 AT ORIGIN
                      base_underside_sheet2 AT ORIGIN
                      base_underside_sheet3 AT ORIGIN
                      base_underside_sheet4 AT ORIGIN
                      base_underside_sheet5 AT ORIGIN
                      base_underside_sheet6 AT ORIGIN)
       INCLUDED_POINT (30,30,0)
       PROPERTIES ( (UNARYPROP 3000 < SIZE < 5000 PEAK 3500 WEIGHT 1)
                    (SUPERTYPE plane))
       )

//////////////////////////////////////////////////////////////////////////////
//
// base_rim
//
(BOUNDARY base_rim_sheet1
          circle_90 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO X)

(BOUNDARY base_rim_sheet2
          circle_90 AT TRANSLATION (5,0,0) ROTATION VECTOR Z INTO X)

(CYLINDER base_rim YRADIUS 45 ZRADIUS 45
       BOUNDARY_LIST (base_rim_sheet1 AT ORIGIN base_rim_sheet2 AT ORIGIN)
       INCLUDED_POINT (2.5,0,90)
       PROPERTIES ( (UNARYPROP 200 < SIZE < 675 PEAK 270 WEIGHT 1)
                    (SUPERTYPE negative_cylinder))
       )

//////////////////////////////////////////////////////////////////////////////
//
// base_screwhole
//
(BOUNDARY base_screwhole_sheet1
          circle_4_2 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO X)

(BOUNDARY base_screwhole_sheet2
          circle_4_2 AT TRANSLATION (5,0,0) ROTATION VECTOR Z INTO X)

(CYLINDER base_screwhole YRADIUS 2.1 ZRADIUS 2.1
       BOUNDARY_LIST (base_screwhole_sheet1 AT ORIGIN base_screwhole_sheet2 AT O
RIGIN)
       INCLUDED_POINT (2.5,0,4.2)
       PROPERTIES ( (UNARYPROP 0 < SIZE < 65 PEAK 5 WEIGHT 1)
                    (SUPERTYPE negative_cylinder))
       )

//////////////////////////////////////////////////////////////////////////////
//
// base_upper
//
DeclareLine(50)
DeclareLine(60)

(BOUNDARY base_upper_sheet1
          line_50 AT TRANSLATION (-30,-25,0) ROTATION VECTOR Z INTO Y
          line_50 AT TRANSLATION ( 30,-25,0) ROTATION VECTOR Z INTO Y
          line_60 AT TRANSLATION (-30, 25,0) ROTATION VECTOR Z INTO X
          line_60 AT TRANSLATION (-30,-25,0) ROTATION VECTOR Z INTO X
)
(BOUNDARY base_upper_sheet2
          circle_90 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_upper_sheet3
          circle_4_2 AT TRANSLATION (40,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_upper_sheet4
          circle_4_2 AT TRANSLATION (-40,0,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_upper_sheet5
          circle_4_2 AT TRANSLATION (0,40,0) ROTATION VECTOR Z INTO Z)
(BOUNDARY base_upper_sheet6
          circle_4_2 AT TRANSLATION (0,-40,0) ROTATION VECTOR Z INTO Z)


(PLANE base_upper
       BOUNDARY_LIST (base_upper_sheet1 AT ORIGIN
                      base_upper_sheet2 AT ORIGIN
                      base_upper_sheet3 AT ORIGIN
                      base_upper_sheet4 AT ORIGIN
                    base_upper_sheet5 AT ORIGIN
                    base_upper_sheet6 AT ORIGIN)
        INCLUDED_POINT (0,35,0)
        PROPERTIES ( (UNARYPROP 1000 < SIZE < 3300 PEAK 1500 WEIGHT 1)
                     (SUPERTYPE plane))
        )

///////////////////////////////////////////////////////////////////////////
//
// holder bit
//
//          5
//     120__---
//    __--    | 50
//   ----------
//     125

DeclareLine(125)
DeclareLine(5)

#define hypotlen (SQRT(50*50+120*120))
(LINE line_hypot_50_120 LENGTH hypotlen)

(BOUNDARY holder_triangley_side_left_pointing_sheet
          line_125 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO X
          line_50 AT TRANSLATION (125,0,0) ROTATION VECTOR Z INTO Y
          line_5 AT TRANSLATION (120,50,0) ROTATION VECTOR Z INTO X
          line_hypot_50_120 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO (120,5
0,0))

