

ClassWar

Class Language Application Support System Within AutoCAD. Really!

An Atlast Application

An embedded object oriented programming environment implements user-defined entities in AutoCAD. Both objects and methods are completely portable. Optionally, methods can be implemented in C to protect proprietary algorithms or maximise performance.

by John Walker
May 6th, 1990

The world is divided into two classes, those who believe the incredible, and those who do the improbable.

OSCAR WILDE, 1893

STARTING WITH THE ORIGINAL prototype of AutoLisp in 1984, we've been moving toward providing general user-defined entities in AutoCAD. Even though the goal was clear, we knew that much groundwork would be necessary before AutoCAD could provide its users such a facility. We needed a high performance programming language, since AutoLisp was far too slow for applications more computationally intense than macro language usage, and we needed a memory architecture that would provide the storage required by a compiled language. We needed to extend our original ASCII attributes for blocks to allow efficient binary storage of arbitrary attributes on any entity. We needed to permit applications to create AutoCAD objects far more efficiently than by submitting commands. And finally, we needed a way to embed object definitions within an AutoCAD database so user-defined entities would be just as transportable as block definitions.

During the development of Release 11, essentially all of the long-term goals we've been working toward for so long have been accomplished. The development of AutoCAD 386 and AutoCAD for OS/2 have made large memory architectures readily accessible to the overwhelming majority of our customer base, freeing us of the extensibility and performance shackles of the original MS-DOS memory architecture (all other AutoCAD implementations are, and have been since inception, large memory architectures). With adequate storage at hand, ADS provided access to AutoCAD facilities from compiled C code, eliminating the interpretive overhead of AutoLisp. Extended Entity Data (entity attributes) allowed applications to attach a wide variety of application data to database objects, data automatically adjusted by AutoCAD when the object undergoes affine transformations. The `ads_entmake()` facility,

along with its ability to dynamically create anonymous geometry blocks, provides a high performance way for applications to add objects to an AutoCAD database.

The realisation that these features enabled applications to extend the basic entity repertoire of AutoCAD in a coherent, almost seamless way, led to the original Eagle integrated solid modeling prototype project, which has resulted in the development of the AME solid modeling package now scheduled for shipment with AutoCAD Release 11. Several subsequent projects to add more powerful modeling and analysis capabilities to AutoCAD, such as NURBS, a constraint manager, and 2D CSG and FEM, also exploit these facilities to extend AutoCAD.

These projects have demonstrated that AutoCAD is extensible through the addition of C coded applications that embody the calculation code for their speciality, but store their data in an AutoCAD database and rely on AutoCAD for user interface and for common geometrical operations. While this has gone a long way toward our original goals for user-defined objects, some desiderata remain undone. To extend AutoCAD's philosophy of open architecture fully into the domain of application-defined objects, we need to adopt many of the principles which have become the goals of "object oriented programming," embodying them in the structure we encourage (but do not require) application developers to use. By providing this framework we can promote interoperability among applications, allow users to build upon applications just as they have done with the primitives provided by AutoCAD, eliminate the penalty for objects defined outside the "AutoCAD core," and hence avert the need for it to grow in size and complexity through time. We can allow developers to draw the line between the open and proprietary parts of their applications at will—governed by their strategy and the norms of the marketplace rather than by a rigid set of rules laid down by our implementation.

CLASSWAR (Class Language Application Support System Within AutoCAD, Really!) is an attempt to achieve all of these goals, to fuse the methodologies of object oriented programming with those of geometric modeling, to create an application environment of reusable pieces where users and developers can easily create customised modeling environments by selecting and combining components, and to avert the chaotic and bewildering consequences that seem inevitable were hundreds of mutually incompatible applications to emerge. Building upon the new AutoCAD facilities in Release 11 and incorporating many of the software components I've been developing for the last several years against this day, CLASSWAR provides a true object oriented framework for AutoCAD applications, one that will automatically and compatibly benefit from the internal restructuring of AutoCAD along object oriented lines anticipated in the next several major releases.

