N92-24401

SDL: A SURFACE DESCRIPTION LANGUAGE

Raymond C. Maple, Capt USAF
Wright Laboratory, Weapon Flight Mechanics Division
Eglin AFB, FL

SUMMARY

A new interpreted language specifically designed for surface grid generation is introduced. Many unique aspects of the language are discussed, including the farray, vector, curve and surface data types and the operators used to manipulate them. Custom subroutine libraries written in the language are used to easily build surface grids for generic missile shapes.

INTRODUCTION

Most new software packages for surface modeling and grid generation attempt to ease the process by providing a graphical "point and click" environment for surface construction. While this type of environment provides many advantages over previous "runstream" type methods, it does have some shortcomings. Flexibility and power are often sacrificed for ease of use. Total reliance on the user interface often means repeating tedious sequences over and over again. Finally, complex user interfaces generally make modification and extension of the surface generation code very difficult. The Surface Description Language (SDL) was written for those situations where the need for power, flexibilty, and extensibility outweighs the need for an easy to use interface.

THE SDL LANGUAGE

SDL is a portable high level interpreted language optimized for the manipulation of one and two dimensional arrays of three–dimensional vectors. The language has its roots in text based grid generators such as EAGLE (Ref 1), but improves in them by providing a more flexible, powerful command syntax. With SDL, surface grid generation is approached with the same logic and problem solving process used when writing a computer program in FORTRAN or C, but SDL simplifies the task by providing language features specifically designed for surface generation.

Like most conventional computer languages, SDL provides named variables, looping and branching (if–then) constructs, and user defined subroutines. While these features give SDL much of its power and flexibility, it is the unique variable types, coupled with the easy manipulation of these types, that make SDL particularly well suited to surface grid generation.

Farrays, Vectors, Curves, and Surfaces

One of the difficulties encountered when using a conventional computer language to generate a surface grid is that such languages are designed to deal with only one floating point number at a time, whereas curves and surfaces are made up of many triplets of numbers. Curves and surfaces must be represented by n x 3 or m x n x 3 arrays of real numbers, which can be difficult for the novice or occasional programmer to manipulate. SDL remedies this problem by providing vectors, curves, and surfaces as basic data types which can be referenced as a single entity. An additional data type, the farray, allows one to treat arrays of floats in a similar manner.

Farrays. Because singly dimensioned arrays of real numbers are often required to specify distributions, etc., the farray (pronounced "f array") data type is provided. This is a one–dimensional array of floating point numbers. Most arithmetic operators are defined for farrays.

Vectors. The vector is a single entity representing a triplet of floating point numbers. It is the basic component from which curves and surfaces are constructed. Many basic operators, including "+", "-", "*" and "/" understand the structure of a vector and perform the correct vector or scalar operation when used in expressions. In addition, special vector operators, such as ".." (vector dot product), ".*" (vector cross product), and "|*vector*|" (vector magnitude) are defined. Individual components of a vector can be accessed by using ".x", ".y", and ".z" suffixes.

Curves. Curves are one–dimensional collections of vectors. As with farrays and vectors, the basic structure of a curve is known to SDL and thus to the basic operators. Curves can appear in expressions involving scalars, farrays, vectors, and other curves (or parts of curves) with the same dimension. The dimension of a curve can be determined at any time by means of the ".d1" suffix. The vector component suffixes can be used with curves, and result in farrays.

Surfaces. Surfaces are two-dimensional collections of vectors that share the same properties as curves. Dimensions are referenced through the ".d1" and ".d2" suffixes. Examples involving vectors, curves, and surfaces in expressions are given later in this paper.

Parametric Curves and Surfaces

In addition to the basic three-space curves and surfaces, data types for parametric curves and surfaces are defined. Objects declared with one of these data types (pcurve or psurface) are independant of any particular parameterization. They are an abstract representation of a curve or surface with parameter(s) varying between zero and one. The actual form of parameterization is remembered when a parametric variable is assigned, and taken into account when the variable is used to distribute points in three-space. Only one operator, the "=" assignment operator, recognizes the parametric data types. Currently implemented parameterizations include linear interpolations and cubic (bicubic) splines.