(PLANE holder_triangley_side_left_pointing
       BOUNDARY_LIST (holder_triangley_side_left_pointing_sheet AT ORIGIN)
       INCLUDED_POINT (60,10,0)
       PROPERTIES ( (UNARYPROP 700 < SIZE < 3000 PEAK 1000 WEIGHT 1)
                    (SUPERTYPE plane))
       )

(BOUNDARY holder_triangley_side_right_pointing_sheet
          line_125 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO -X
          line_hypot_50_120 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO (-120,
50,0)
          line_5 AT TRANSLATION (-125,50,0) ROTATION VECTOR Z INTO X
          line_50 AT TRANSLATION (-125,0,0) ROTATION VECTOR Z INTO Y)

(PLANE holder_triangley_side_right_pointing
       BOUNDARY_LIST (holder_triangley_side_right_pointing_sheet AT ORIGIN)
       INCLUDED_POINT (-60,10,0)
       PROPERTIES ( (UNARYPROP 700 < SIZE < 3000 PEAK 1000 WEIGHT 1)
                    (SUPERTYPE plane))
       )

///////////////////////////////////////////////////////////////////////////
//
// 5x60 rectangle
//

(BOUNDARY holder_rect5x60_sheet
       line_5 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Y
       line_60 AT TRANSLATION (0,5,0) ROTATION VECTOR Z INTO X
       line_5 AT TRANSLATION (60,5,0) ROTATION VECTOR Z INTO -Y
       line_60 AT TRANSLATION (60,0,0) ROTATION VECTOR Z INTO -X)

(PLANE holder_rect5x60
       BOUNDARY_LIST (holder_rect5x60_sheet AT ORIGIN)
       INCLUDED_POINT (30,2.5,0)
       PROPERTIES ( (UNARYPROP 100 < SIZE < 300 PEAK 125 WEIGHT 1)
                    (SUPERTYPE plane))
       )

///////////////////////////////////////////////////////////////////////////
```

```
//
// 125x60 rectangle
//

(BOUNDARY holder_rect125x60_sheet
       line_125 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Y
       line_60 AT TRANSLATION (0,125,0) ROTATION VECTOR Z INTO X
       line_125 AT TRANSLATION (60,125,0) ROTATION VECTOR Z INTO -Y
       line_60 AT TRANSLATION (60,0,0) ROTATION VECTOR Z INTO -X)

(PLANE holder_rect125x60
       BOUNDARY_LIST (holder_rect125x60_sheet AT ORIGIN)
       INCLUDED_POINT (30,62,0)
       PROPERTIES ( (UNARYPROP 4000 < SIZE < 7500 PEAK 6000 WEIGHT 1)
                    (SUPERTYPE plane))
       )

//////////////////////////////////////////////////////////////////////////
//
// Flanged Plane
//

(BOUNDARY outer_flanged_plane_sheet1
          line_60 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO Y
          line_hypot_50_120 AT TRANSLATION (0,60,0) ROTATION VECTOR Z INTO X
          line_60 AT TRANSLATION (hypotlen,60,0) ROTATION VECTOR Z INTO -Y
          line_hypot_50_120 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO X)

DeclareLine(39)
DeclareLine(104)

DeclareQuadrant(3)

(BOUNDARY outer_flanged_plane_sheet2
          line_39 AT TRANSLATION (15,7.5+3,0) ROTATION VECTOR Z INTO Y
          line_39 AT TRANSLATION (110+15,7.5+3,0) ROTATION VECTOR Z INTO Y
          line_104 AT TRANSLATION (15+3,7.5,0) ROTATION VECTOR Z INTO X
          line_104 AT TRANSLATION (15+3,45+7.5,0) ROTATION VECTOR Z INTO X
          quadrant_3 AT TRANSLATION (15+3,7.5+3,0) ROTATION RST (PI,0,0)
          quadrant_3 AT TRANSLATION (15+110-3,7.5+3,0) ROTATION RST (-PI/2,0,0)
          quadrant_3 AT TRANSLATION (15+3,7.5+45-3,0) ROTATION RST (PI/2,0,0)
          quadrant_3 AT TRANSLATION (15+110-3,7.5+45-3,0) ROTATION RST (0,0,0))

DeclareCircle(2)