In the large, CLASSWAR can be seen as a unifying framework for all AutoCAD applications, bestowing the benefits of interoperability and user extensibility upon all that conform to its standards. In the small, CLASSWAR is an ADS application which can be shipped in either binary or source code form with AutoCAD Release 11 which allows simple user-defined entities to be created, manipulated, and exchanged among users. These views describe the same piece of software; the distinction is one of perception, strategy, and consequent market positioning. I am confident enough in the wisdom of our users that I'm sure that regardless of how much or how little fanfare accompanies the delivery of CLASSWAR into their hands, they will rapidly determine its true value and apply it accordingly.

CLASSWAR Overview

Note: *To those experienced with conventional AutoCAD applications, or familiar with object oriented programming languages but not AutoCAD, the concepts of CLASSWAR may seem alien and difficult to master. CLASSWAR is a system in which much of the power is implicit in its programmability and the ability to build on previously-defined components. Like most such systems it may be difficult to come up to speed in its use from a description of its commands and language features. To assist in mastering the system, I produced a 90 minute video demonstration titled "CLASSWAR—THE VIDEO." An NTSC/VHS copy of this tape is available in the Autodesk technical library in Sausalito. Notwithstanding the tacky production values and crashingly boring dramatic merits of this opus, I highly recommend you view it before reading this document.*

Basic concepts

Classes. CLASSWAR defines and manipulates user-defined entities which are stored in the AutoCAD database. Each of these entities is an *instance* of a *class*. The entity consists of an AutoCAD anonymous block definition and a single insertion of it; the block definition contains the (possibly void) geometrical object specified by this instance. Attached to the insertion of the block are extended entity data which contain the *instance variables* specifying the object. The definition of these variables, the relationship between them and the geometry of the object, and all operations which manipulate them are given in the *class definition*. The class definition is specified by a block in the same drawing that contains instances of it. The class definition can either contain the actual CLASSWAR source code defining the class, or can reference a definition of the class given in an ASCII text file. A class definition consists of declarations of its instance variables, which specify the properties of a specific instance, its *class variables*, which store data common to all instances of the class, and *method definitions*: executable CLASSWAR code specifying the operations that manipulate instances of the class.

Methods. Method definitions can either be given as source code within the class definition or can be *external methods*, implemented in other ADS applications and called with `ads_invoke()`. Methods, however specified, can be invoked from within other CLASSWAR programs or, if declared as *command methods*, can be called as an AutoCAD command with the method's name. Methods can have *arguments* which are passed on the ATLAST stack when they're invoked within programs and obtained by prompting the user when the method is used as an AutoCAD command. Any number of different class definitions may define methods with the same name; CLASSWAR automatically executes the correct method based on the class an object belongs to. Methods used as AutoCAD commands may be applied to selection sets containing objects of different classes, and CLASSWAR will obtain the information from the user required by the union of the argument lists of the method definitions for all objects in the selection set, combining identical argument requests to avoid duplication, and then run the correct method for each selected object, passing it the arguments it expects.

Variables. Class and instance variables may be either *primitive data types*, which include all the data types representable as AutoCAD entity attributes (such as real numbers, integers, strings, distances, locations in 3-space, orientation vectors, scale factors, etc.), or instances of

other classes. The ability to include class instances within other class definitions allows assembling composite objects from previously defined pieces. When an instance is included within another class definition, all of its *public* methods may be applied to it by methods of the class including it.

Inheritance. Class definitions may *inherit* (or, in other words, be *derived from*), previously defined classes. When a class inherits the properties of another, it contains all the instance and class variables, and methods of its *parent* class, although some of these may not be directly accessible if declared *private* in the parent class definition. *Protected* methods and data are accessible to classes derived from the parent, but not to those that simply include instances of the class—they may use only *public* items. A derived class inherits all the methods of the parent class, but may redefine, or *overload*, any of them with a method of its own, should the derived class wish a different action for that method. Even if a method is overloaded, the parent’s method with the same name may still be called by explicitly specifying invocation of that method. A derived class may either allow all the methods of the parent to “show through,” in which case the class is considered *publicly* derived, or may *encapsulate* (hide) them, being then deemed *privately* derived.

Constructors. A class definition may specify a *class constructor* or *newclass method*, executed when the class is initially defined; this method is frequently used to initialise class variables. In addition, an *instance constructor*, or *new method*, can be specified. It is executed whenever a new instance of the class is created and typically sets instance variables to default values for new objects of that class. If a class definition contains an *acquire method*, an AutoCAD user can create instances of the class simply by entering the class name as an AutoCAD command. The acquire method normally prompts the user for information that defines the instance.