Other Data Types

Ints and Floats. SDL provides for variables with the traditional integer (int) and floating point (float) data types. Variables with these types are single values, and cannot be used to form arrays.

Strings and Files. The string and file data types are special purpose data types that hold quoted string and file IO "unit" numbers respectively.

Parts of Objects: the Range Operator

With conventional computer languages, components of an array are referenced by providing array indices. These indices are used to reference a single array element. Working with a portion of a "curve" or "surface" involves looping over the proper array indices. SDL extends the concept of the array index and provides the range operator, which allows one to specify a range of indices with one term.

While parts of curves and surfaces referenced with the range operator can be thought of as array elements, it is more useful to think of them as vectors, sub–curves, and sub–surfaces. When a single element of a curve or surface is referenced, i.e. simple indices are used, that element functions as a vector. When a range operator is used with a curve, the result is functionally a curve of smaller or equal dimension. The results with a surface depend on whether the range operator is used in one index (result is a curve) or both (result is a surface).

The range operator is simple to use. Instead of referencing a single element, as in `c[7]`, (SDL uses square brackets for indexing) one specifies the beginning and end of the desired range separated by a colon, as in `c[3:7]`. If no index is given, as in `s[1,]`, where `s` is a surface and no index or range is given for the second index, then the entire range for that dimension is assumed. (Therefore the example specifies the entire first i line in surface `s`, which is functionally a curve.)

The range operator can be used with any dimensioned object, including farrays, curves, and surfaces. Because the resulting sub–objects are functionally identical to their "complete" counterparts, sub–objects can be used anywhere a complete object is appropriate. This makes it easy to restrict operations to portions of an object.

Expressions Using Farrays, Vectors, Curves, and Surfaces

As was previously mentioned, many arithmetic operators are defined for farrays, vectors, curves, and surfaces. In this section, several examples of expressions using these objects are provided. In the following examples, `v` is a variable of type vector, `c` is a variable of type curve, and `s` is a variable of type surface.

Scaling an Object

One of the simplest geometric operations that is regularly performed is to scale an existing curve or surface. In SDL, this is achieved by simple scalar multiplication.

```
s *= 2.0;
```

The preceding example multiplies each component of each element in s by 2.0. (NOTE: `a *= b`, like in the C language, is equivalent to writing `a = a * b`. Also, as in C, statements in SDL are terminated with a semicolon.) Scaling a single component of an object is performed using a component suffix. For example:

```
s.x *= 2.0;
```

scales only the x components of s.

Translating an Object

Translating an object is just as simple as scaling one. To perform a translation, one adds a constant vector to each element of the object. For example:

```
c += {0,10,0};
```

will translate the curve c 10 units in the y direction. Another way to achieve the same results would be:

```
c.y += 10;
```

Point Distribution on a Line

A slightly more complex example is the distribution of points along a straight line between two given points. For this example, suppose we have an farray, fa, with the same dimension as curve c. This variable will hold the relative distribution of points, varying from 0 to 1. Furthermore, assume that the curve starting and ending points (vectors) have been placed in the first and last elements of c. Distributing the interior points reduces to:

```
c = c[1] + fa * (c[c.d1] - c[1]);
```

Note that a vector (`c[c.d1] - c[1]`) multiplied by an farray results in a curve, each element of which is formed by multiplying the vector by the respective element of the farray. Since the distribution begins at 0, the resulting curve starts at (0,0,0). Adding the vector c[1] to the result translates the curve to the proper location.