(BOUNDARY outer_flanged_plane_sheet3
          circle_2 AT TRANSLATION (19,4,0) ROTATION VECTOR Z INTO Z
          circle_2 AT TRANSLATION (19+51,4,0) ROTATION VECTOR Z INTO Z
          circle_2 AT TRANSLATION (19+51+50,4,0) ROTATION VECTOR Z INTO Z
          circle_2 AT TRANSLATION (19,56,0) ROTATION VECTOR Z INTO Z
          circle_2 AT TRANSLATION (19+51,56,0) ROTATION VECTOR Z INTO Z
          circle_2 AT TRANSLATION (19+51+50,56,0) ROTATION VECTOR Z INTO Z)

(PLANE outer_flanged_plane
       BOUNDARY_LIST (outer_flanged_plane_sheet1 AT ORIGIN
                      outer_flanged_plane_sheet2 AT ORIGIN
                      outer_flanged_plane_sheet3 AT ORIGIN)
       INCLUDED_POINT (5,5,0)
       DEFAULT_POSITION AT TRANSLATION (-50,0,300) ROTATION RST(0,0,0)
       PROPERTIES ( (UNARYPROP 500 < SIZE < 2850 PEAK 1600 WEIGHT 1)
                    (SUPERTYPE plane)
                    (VALUEPROP CENTROID (65,30,0))    )
       )

//////////////////////////////////////////////////////////////////////////
//
// Inner (deeper) flanged bit.
//

DeclareQuadrant(2)
```

```
DeclareLine(33) // 37 - 2 - 2
DeclareLine(98)

(BOUNDARY inner_flanged_plane_sheet1
          line_33 AT TRANSLATION (19,11.5+2,0) ROTATION VECTOR Z INTO Y
          line_33 AT TRANSLATION (102+19,11.5+2,0) ROTATION VECTOR Z INTO Y
          line_98 AT TRANSLATION (19+2,11.5,0) ROTATION VECTOR Z INTO X
          line_98 AT TRANSLATION (19+2,37+11.5,0) ROTATION VECTOR Z INTO X
          quadrant_2 AT TRANSLATION (19+2,11.5+2,0) ROTATION RST (PI,0,0)
          quadrant_2 AT TRANSLATION (19+102-2,11.5+2,0) ROTATION RST (-PI/2,0,0)
          quadrant_2 AT TRANSLATION (19+2,11.5+37-2,0) ROTATION RST (PI/2,0,0)
          quadrant_2 AT TRANSLATION (19+102-2,11.5+37-2,0) ROTATION RST (0,0,0))

(PLANE inner_flanged_plane
       BOUNDARY_LIST (outer_flanged_plane_sheet2 AT ORIGIN
                      inner_flanged_plane_sheet1 AT ORIGIN)
       INCLUDED_POINT (17,10,0)
//       PROPERTIES ( (UNARYPROP 100 < SIZE < 1250 PEAK 200 WEIGHT 1)
//                    (SUPERTYPE plane))
// unlikely to be distinguished from innermost
       )

//////////////////////////////////////////////////////////////////////////////
//
// Innermost flanged bit.
//

DeclareQuadrantDecimal(0,5)
DeclareLine(32) // 35 - .5 - .5
DeclareLine(97)

(BOUNDARY innermost_flanged_plane_sheet1
          line_32 AT TRANSLATION (21,13.5+0.5,0) ROTATION VECTOR Z INTO Y
          line_32 AT TRANSLATION (98+21,13.5+0.5,0) ROTATION VECTOR Z INTO Y
          line_97 AT TRANSLATION (21+0.5,13.5,0) ROTATION VECTOR Z INTO X
          line_97 AT TRANSLATION (21+0.5,33+13.5,0) ROTATION VECTOR Z INTO X
          quadrant_0_5 AT TRANSLATION (21+0.5,13.5+0.5,0) ROTATION RST (PI,0,0)
          quadrant_0_5 AT TRANSLATION (21+98-0.5,13.5+0.5,0) ROTATION RST (-PI/0
.5,0,0)
          quadrant_0_5 AT TRANSLATION (21+0.5,13.5+33-0.5,0) ROTATION RST (PI/0.
5,0,0)
          quadrant_0_5 AT TRANSLATION (21+98-0.5,13.5+33-0.5,0) ROTATION RST (0,
0,0))

#define cterx (15*13/5)
#define ctery 15

(ELLIPSE cylinder_top_ellipse_for_plane
         XRADIUS cterx
         YRADIUS ctery
         ENDPOINTS (cterx,0,0) (cterx,0,0))

(BOUNDARY innermost_flanged_plane_sheet2
          cylinder_top_ellipse_for_plane AT TRANSLATION (68,30,0) ROTATION RST (
0,0,0))