Geometry. Most class instances will have a geometrical representation that is visible and manipulable with AutoCAD (although this is not required). A class definition specifies the geometry that represents it in its *draw method*, automatically invoked when an instance of the class is created or modified. The draw method can create AutoCAD geometry using three distinct facilities, or any combination thereof. The most basic mechanism provides direct access to the `ads_entmake()` facilities of ADS. A set of ATLAST definitions collectively referred to as the ATLAST ADS bindings allow method code to assemble

result item chains and enter them in the database. The ADS bindings are derived from, and essentially compatible with, the mechanisms introduced for manipulating DXF information in DXFIX. The ADS binding package allows access to all objects in AutoCAD’s repertoire and precise control over the entities created, but requires corresponding care on the user’s part to achieve the desired results. A higher level of abstraction is provided by the second geometric definition facility, the *SGLIB binding*. This package provides ATLAST definitions for all the functions of my C language Simple Graphics Library, providing both comprehensive three dimensional linear algebra and geometric facilities, plus tools for generating vectors, faces, meshes, and commonly used objects including all the regular polyhedra and spheres. The linear algebra components of the SGLIB package can be used to calculate components for the lower level ADS package as well. The highest level of geometry creation is embodied in the *turtle geometry package*. Based upon the turtle procedure notation given in the book *Turtle Geometry*,¹ but extended to three dimensions along the guidelines given in that book, one or more turtles can be used to construct arbitrary geometric features, “leaving tracks” as they draw either in the form of ephemeral vectors that disappear on the next **REDRAW**, Lines, Polylines, or polygonal faces traced out by the turtle as closed polygons. The turtles provided by CLASSWAR are neither teenage, mutant, nor Ninja.

Using CLASSWAR

Installing CLASSWAR in your AutoCAD environment and using it is relatively simple; the power of CLASSWAR comes from the classes you create using it, not from its built in commands. To activate CLASSWAR, use the AutoCAD command:

```
(xload "class")
```

to load the application. If you want CLASSWAR automatically loaded whenever you run AutoCAD, add it to the list of standard applications in your `acad.ads` file. After CLASSWAR is loaded, it immediately reads configuration information and the definition of the “object superclass” from the files `config.cls` and `object.cls` respectively. These files are supplied with CLASSWAR and should not be modified as they interact intimately with the `class` application (substantial parts of CLASSWAR are written in its own programming language and are contained in these files). The `config.cls` and `object.cls` files may

¹Harold Abelson & Andrea diSessa, *Turtle Geometry*, MIT Press: Cambridge (Mass), 1980, 1986.

be placed anywhere on AutoCAD's search path, and are normally kept with other support files such as `acad.pat` and `acad.lin`.

CLASSWAR stores class definitions in a temporary memory area called the "heap." The heap must be sufficiently large to simultaneously hold all the class definitions used by a drawing. If the heap is too small, CLASSWAR will run out of memory attempting to compile the class definitions and will disable all class commands for the duration of that drawing editor session. The heap is specified in terms of items, each 4 bytes in length. The default heap size of 5000 items (20K bytes) is adequate for most simple class definitions. You can specify the heap size by defining the environment variable "CLASSHEAP" as the desired heap size in items. To determine how much heap is used by the classes referenced by a drawing, enter the command `/ MEMSTAT` at the AutoCAD command prompt after all classes have been loaded. Be sure to include the blank after the slash in this command. Class definitions are stored in a very compact form, so large amounts of heap are rarely required except for classes that contain large amounts of floating point co-ordinate data.

Loading classes

You can define classes interactively within the AutoCAD drawing editor or import their definitions from text files prepared with your favourite text editor. Classes can be stored either within the AutoCAD drawing file or externally in text files.

The CLASSDEF command

The **CLASSDEF** command defines a class whose definition is stored within the AutoCAD drawing file. If all class definitions used by a drawing are embedded within it, and you send the `.dwg` file to another AutoCAD user who's installed CLASSWAR, that user can perform all the operations defined by the embedded classes.

To define an embedded class, enter the **CLASSDEF** command. AutoCAD presents you with an attribute editing window. Enter the class name in the first line of this window; the class name must not duplicate the name of any other class in the drawing, and should conform to AutoCAD's conventions for object names (such as layer, line type, and block names).