As a final example of the conciseness and power of these expressions, a Bézier curve with four control points is constructed. The Bézier curve is a parametric curve defined by the following vector function. (Ref 2)

$$\mathbf{P}(u) = \sum_{k=0}^{n} p_k B_{k,n}(u) \quad (1)$$

$$B_{k,n}(u) = C(n,k)u^k(1-u)^{n-k} \quad (2)$$

$$C(n,k) = \frac{n!}{k!\,(n-k)!} \quad (3)$$

In the above equations, u is the parameter varying between 0 and 1, and p_k are the $n+1$ control points. The SDL code to implement these equations is:

```
c = p0*(1-u)^3 + p1*3*u*(1-u)^2 + p2*3*u^2*(1-u) + p3*u^3;
```

where u is an farray containing the values of the parameter u, p0, p1, etc. are vectors containing the control points, and $C(3,0) = C(3,3) = 1$ and $C(3,1) = C(3,2) = 3$.

User Subroutines

Clearly one does not want to have write and rewrite SDL code to do such common tasks as generating arcs and lines. The obvious thing to do is to create subroutines which perform those functions. By producing a set of subroutines to perform common functions such as curve generation, interpolation, blending, etc., one reduces most surface generation tasks to series of subroutine calls. Such a subroutine library is included with SDL, and is called stdsubs.sdl.

While the standard subroutine library supplies a large number of basic capabilities, there are always requirements specific to a particular task. SDL provides an easy way to automate portions of a task by allowing the end user to build specialized subroutines that either work in conjunction with or replace the subroutines in the standard library. This makes SDL infinitely expandable. Examples of the use of custom subroutine libraries are given below.

SDL subroutines have the following properties:

- A subroutine can have any number of declared arguments.
- Subroutine arguments can be passed back or not at the users option.
- Subroutine arguments automatically inherit the dimensions of the object passed in. Thus if a portion of a curve or surface is used as a subroutine calling argument, the subroutine recieves a curve or surface with the dimensions of the portion passed in.
- The final declared argument in a subroutine can be declared as a multiple argument, allowing a variable number of objects of the same type to passed into the subroutine. Special constructs and operators allow sequential access to the objects passed in.

While "native" SDL is very powerful, it does have its limitations. Some operations, such as rotations about an axis, must be done on a component by component basis. The coding, while certainly possible, is not simplified, and the overhead of executing the SDL code would result in some unnecessary performance loss. Other functions, such as the splining of objects and distribution of points on a spline, would be far too cumbersome to program in SDL. To perform these types of functions, SDL uses intrinsic, or built–in functions.

Intrinsic Functions in SDL

FORTRAN programmers are familiar with intrinsic functions which perform trigonometric, logarithmic, and other mathematical functions. SDL provides these same functions, but it also provides others that are specifically oriented toward curve and surface generation, such as curve and surface rotation, and parametric distribution functions. These functions, in combination with the subroutines provided in the standard library, complete the basic surface generation capability of SDL.

Adding Intrinsic Functions to SDL

SDL is designed to be easily extended by adding new intrinsic functions. This is done by writing a C function which performs the actions taken by the intrinsic function. This function must comply with a simple, well defined typing and argument specification. Once the function is written, it is added to SDL by adding an entry to a list which contains such information as the name of the intrinsic function and the number and types of its arguments. When SDL is recompiled, the new function will be a part of the language. This ease of adding new functions makes SDL an ideal testbed for new algorithms.

EXAMPLE: MISSILE BODY SURFACE

Two bodies commonly modeled to validate Computational Fluid Dynamics (CFD) codes are the tangent–ogive–cylinder and the tangent–ogive–cylinder–ogive. These generic missile shapes occur with and without fins. If the CFD data is being compared to wind tunnel test results, a sting is usually modeled. The following examples illustrate the use of SDL library routines to easily generate the missile body surface grids.

Naca.sdl

This file contains routines which generate NACA airfoil sections using the defining polynomials. Specifically, the subroutine naca4, which generates a NACA 4 series airfoil, is used. (Ref 3)

Mslbody.sdl

This file contains routines which generate a tangent–ogive–cylinder or tangent–ogive–cylinder–ogive body surface. All parameters controlling the dimensions of the body are configurable. In addition, the routines will optionally generate a sting and make allowances for any number of fins of an arbitrary cross section. If fins are specified, then a section of the surface lying between two fins is generated. Otherwise one half of the missile surface is produced.