(PLANE innermost_flanged_plane
       BOUNDARY_LIST (innermost_flanged_plane_sheet2 AT ORIGIN
                      innermost_flanged_plane_sheet1 AT ORIGIN)
       INCLUDED_POINT (22,14.5,0)
       PROPERTIES ( (UNARYPROP 1000 < SIZE < 2000 PEAK 4950 WEIGHT 1)
                    (SUPERTYPE plane)
                    (VALUEPROP CENTROID (48,16,0)))
       )


#define cteangle 0.05
#define cex (cterx * COS(cteangle))
#define cey (ctery * SIN(cteangle))
```

```
(ELLIPSE cylinder_top_ellipse_for_mask
         XRADIUS cterx
         YRADIUS ctery
         ENDPOINTS (-cex,-cey,0) (-cex,cey,0))

(BOUNDARY inside_cylinder_circle
          circle_15 AT TRANSLATION (0,0,0) ROTATION VECTOR Z INTO X)

#define cosa (5/13)
#define sina (12/13)

(BOUNDARY inside_cylinder_ellipse
          cylinder_top_ellipse_for_mask AT TRANSLATION (-65,0,0)
          ROTATION VECTOR_PAIR Y INTO Y, Z INTO (-5,0,12))

////////////////////////////////////////////////////////////////
// THIS section only to get inside_cylinder to draw properly
// because of some sort of wrap-around bug that does not properly close
// off a surface
////////////////////////////////////////////////////////////////
//DeclareLine(98)
(BOUNDARY bndline1
          line_98 AT TRANSLATION (-98,0,-15) ROTATION VECTOR Z INTO X)
(BOUNDARY bndline2
          line_98 AT TRANSLATION (-98,0.1,-14.9) ROTATION VECTOR Z INTO X)
(BOUNDARY bndline3
                circle_30 AT TRANSLATION (0,5,-20) ROTATION VECTOR Z INTO X)
////////////////////////////////////////////////////////////////

(CYLINDER inside_cylinder
          YRADIUS -15
          ZRADIUS -15
          BOUNDARY_LIST (inside_cylinder_circle AT ORIGIN
                         inside_cylinder_ellipse AT ORIGIN
//                       bndline1 AT ORIGIN      // needed only for drawing bug
//                       bndline2 AT ORIGIN      // needed only for drawing bug
//                       bndline3 AT ORIGIN      // needed only for drawing bug
                        )
          INCLUDED_POINT (-1,1,15)
          DEFAULT_POSITION AT TRANSLATION (50,0,300) ROTATION RST (0,4.7,1.57)
          PROPERTIES ( (UNARYPROP 500 < SIZE < 1000 PEAK 700 WEIGHT 1)
                       (SUPERTYPE negative_cylinder))
          )


///////////////////////////////////////////////////////////////////////////////
//
//
//
(ASSEMBLY pencil_holder
        PLACED_CURVES
        // Surface: base_underside of Assembly pencil_holder
        // boundary base_underside_sheet1
        circle_30 AT TRANSLATION (0.00,0.00,0.00)
        ROTATION RST(0,0,0)

        // boundary base_underside_sheet2
        circle_90 AT TRANSLATION (0.00,0.00,0.00)
        ROTATION RST(0,0,0)

        // boundary base_underside_sheet3
        circle_4_2 AT TRANSLATION (40.00,0.00,0.00)
        ROTATION RST(0,0,0)

        // boundary base_underside_sheet4
        circle_4_2 AT TRANSLATION (-40.00,0.00,0.00)
        ROTATION RST(0,0,0)

        // boundary base_underside_sheet5
        circle_4_2 AT TRANSLATION (0.00,40.00,0.00)
```

```
ROTATION RST(0,0,0)

// boundary base_underside_sheet6
circle_4_2 AT TRANSLATION (0.00,-40.00,0.00)
ROTATION RST(0,0,0)

// Surface: base_rim of Assembly pencil_holder
// boundary base_rim_sheet1
circle_90 AT TRANSLATION (0.00,0.00,0.00)
ROTATION RST(0,0,0)

// boundary base_rim_sheet2
circle_90 AT TRANSLATION (0.00,0.00,5.00)
ROTATION RST(0,0,0)

// Surface: base_screwhole of Assembly pencil_holder
// boundary base_screwhole_sheet1
circle_4_2 AT TRANSLATION (40.00,0.00,0.00)
ROTATION RST(0,0,0)