You can either directly type the class definition in to this window, using the CLASSWAR programming language described later in this document, or import a class definition

in that language from an external text file. To load a text file, enter:

`<filename`

as the first line of the class definition. When you pick OK, the class definition will be loaded from the named file. If no explicit path is specified in *filename*, it is searched for using the same search path AutoCAD uses for block insertions. If no extension is given `.cls` is used. Files loaded with this mechanism may use any of the end of line conventions accepted by AutoCAD for text files it reads.

The CLASSFILE command

Classes defined with **CLASSDEF** are stored within the AutoCAD drawing file. While this makes the drawing self-contained and easy to interchange with other users, it may prove inefficient if lengthy class definitions are included in a large collection of drawings. In addition, in some environments users may want to maintain a standard library of class definitions in a central repository so updates to the master set of classes immediately affect any drawing that uses one of them.

The **CLASSFILE** command defines a class used within a drawing but defined in an external text file. Every time the drawing is edited, the class definition is loaded from the text file. If the class definition file cannot be located when the drawing is edited (for example, you've sent the drawing to another user but forgotten to include one of the class definitions it uses), normal AutoCAD commands will be able to manipulate objects of that class but all object-specific commands defined in the class definition will be unavailable.

To declare a class defined in a text file, enter the **CLASSFILE** command. AutoCAD presents you with an attribute edit box requesting the class name and the file where it is defined. Enter the class name, which must be a valid AutoCAD symbol name not used by any previously-defined class, and the file name containing the class definition. If you leave the file name blank, the class will be loaded from a file with the same name as the class with the default extension of `.cls` appended, located using the AutoCAD search path. If a file name is given, the AutoCAD search path is used to find it unless an explicit path is specified. Class definition files may use any of the end of line conventions accepted by AutoCAD.

Editing classes: the CLASEDIT command

The CLASEDIT command allows you to modify the definition of any class embedded within a drawing (defined with CLASSDEF), or direct a class definition in an external text file to load a different file. Enter the CLASEDIT command, and AutoCAD will prompt you:

Select class instance, or CR to name class:

The easiest way to specify which class you wish to edit is just to pick an object of that type on the graphics screen. If you'd rather specify the class by name (for example, no object of the desired class is visible, or none is present in the drawing), just press **Return** and you'll be asked:

Class name or ?:

Enter the class name as specified by CLASSDEF or CLASSFILE. You can list the classes present in the drawing by entering "?", after which you're re-prompted to select a class to edit.

Once the class has been specified, an attribute editing box appears containing the actual class definition (if created with CLASSDEF), or the class name and file that defines it (if loaded with CLASSFILE). Make whatever modifications you like to the class definition and pick the **OK** box. CLASSWAR will recompile the class definition and update all objects of that class to conform to the new class definition. Picking **Cancel** discards all changes made to the class definition.

If the class is defined in an external file, CLASEDIT only allows you to change the file name defining the class, but not the actual class definition. If the class definition has changed since it was last loaded (for example, you've edited it in another window of a multitasking system), picking **OK** will cause the new version to be used within the drawing, even if the file name hasn't changed.

The CLASSUPD command

If you modify a class definition in an external text file while the drawing editor is running (by editing it in another window of a multitasking system), the changes are not normally applied until AutoCAD reloads the class the next time the drawing is edited. To immediately load the updated class definition, you can use CLASEDIT on the updated class as described above, or just enter the CLASSUPD command. CLASSUPD causes *all* externally defined classes to be reloaded and recompiled and guarantees that AutoCAD is using the most recent versions. Un-

like CLASEDIT, however, CLASSUPD does not regenerate existing instances of the classes it reloads.

Examining objects

Once you've created objects that are instances of classes you've loaded, you normally manipulate them with the methods defined by those classes. A standard set of commands, INSPECT, INSPCLASS, SPY, and SPYCLASS, which operate on all objects created with CLASSWAR, allow interactive examination and modification of the variables that define the properties of an object.

INSPECT—Examine instance variables

The INSPECT command allows you to examine and, if you wish, change any of the public instance variables defined by a class. When you enter INSPECT you're asked to pick an instance. A dialogue box appears which displays the public instance variables of that instance. If you'd like to change any of them, just enter the new values in the dialogue box and pick **OK**. To leave the variables unchanged, pick **Cancel**. If you pick **OK**, the object will be regenerated to reflect the changes you've made.