Mslbody.sdl is a relatively complex SDL code, and is too long (about 350 lines of SDL code) to include in this paper. It is a good deal shorter than if it were written in a conventional language, however. Most of its complexity comes from the number of parameters that can be varied.

Stdsubs.sdl and Intrinsic Functions

The following examples also use several variables and subroutines from the standard subroutine library, along with some intrinsic functions. A complete reference manual describing all of the functions and subroutines in SDL is beyond the scope of this paper. However, a brief description of those used in the examples follows:

PI, PI_d2 – These are global variables defined in stdsubs.sdl. They contain the values of Pi and Pi/2 respectively.

arc – Subroutine to generate a circular arc in the xy plane. Requires four arguments: `curve` to be filled (returned); start angle; end angle; and radius.

line – Subroutine to generate straight line with linear distribution. Requires one argument: `curve` (returned) to be filled, with endpoints preloaded.

dline – Subroutine to generate straight line with specified point distribution. Requires two arguments: `curve` (returned) to be filled, with endpoints preloaded; `farray` with desired point distribution.

curdist – Subroutine to redistribure points on a curve by splining the curve and redistributing points on the spline. Requires two arguments: `curve` (returned) to be redistributed; `farray` with desired point distribution.

d_tanh – Intrinsic to generate one or two sided hyperbolic tangent point distribution. Requires five arguments: start of distribution; end of distribution; start spacing; end spacing; number of points in distribution. One zero spacing results in a one sided distribution. Returns `farray`.

makecc – Intrinsic function to generate cubic spline from curve. Requires three arguments: `curve` to be splined; `string` specifying end conditions (*"natural", "quadratic", or "specified"*); `string` specifying spline basis. (*"arc" or "index"*). Returns `pcurve`.

distc – Intrinsic function to distribute points on a pcurve. Requires two arguments: `pcurve` to be distributed on; `farray` containing desired point distribution. Returns `curve`.

Case 1: Tangent–Ogive–Cylinder, No Fins

To demonstrate the simplicity of using SDL libraries, a tangent–ogive–cylinder with no fins is generated. (see Figures 1 and 2) Library calls are made to set the physical dimensions of the body, then the body surface is generated with a call to mk_missile_body(). Note that the i and j dimensions of the generated surface are determined by the dimensions of the surface variable provided as an argument.

The SDL code in Figure 1 would typically be only part of a longer program which would generate the additional surfaces necessary for the generation of a volume grid. Figure 3 demonstrates how one might continue the program and generate upper and lower reflection planes. This code example makes use of standard subroutine library calls to generate lines and arcs, and to do an arclength based transfinite interpolation . Intrinsic functions are used to generate point distributions and parametric representations of curves. The results are shown in Figure 4.

Case 2: Tangent–Ogive–Cylinder, w/ sting, fins.

In this example, a much more complex surface is generated. Figure 5 shows, however, that the required SDL code is not much more involved than in the simple example. Note that the NACA library is used to generate an airfoil section, which is then splined to redistribute points. The resulting section, along with positional information, is recorded using mslbody routines. Additional information about the rear ogive and sting are also provided, and the surface is generated with a call to mk_missile_body(). The results are shown in Figure 6.

CONCLUSIONS AND RECOMMENDATIONS

SDL is a powerful language specifically designed for the generation of curves and surfaces. It

provides a maximum degree of flexibility by allowing the end user to indefinitely expand, customize, and improve its surface grid generation capabilities. Because the SDL interpreter is written in standard portable C and does not depend on hardware architecture, it should be easily ported to any hardware platform.

SDL is recommended for use anywhere its unique capabilities are needed. Potential uses range from production work in a grid generation shop to the classroom, where its implementation of basic vector algebra makes it ideal for learning and exploring geometric algorithms.

REFERENCES

1. Thompson, J. F.; and Gatlin, B.: *Program EAGLE User's Manual, Volume II: Surface Generation System*, AFATL-TR-88-117, September 1988.