// boundary base_screwhole_sheet2
circle_4_2 AT TRANSLATION (40.00,0.00,5.00)
ROTATION RST(0,0,0)

// Surface: base_screwhole of Assembly pencil_holder
// boundary base_screwhole_sheet1
circle_4_2 AT TRANSLATION (-40.00,0.00,0.00)
ROTATION RST(0,0,0)

// boundary base_screwhole_sheet2
circle_4_2 AT TRANSLATION (-40.00,0.00,5.00)
ROTATION RST(0,0,0)

// Surface: base_screwhole of Assembly pencil_holder
// boundary base_screwhole_sheet1
circle_4_2 AT TRANSLATION (0.00,40.00,0.00)
ROTATION RST(0,0,0)

// boundary base_screwhole_sheet2
circle_4_2 AT TRANSLATION (0.00,40.00,5.00)
ROTATION RST(0,0,0)

// Surface: base_screwhole of Assembly pencil_holder
// boundary base_screwhole_sheet1
circle_4_2 AT TRANSLATION (0.00,-40.00,0.00)
ROTATION RST(0,0,0)

// boundary base_screwhole_sheet2
circle_4_2 AT TRANSLATION (0.00,-40.00,5.00)
ROTATION RST(0,0,0)

// Surface: base_upper of Assembly pencil_holder
// boundary base_upper_sheet1
line_50 AT TRANSLATION (30.00,-25.00,5.00)
ROTATION RST(3.14159,1.5708,1.5708)

line_50 AT TRANSLATION (-30.00,-25.00,5.00)
ROTATION RST(3.14159,1.5708,1.5708)

line_60 AT TRANSLATION (30.00,25.00,5.00)
ROTATION RST(0,1.5708,0)

line_60 AT TRANSLATION (30.00,-25.00,5.00)
ROTATION RST(0,1.5708,0)

// boundary base_upper_sheet2
circle_90 AT TRANSLATION (0.00,0.00,5.00)
ROTATION RST(0,3.14159,4.71239)

// boundary base_upper_sheet3
```

```
        circle_4_2 AT TRANSLATION (-40.00,0.00,5.00)
        ROTATION RST(0,3.14159,4.71239)

        // boundary base_upper_sheet4
        circle_4_2 AT TRANSLATION (40.00,-0.00,5.00)
        ROTATION RST(0,3.14159,4.71239)

        // boundary base_upper_sheet5
        circle_4_2 AT TRANSLATION (0.00,40.00,5.00)
        ROTATION RST(0,3.14159,4.71239)

        // boundary base_upper_sheet6
        circle_4_2 AT TRANSLATION (-0.00,-40.00,5.00)
        ROTATION RST(0,3.14159,4.71239)

        // Surface: holder_triangley_side_left_pointing of Assembly pencil_holde
r
        // boundary holder_triangley_side_left_pointing_sheet
        line_125 AT TRANSLATION (-30.00,-25.00,130.00)
        ROTATION RST(0,3.14159,3.14159)

        line_50 AT TRANSLATION (-30.00,-25.00,5.00)
        ROTATION RST(1.5708,1.5708,3.14159)

        line_5 AT TRANSLATION (-30.00,25.00,10.00)
        ROTATION RST(0,3.14159,3.14159)

        line_hypot_50_120 AT TRANSLATION (-30.00,-25.00,130.00)
        ROTATION RST(2.7468,2.7468,1.96559)

        // Surface: holder_rect125x60 of Assembly pencil_holder
        // boundary holder_rect125x60_sheet
        line_125 AT TRANSLATION (30.00,-25.00,5.00)
        ROTATION RST(3.14159,0,0)

        line_60 AT TRANSLATION (30.00,-25.00,130.00)
        ROTATION RST(4.71239,1.5708,1.5708)

        line_125 AT TRANSLATION (-30.00,-25.00,130.00)
        ROTATION RST(0,3.14159,1.5708)

        line_60 AT TRANSLATION (-30.00,-25.00,5.00)
        ROTATION RST(1.5708,1.5708,1.5708)

        // Surface: holder_triangley_side_right_pointing of Assembly pencil_hold
er
        // boundary holder_triangley_side_right_pointing_sheet
        line_125 AT TRANSLATION (30.00,-25.00,130.00)
        ROTATION RST(0,3.14159,0)

        line_hypot_50_120 AT TRANSLATION (30.00,-25.00,130.00)
        ROTATION RST(3.53638,2.7468,1.17601)

        line_5 AT TRANSLATION (30.00,25.00,5.00)
        ROTATION RST(0,0,0)

        line_50 AT TRANSLATION (30.00,-25.00,5.00)
        ROTATION RST(4.71239,1.5708,0)