Since INSPECT operates directly upon the instance variables, bypassing all consistency checking that may be done in methods of the class, it violates the fundamental principle of encapsulation of data that's key to object oriented programming. Hence INSPECT should be viewed largely as a debugging feature; it is extremely convenient when developing new classes.

INSPCLASS—Examine class variables

INSPCLASS works precisely like the INSPECT command, but examines and modifies class variables rather than instance variables. You specify the class to be inspected by selecting an instance of it, even though a particular instance isn't relevant to the class variables. If a change to a class variable should affect all objects of that class, it does *not* automatically get applied. Your class should define methods for such operations and not rely on INSPCLASS to change such variables.

SPY—Snoop on instance variables

The INSPECT command only lets you see and change variables declared **PUBLIC**: in their class definitions. Since

such variables are normally accessible as fields in programs that use objects of the class, changing them is generally considered safe. **SPY** behaves identically to **INSPECT**, but also lets you examine and change **PRIVATE:** and **PROTECTED:** variables. **SPY** is provided entirely for the convenience of developers debugging class definitions and should not be used in the normal operation of applications.

SPYCLASS—Snoop on class variables

SPYCLASS is the snoop analogue of **INSPCLASS**; it behaves identically but lets you examine and change **PRIVATE:** and **PROTECTED:** class variables as well those declared **PUBLIC:**.

External classes: CLASSTOC

A simple **CLASSWAR** class definition is contained entirely in the drawing or the text file named with **CLASSFILE**. Complicated applications that perform computationally intense tasks that would be too slow if implemented in ATLAST, or proprietary applications that developers don't want to release in source code form, can consist of a **CLASSWAR** class definition that links to one or more concurrently loaded ADS applications that implement *external methods* of the class.

The interface between **CLASSWAR** and the external method is mediated by definitions in a C language header file automatically generated by **CLASSWAR** from the class definition. To create the linkage definition for a class, enter the **CLASSTOC** command. You're prompted to select the class either by pointing to an instance of it or by entering its name in the same manner used by the **CLASEDIT** command. You're then asked for the file name into which the definitions should be written; this defaults to the same name as the class, in the current directory. Here's an example of the **CLASSTOC** command.

```
Command: classtoc
Select class instance, or CR to name class:
Class name or ?: mountain
C header file name <mountain.h>:
Command:
```

Details of how the interface file created by **CLASSTOC** is used to implement external methods appear later in this document.

Embedded ATLAST

Since **CLASSWAR** is an ATLAST application it can provide direct access to the underlying facilities of ATLAST within the AutoCAD drawing editor. You can interactively "talk to" the ATLAST interpreter by entering the command:

ATLAST

The ATLAST interpreter prompt (extended to show the current stack depth in brackets) appears, and you can enter any ATLAST program you like. Entering a blank line returns to the AutoCAD command prompt. For example, here a user employs ATLAST as a desk calculator to solve a trigonometry problem.

```
Command: atlast
Atlast[0]-> : dtr 180.0 f/ pi f* ;
Atlast[0]-> 90.0 dtr f. cr
1.5708
Atlast[0]-> 60.0 dtr sin 12.5 f* f. cr
10.8253
Atlast[0]->
Command:
```

If you only want to enter a single line of ATLAST text, you can use the "/" command, which silently accepts a line terminated by a carriage return and executes it as an ATLAST program. For example, using the **dtr** definition we've previously entered, we calculate another trigonometric quantity.

```
Command: / 30.0 dtr sin 24.0 f* f.
12
Command:
```

Class Definition Programming

To define a **CLASSWAR** class, you write a program that declares *variables* which specify the properties of objects of that class, and executable code for the *methods* that operate on the objects. You can use objects of other classes as members of new classes you define, or *derive* classes from preexisting classes, *inheriting* their variables and methods into the new class.

CLASSWAR is built on top of the ATLAST programming language and shares its notation and built-in primitives. This manual assumes you have a rudimentary knowledge of ATLAST and does not repeat information given in the

ATLAST documentation. Please refer to that manual for details of ATLAST concepts and programming.