2. Hearn, D.; and Baker, M.P.: *Computer Graphics*, Prentice-Hall Inc, Englewood Cliffs, NJ, 1986.

3. Abbot, I.H.; and Von Doenhoff, A.E: *Theory of Wing Sections*, Dover Publications Inc, New York, 1959.

```
# This file uses the mslbody.sdl library routines to generate a
# tangent-ogive-cylinder body surface.

load "mslbody.sdl"

surface body[60,20];

set_fwd_ogive(5);  #forward ogive radius is 5.
set_barrel(.75,7.5); #cylinder has radius of 1 and len of 5

mk_missile_body( body );

# body now holds 1/2 missile surface
```

Figure 1: SDL code for Tangent–Ogive–Cylinder.

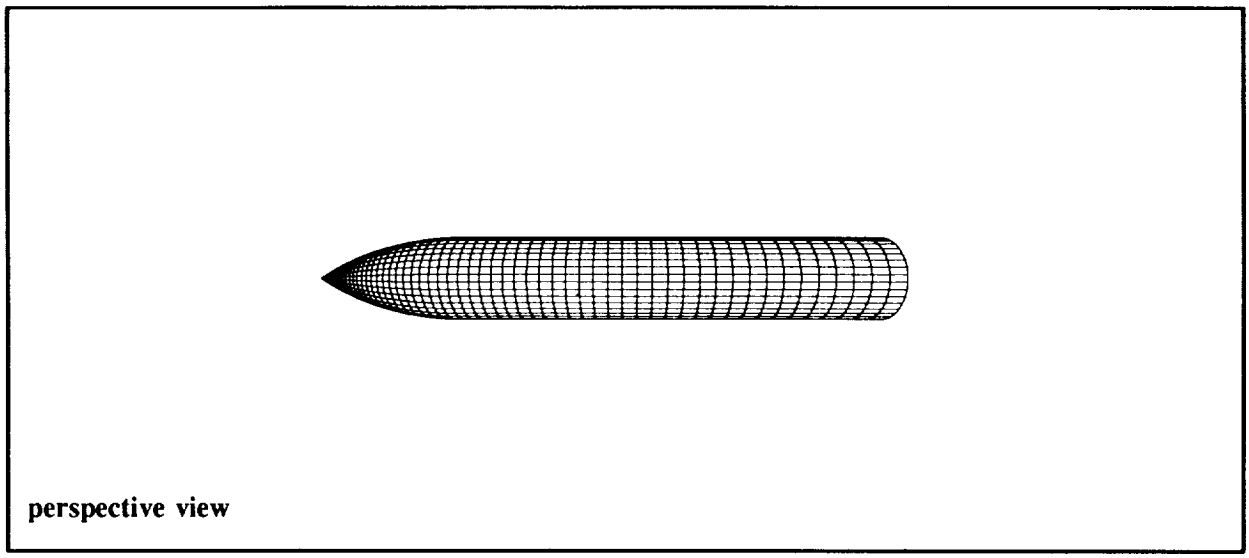

Figure 2: Tangent–Ogive–Cylinder body surface.

```
# ..... continued from above

curve outer_profile[60];              # get the req'd curve
surface uref[60,25], lref[60,25];     # and surface

# build the outer profile

arc( outer_profile[ :45], PI, PI_d2, 1.2 * body[60,1].x );
outer_profile[1].y = 0.0;  #eliminate roundoff error!

outer_profile[60] = body[60,1];
outer_profile[60].y = 1.2 * body[60,1].x;
line( outer_profile[45:] );
curdist( outer_profile, d_tanh(0,1, .07, .01, 60) );

# fill the upper reflection plane

uref[ ,1] = body[ ,1];
uref[ ,25] = outer_profile;
dline( uref[1, ], d_tanh(0,1,.01, 0, 25) );
dline( uref[60, ], d_tanh(0,1,.01, 0, 25) );

transfinite( uref );

# fill the lower reflection plane

lref = uref;
lref.y *= -1;
```

Figure 3: SDL code for Tangent–Ogive–Cylinder reflection planes.