        // Surface: holder_rect5x60 of Assembly pencil_holder
        // boundary holder_rect5x60_sheet
        line_5 AT TRANSLATION (-30.00,25.00,5.00)
        ROTATION RST(0,0,0)

        line_60 AT TRANSLATION (-30.00,25.00,10.00)
        ROTATION RST(1.5708,1.5708,1.5708)

        line_5 AT TRANSLATION (30.00,25.00,10.00)
        ROTATION RST(3.14159,3.14159,1.5708)
```

```
line_60 AT TRANSLATION (30.00,25.00,5.00)
ROTATION RST(4.71239,1.5708,1.5708)

// Surface: outer_flanged_plane of Assembly pencil_holder
// boundary outer_flanged_plane_sheet1
line_60 AT TRANSLATION (-30.00,-25.00,130.00)
ROTATION RST(0.394791,1.5708,2.7468)

line_hypot_50_120 AT TRANSLATION (30.00,-25.00,130.00)
ROTATION RST(4.71239,2.7468,0)

line_60 AT TRANSLATION (30.00,25.00,10.00)
ROTATION RST(2.7468,1.5708,3.53638)

line_hypot_50_120 AT TRANSLATION (-30.00,-25.00,130.00)
ROTATION RST(4.71239,2.7468,0)

// boundary outer_flanged_plane_sheet2
line_39 AT TRANSLATION (-19.50,-19.23,116.15)
ROTATION RST(0.394791,1.5708,2.7468)

line_39 AT TRANSLATION (-19.50,23.08,14.62)
ROTATION RST(0.394791,1.5708,2.7468)

line_104 AT TRANSLATION (-22.50,-18.08,113.38)
ROTATION RST(4.71239,2.7468,0)

line_104 AT TRANSLATION (22.50,-18.08,113.38)
ROTATION RST(4.71239,2.7468,0)

quadrant_3 AT TRANSLATION (-19.50,-18.08,113.38)
ROTATION RST(1.57079,1.96559,2.55623e-06)

quadrant_3 AT TRANSLATION (-19.50,21.92,17.38)
ROTATION RST(3.14159,1.96559,4.71239)

quadrant_3 AT TRANSLATION (19.50,-18.08,113.38)
ROTATION RST(6.28318,1.96559,1.5708)

quadrant_3 AT TRANSLATION (19.50,21.92,17.38)
ROTATION RST(4.71239,1.96559,3.14159)

// boundary outer_flanged_plane_sheet3
circle_2 AT TRANSLATION (-26.00,-17.69,112.46)
ROTATION RST(4.71239,1.96559,3.14159)

circle_2 AT TRANSLATION (-26.00,1.92,65.38)
ROTATION RST(4.71239,1.96559,3.14159)

circle_2 AT TRANSLATION (-26.00,21.15,19.23)
ROTATION RST(4.71239,1.96559,3.14159)

circle_2 AT TRANSLATION (26.00,-17.69,112.46)
ROTATION RST(4.71239,1.96559,3.14159)

circle_2 AT TRANSLATION (26.00,1.92,65.38)
ROTATION RST(4.71239,1.96559,3.14159)

circle_2 AT TRANSLATION (26.00,21.15,19.23)
ROTATION RST(4.71239,1.96559,3.14159)

// Surface: inner_flanged_plane of Assembly pencil_holder
// boundary outer_flanged_plane_sheet2
line_39 AT TRANSLATION (-19.50,-20.00,114.31)
ROTATION RST(0.394791,1.5708,2.7468)

line_39 AT TRANSLATION (-19.50,22.31,12.77)
ROTATION RST(0.394791,1.5708,2.7468)

line_104 AT TRANSLATION (-22.50,-18.85,111.54)
```

```
ROTATION RST(4.71239,2.7468,0)

line_104 AT TRANSLATION (22.50,-18.85,111.54)
ROTATION RST(4.71239,2.7468,0)

quadrant_3 AT TRANSLATION (-19.50,-18.85,111.54)
ROTATION RST(1.57079,1.96559,2.55623e-06)

quadrant_3 AT TRANSLATION (-19.50,21.15,15.54)
ROTATION RST(3.14159,1.96559,4.71239)

quadrant_3 AT TRANSLATION (19.50,-18.85,111.54)
ROTATION RST(6.28318,1.96559,1.5708)

quadrant_3 AT TRANSLATION (19.50,21.15,15.54)
ROTATION RST(4.71239,1.96559,3.14159)