Preliminary tour

Before we wade into the plethora of minuscule details that characterise any programming language, it's worth walking through a simple program to get our bearings. The following is a definition of a simple CLASSWAR class: one that implements a regular polygon object.

```
( Polygon class definition )

PUBLIC:
  ( Total polygons in drawing )
  STATIC INTEGER polycount
  ( Radius of circumscribed circle )
  REAL size
  ( Number of sides )
  INTEGER nsides

PRIVATE:
  ( Unique sequence number )
  INTEGER polyseq

TEMPORARY:
  ( Angle increment )
  REAL anginc
  POINT kp
  1.0 2.0 0.0 kp POINT!

: deg2rad
  Pi f* 180.0 f/
;

PUBLIC:

method draw
{
  360.0 nsides @ float f/ 2dup anginc 2!
  penup
    nsides @ 1 and 0= if
      2dup 2.0 f/ 90.0 f+ 2dup right
      size 2@ forward left
    else
      90.0 2dup right size 2@ forward
      anginc 2@ 2.0 f/ f+ left
    then
  pendown
  deg2rad cos 1.0 f- fnegate size 2@
  2dup f* 2.0 f* f* sqrt
  nsides @ 0 do
    2dup forward anginc 2@ left
  loop
  2drop
}
```

```
PRIVATE:

( Class constructor )

method newclass
{
  1000 polycount !
}

PROTECTED:

variable dnsides 3 dnsides !
2variable dsize 1.0 dsize 2!

method acquire
{
  integer "Number of sides"
  dnsides default arg
  0<> if false return then nsides !
  distance "Edge size"
  dsize default arg
  0<> if false return then size 2!
  1 polycount +! polycount @ polyseq !
  ( Save last acquisition parameters as
    defaults for the next )
  nsides @ dnsides !
  size 2@ dsize 2!
  true
}

PUBLIC:

command method grow
{
  1.5 size 2@ f* size 2!
}

command method shrink
{
  2.0 3.0 f/ size 2@ f* size 2!
}

command method more
{
  1 nsides +!
}

command method less
{
  nsides @ 3 > if
    -1 nsides +!
  then
}

( Instance constructor )

method new
{
```

```

8 nsides !
0.5 size 2!
10 polyseq !
}

```

Now let's scrutinise this class definition, pointing out features of interest along the way. This should give you a general feel for CLASSWAR programming, and give you a framework on which to hang all the concepts to be described in the sections that follow.

(Polygon class definition)

ClassWar shares the syntactic conventions of Atlast, including its comment delimiters.

PUBLIC:

This declares the following variables as public—accessible to all users of the class, and by the INSPECT command.

(Total polygons in drawing)

STATIC INTEGER polycount

The word STATIC at the start of the declaration marks this variable as a class variable. A single copy of it is shared by all instances of the class. INTEGER declares a 32 bit integer variable with the name that follows.

(Radius of circumscribed circle)

REAL size

This declaration isn't STATIC, so it's an instance variable. Each object of this class has its own private copy of this variable, declared as a double precision floating point number.

(Number of sides)

INTEGER nsides

Here's an INTEGER instance variable.

PRIVATE:

The preceding variables were public: generally accessible. The variables that follow are private. They are visible only within this class definition itself.

(Unique sequence number)

INTEGER polyseq

We've declared the polygon sequence number (a unique identifier we'll attach to every polygon we create in the drawing) to be a private instance variable: stored with each polygon, but visible only within this class definition.

TEMPORARY:

The variables that follow are temporary variables that retain their values only during the drawing editor session; they're not stored in either the instance or the class. Temporary variables are normally used for intermediate results calculated within methods.

(Angle increment)

REAL anginc

POINT kp

The formal variable declarations end here. All non-TEMPORARY variable declarations must precede the executable code of the class definition.

1.0 2.0 0.0 kp POINT!

The following Atlast definition converts degrees to radians. Since Atlast

underlies ClassWar, you can use all of its facilities in class definitions.

```

: deg2rad
  Pi f* 180.0 f/
;

```

The following methods are PUBLIC, callable by any user of the class. You don't have to group public, private, and protected items together; you can switch modes as often as you like.

PUBLIC:

This is the DRAW method for the polygon class. Every class that has a geometric representation must have a draw method to generate it. This draw method uses the turtle to trace out the polygon.

method draw

{

Calculate the angle to turn between polygon sides.