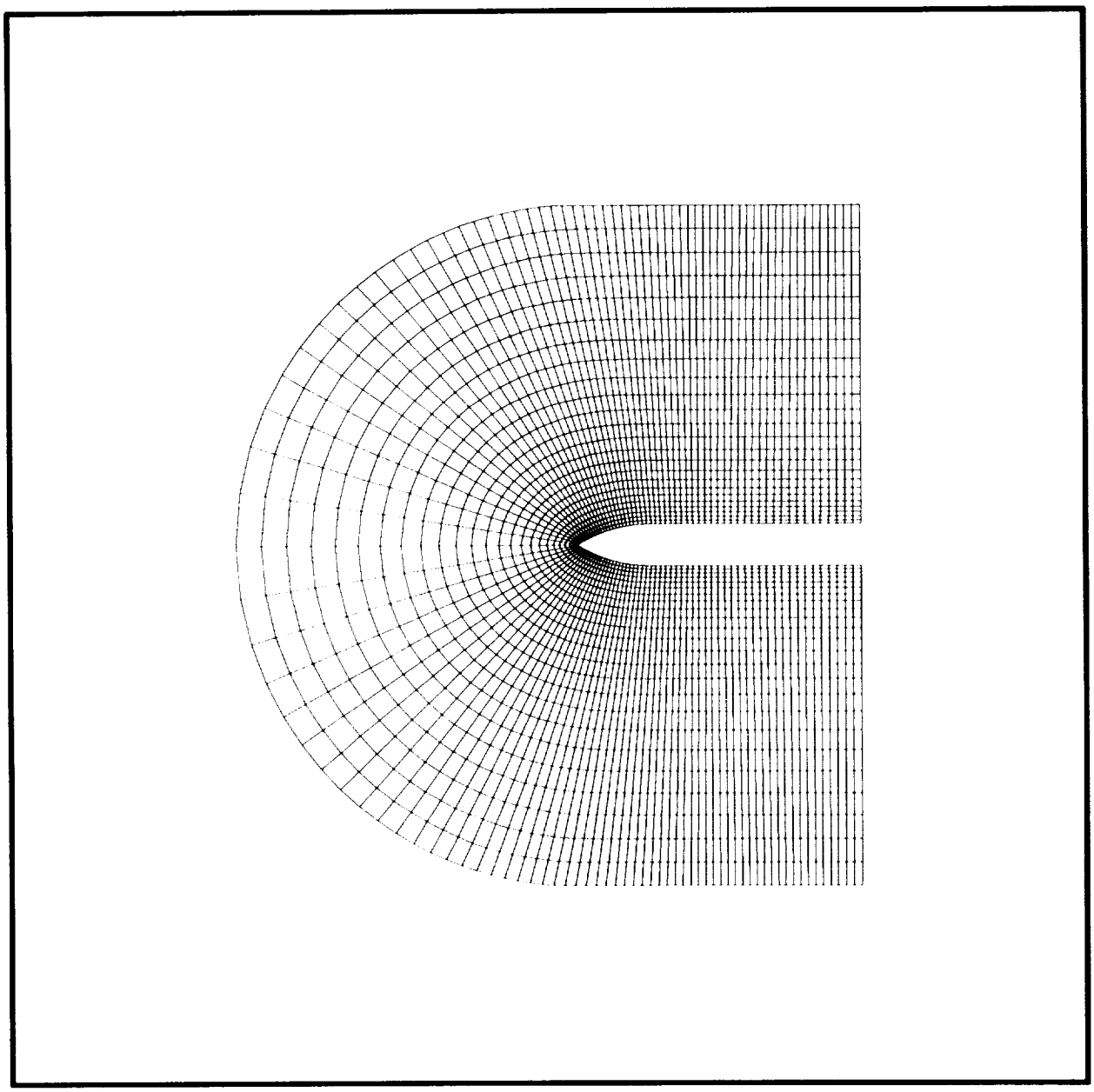

Figure 4: Tangent–Ogive–Cylinder reflection planes.

```
# This file produces a tangent-ogive-cylinder-ogive body, with sting
# and allowance for 4 NACA 0012 fins.

load "stdsubs.sdl"
load "mslbody.sdl"
load "naca.sdl"

# main body parameters

surface body[90,11];
float x_fwd_shoulder;

set_fwd_ogive( 5);
set_barrel(.75,7.5);
set_rear_ogive( 3 );

get_fwd_shoulder( x_fwd_shoulder );

# generate fin profile and set fin parameters

curve naca_profile[200], root[20];