// boundary inner_flanged_plane_sheet1
line_33 AT TRANSLATION (-16.50,-18.46,110.62)
ROTATION RST(0.394791,1.5708,2.7468)

line_33 AT TRANSLATION (-16.50,20.77,16.46)
ROTATION RST(0.394791,1.5708,2.7468)

line_98 AT TRANSLATION (-18.50,-17.69,108.77)
ROTATION RST(4.71239,2.7468,0)

line_98 AT TRANSLATION (18.50,-17.69,108.77)
ROTATION RST(4.71239,2.7468,0)

quadrant_2 AT TRANSLATION (-16.50,-17.69,108.77)
ROTATION RST(1.57079,1.96559,2.55623e-06)

quadrant_2 AT TRANSLATION (-16.50,20.00,18.31)
ROTATION RST(3.14159,1.96559,4.71239)

quadrant_2 AT TRANSLATION (16.50,-17.69,108.77)
ROTATION RST(6.28318,1.96559,1.5708)

quadrant_2 AT TRANSLATION (16.50,20.00,18.31)
ROTATION RST(4.71239,1.96559,3.14159)

// Surface: innermost_flanged_plane of Assembly pencil_holder
// boundary innermost_flanged_plane_sheet2
cylinder_top_ellipse_for_plane AT TRANSLATION (-0.00,0.38,65.38)
ROTATION RST(4.71239,1.96559,3.14159)

// boundary innermost_flanged_plane_sheet1
line_32 AT TRANSLATION (-16.00,-17.69,108.77)
ROTATION RST(0.394791,1.5708,2.7468)

line_32 AT TRANSLATION (-16.00,20.00,18.31)
ROTATION RST(0.394791,1.5708,2.7468)

line_97 AT TRANSLATION (-16.50,-17.50,108.31)
ROTATION RST(4.71239,2.7468,0)

line_97 AT TRANSLATION (16.50,-17.50,108.31)
ROTATION RST(4.71239,2.7468,0)

quadrant_0_5 AT TRANSLATION (-16.00,-17.50,108.31)
ROTATION RST(1.57079,1.96559,2.55623e-06)

quadrant_0_5 AT TRANSLATION (-16.00,19.81,18.77)
ROTATION RST(4.71239,1.96559,3.14159)

quadrant_0_5 AT TRANSLATION (16.00,-17.50,108.31)
ROTATION RST(4.71238,1.96559,3.1416)

quadrant_0_5 AT TRANSLATION (16.00,19.81,18.77)
```

```
        ROTATION RST(4.71239,1.96559,3.14159)

        // Surface: inside_cylinder of Assembly pencil_holder
        // boundary inside_cylinder_circle
        circle_30 AT TRANSLATION (0.00,0.00,0.00)
        ROTATION RST(4.71239,3.14159,4.71239)

        // boundary inside_cylinder_ellipse
        cylinder_top_ellipse_for_mask AT TRANSLATION (-0.00,-0.00,65.00)
        ROTATION RST(1.5708,1.17601,3.14159)



        PLACED_SURFACES
#if 1
        base_underside AT TRANSLATION (0,0,0) ROTATION VECTOR_PAIR -Z INTO -Z, X
 INTO X
        base_rim AT TRANSLATION (0,0,0) ROTATION VECTOR X INTO Z
        base_screwhole AT TRANSLATION (40,0,0) ROTATION VECTOR X INTO Z
        base_screwhole AT TRANSLATION (-40,0,0) ROTATION VECTOR X INTO Z
        base_screwhole AT TRANSLATION (0,40,0) ROTATION VECTOR X INTO Z
        base_screwhole AT TRANSLATION (0,-40,0) ROTATION VECTOR X INTO Z

        base_upper AT TRANSLATION (0,0,5) ROTATION VECTOR_PAIR -Z INTO Z, X INTO
 X
        holder_triangley_side_left_pointing AT TRANSLATION (-30,-25,130)
                   ROTATION VECTOR_PAIR Y INTO Y, X INTO -Z
        holder_rect125x60 AT TRANSLATION (30,-25,5)
                   ROTATION VECTOR_PAIR Y INTO Z, X INTO -X
        holder_triangley_side_right_pointing AT TRANSLATION (30,-25,130)
                   ROTATION VECTOR_PAIR Y INTO Y, X INTO Z
        holder_rect5x60 AT TRANSLATION (-30,25,5)
                   ROTATION VECTOR_PAIR Y INTO Z, -Z INTO Y