360.0 nsides @ float f/ 2dup anginc 2!

Getting the polygon in the right place is more complicated than drawing it! We raise the pen and move to the calculated starting point of the first polygon edge.

penup

nsides @ 1 and 0= if

2dup 2.0 f/ 90.0 f+ 2dup right

size 2@ forward left

else

90.0 2dup right size 2@ forward

anginc 2@ 2.0 f/ f+ left

then

pendown

Now to draw the polygon, all we need to do is take the number of steps given by the number of sides, turning the turtle left the calculated amount between sides.

deg2rad cos 1.0 f- fnegate size 2@

2dup f* 2.0 f* f* sqrt

nsides @ 0 do

2dup forward anginc 2@ left

loop

2drop

}

The following method is private—accessible only within this definition.

PRIVATE:

(Class constructor)

A NEWCLASS method, if defined, is called just once: when the class is initially defined within the drawing; this is referred to as the class constructor. We use a NEWCLASS method to initialise the sequence number class variable to 1000.

method newclass

{

1000 polycount !

}

The following method is protected. It can be accessed from within this definition and in classes derived from this class, but not in code that declares instances of the class.

PROTECTED:

Regular Atlast variable definitions can be used as temporary variables. They are always considered temporary and private, regardless of where declared in the class definition.

```
variable dnsides 3 dnsides !
2variable dsize 1.0 dsize 2!
```

If an ACQUIRE method is defined by the class, an AutoCAD command with the same name as the class is defined to create objects of the class. The acquire method is responsible for prompting the user, if appropriate, for the properties of the object and storing them in the instance variables of the object. The draw method is called automatically once the acquire method is done.

```
method acquire
{
```

The ARG primitive is used here to obtain the polygon's number of faces and size. Each argument request specifies its type, the user prompt, and the default value if none is entered by the user. The ARG mechanism is very flexible—used both within methods and to obtain arguments for methods activated through selection sets of objects.

```
integer "Number of sides"
dnsides default arg
0<> if false return then nsides !
distance "Edge size"
dsize default arg
0<> if false return then size 2!
```

We increment the POLYCOUNT class variable and use it to assign a unique sequence number to the polygon. The change to the class variable is automatically saved in the database by ClassWar.

```
1 polycount +! polycount @ polyseq !
```

The user specifications for this polygon are saved in temporary variables and become the defaults for the next time. Since these are temporaries, they're retained only for the current drawing editor session. If we'd made them class variables (STATIC), they would be remembered from session to session.

```
( Save last acquisition parameters as
defaults for the next )
nsides @ dnsides !
size 2@ dsize 2!
```

The ACQUIRE method leaves a status on the stack indicating whether the object was successfully acquired. This method leaves TRUE to indicate success. If it leaves FALSE on the stack, the AutoCAD acquisition command terminates with no action.

```
true
}
```

The following methods are public.

PUBLIC:

This method is declared as a "COMMAND METHOD". This means that in addition to being used within ClassWar code, it can be invoked as an AutoCAD command. When a command method is entered, a selection set is requested, the objects in are sorted by class, and the appropriate method is invoked for each object in the selection set.

```
command method grow
{
```

Our GROW method just multiplies the size by 1.5. ClassWar automatically stores the change to the instance variable with the entity and calls the draw method to update the geometry on the screen.

```
1.5 size 2@ f* size 2!
```

```
}
```

Similarly, the SHRINK method sets the side to 2/3 of its former size.

```
command method shrink
{
2.0 3.0 f/ size 2@ f* size 2!
}
```

Our MORE method just increments the number of sides.

```
command method more
{
1 nsides +!
}
```

And the LESS method decrements the number of sides, unless the figure is already a triangle.

```
command method less
{
nsides @ 3 > if
-1 nsides +!
then
}
```

```
( Instance constructor )
```

A NEW method, if present, is the instance constructor. The instance constructor is called whenever a new instance of the class is created, whether by an AutoCAD command invoking the ACQUIRE method, or by the declaration of an instance of this class within another class definition.

```
method new
{
8 nsides !
0.5 size 2!
10 polyseq !
}
```

This the entire definition of the polygon class, as supplied in the file POLY.CLS in the CLASSWAR distribution. When this class is loaded, the following commands become available:

POLY. Draws a polygon. The user is asked for the number of sides and the size of the polygon. The polygon is centred around the insertion point.