naca4(naca_profile,0,0,12);
pcurve proot = makecc( naca_profile, "quadratic", "arc" );
root = distc( proot, d_tanh( 0,1,0.05,0, 20 ) );

set_fin_root( root );
set_fin_cnt(4);
set_fin_loc(x_fwd_shoulder + 6.3, 45);    # fin starts at i = 40
set_off_fin_sp( .03 );

# set sting parameters

set_sting( .4, 10, 15 );

mk_missile_body( body );

# body now holds 1/4 missile surface
```

Figure 5: SDL code for Tangent–Ogive–Cylinder–Ogive.

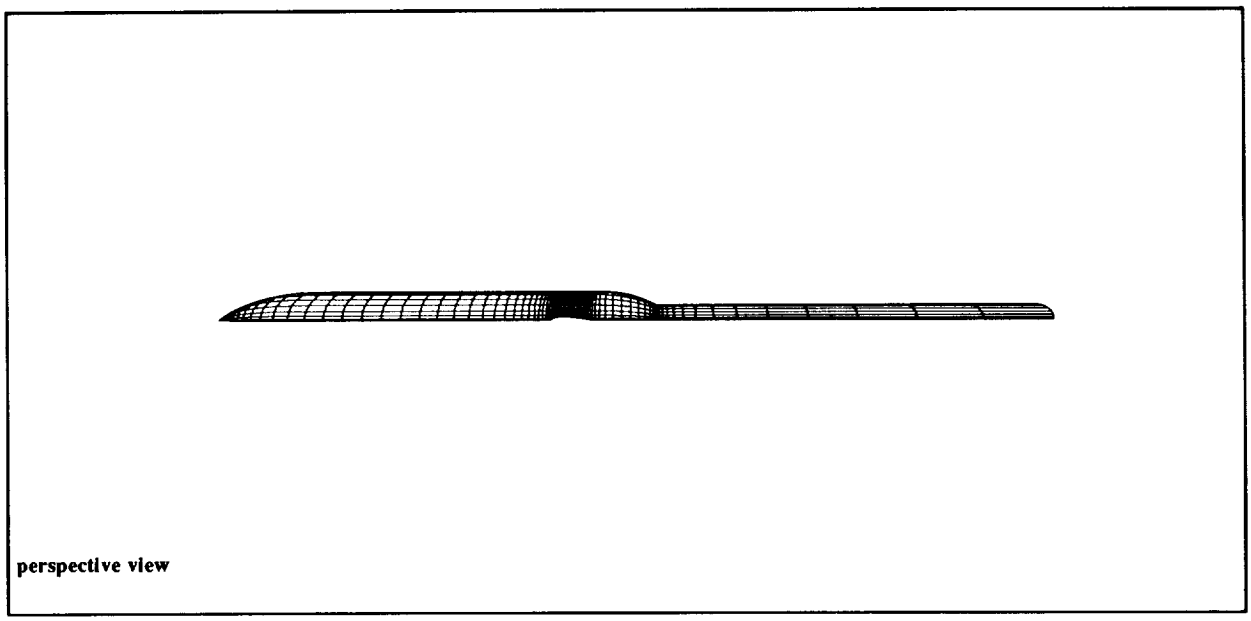

Figure 6: Tangent–Ogive–Cylinder–Ogive body surface.