        outer_flanged_plane AT TRANSLATION (-30,-25,130)
                   ROTATION VECTOR_PAIR Y INTO X, X INTO (0,50,-120)

        inner_flanged_plane AT TRANSLATION (-30,-25-2*cosa,130-2*sina)
                   ROTATION VECTOR_PAIR Y INTO X, X INTO (0,50,-120)

#endif
        innermost_flanged_plane AT TRANSLATION (-30,-25-2*cosa,130-2*sina)
                   ROTATION VECTOR_PAIR Y INTO X, X INTO (0,50,-120)

        inside_cylinder AT TRANSLATION (0,0,0) ROTATION VECTOR_PAIR X INTO -Z, Z
 INTO Y

        VDFG_LIST ( pencil_holder_view_2 pencil_holder_view_3 pencil_holder_view
_5 pencil_holder_view_6 pencil_holder_view_7 pencil_holder_view_8 pencil_holder_
view_9 pencil_holder_view_10 pencil_holder_view_11 pencil_holder_view_12 pencil_
holder_view_13 pencil_holder_view_14 pencil_holder_view_15 pencil_holder_view_16
 )

        DEFAULT_POSITION AT TRANSLATION (50,-50,300)
        ROTATION RST (0,1.6,0)
        )


(VDFG   pencil_holder_view_2
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_right_pointing#1 base_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.939347,-0.305212,-0.156434) < -0.7999)))
```

```
(VDFG   pencil_holder_view_3
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_right_pointing#1 base_upper#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.880037,0.139384,0.45399) < -0.7999)))


(VDFG   pencil_holder_view_5
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( innermost_flanged_plane#1 outer_flanged_plane#1 base_upper#1
 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.140291,0.431771,0.891007) < -0.7999)))


(VDFG   pencil_holder_view_6
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( innermost_flanged_plane#1 outer_flanged_plane#1  holder_tria
ngley_side_right_pointing#1 base_rim#1 inside_cylinder#1)
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.580549,0.799057,-0.156434) < -0.7999)))


(VDFG   pencil_holder_view_7
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( inside_cylinder#1 innermost_flanged_plane#1 outer_flanged_pl
ane#1  base_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.154509,0.975528,0.156434) < -0.7999)))


(VDFG   pencil_holder_view_8
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( innermost_flanged_plane#1 outer_flanged_plane#1  holder_tria
ngley_side_left_pointing#1 base_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.580549,0.799057,-0.156434) < -0.7999)))


(VDFG   pencil_holder_view_9
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_left_pointing#1 base_underside#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.475528,0.345491,-0.809017) < -0.7999)))


(VDFG   pencil_holder_view_10
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_left_pointing#1 base_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.951057,2.32934e-16,-0.309017) < -0.7999)))


(VDFG   pencil_holder_view_11
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_left_pointing#1 base_upper#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.880037,0.139384,0.45399) < -0.7999)))


(VDFG   pencil_holder_view_12
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_rect125x60#1 holder_triangley_side_left_pointing#1 ba
se_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.672499,-0.672499,-0.309017) < -0.7999)))


(VDFG   pencil_holder_view_13
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_rect125x60#1 base_underside#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.345491,-0.475528,-0.809017) < -0.7999)))


(VDFG   pencil_holder_view_14
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_rect125x60#1 base_upper#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-0.25,-0.769421,0.587785) < -0.7999)))


(VDFG   pencil_holder_view_15
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_rect125x60#1 base_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (-6.04765e-17,-0.987688,-0.156434) < -0.7999))
)


(VDFG   pencil_holder_view_16
        ASSEMBLY pencil_holder
        // cached placed features
        VIS_GROUP ( holder_triangley_side_right_pointing#1 holder_rect125x60#1 b
ase_rim#1 )
        TAN_GROUP (NONE )
        PART_OBSCURED_GROUP (NONE )
        CONNECT_CONSTRAINTS (NONE)
        NEW_FEAT_CONSTRAINTS (NONE)
        POSITION_CONSTRAINTS (
                (VIEWER DOTPR MAP (0.672499,-0.672499,-0.309017) < -0.7999)))
```