GROW. The polygon is increased in 50% in size.

SHRINK. The polygon is reduced to 2/3 of its former size.

MORE. The number of sides of the polygon is increased by one.

LESS. The number of sides of the polygon is reduced by one, unless the polygon is already a triangle.

This definition, then, is the complete specification of a new entity for AutoCAD and the commands that operate on it. All the standard AutoCAD commands such as MOVE, COPY, ERASE, ROTATE, etc., will also work as expected with the new object.

Deriving a class

Much of the power of object oriented programming stems from the ability to create *derived* classes, which inherit variables and methods from a parent class, allowing methods in the parent class to be redefined (or overloaded), and new data and methods to be added. To complete our overview of CLASSWAR definitions, let's walk through an example of a derived class. The following definition is supplied on the CLASSWAR distribution as DPOLY.CLS.

This class implements polygons that behave just like those of its parent class, POLY, but with each polygon labeled at its centre with the number of sides it contains. Rather than duplicating the entire definition of POLY, we implement this new object with the following derived class definition.

(Labeled polygon class definition)

Here is where we inherit the properties of the POLY class. The word "poly" is the name of the POLY class (as opposed to "poly," which is the declaring word used to declare instances of that class). The PUBLIC specification causes variables and methods declared PUBLIC: and PROTECTED: to retain those attributes in the derived class. Were PUBLIC not specified here, the inherited components would be considered PRIVATE: in this class. DERIVED declares the class as derived. All the non-PRIVATE variables and methods of the parent class are defined within this class.

```
:poly PUBLIC DERIVED
```

A little utility Atlast definition to add a new group to the text entity and set its value from the stack.

```
: tack
  dup addgroup setgroup
;
```

This temporary string (declared as a simple Atlast string) is used to edit the number of sides in the polygon.

```
10 string ns
```

```
PUBLIC:
```

All of the methods of the existing POLY class can be used as-is, except for the DRAW method. We want this class to include text to label the polygon with its number of sides. Consequently, we redefine the DRAW method to include the text. We use the direct ADS binding primitives

to assemble a group list for the Text entity and add it to the database.

```
method draw
{
  clearitem
  "TEXT" 0 tack
  xcor ycor zcor 10 tack
  xcor ycor zcor 11 tack
  size 2@ 4.0 f/ 40 tack
  nsides @ "%ld" ns strform ns 1 tack
  4 72 tack
}
```

Add the newly-assembled text entity to the database.

```
ads_entmake drop
```

Now that the text that labels the polygon has been generated, we need to draw the polygon itself. Rather than physically copying the draw code from the POLY method, we just use the DRAW method of the parent class to get the job done. If, however, we just used DRAW, we'd recursively call this very DRAW method, recursing until a stack overflow abruptly brought the festivities to an end. The following line uses THIS, which places the current instance on the stack, then uses the construction "DRAW <-", which causes the parent class' DRAW method to be called instead.

```
  this draw <- drop
}
```

Variable declarations

The first section of a class declaration defines the class variables (one copy shared by all members of the class) and instance variables (one copy per object) used in the class. In addition, temporary variables used within the class declaration but not saved in the drawing may be declared in a compatible fashion. All variable declarations must appear before the first method definition.

Simple variable types

The data types available as CLASSWAR variables correspond directly to those provided by AutoCAD's extended entity data facilities. Several data types share the same storage allocation and representation (for example, real numbers and distances) but behave differently when transformed by AutoCAD commands. Consequently, it's important to declare the correct type and not indiscriminately use variables of the same generic type. Otherwise, objects of your class will misbehave when moved, rotated, or scaled with AutoCAD's built in commands.

The standard variable types are as follows:

INTEGER A 32 bit signed integer. Not transformed by AutoCAD commands.

REAL A 64 bit floating point number. Not transformed by AutoCAD commands.

SCALEFACTOR A 64 bit floating point number. Adjusted when the object is scaled.

DISTANCE A 64 bit floating point number. Adjusted when the object is scaled.

TRIPLE A triple of 64 bit floating point numbers. Not transformed by AutoCAD commands.

POINT A triple of 64 bit floating point numbers representing a location in three dimensional world coordinate space. Adjusted by AutoCAD when the object is moved, scaled, rotated, or mirrored.

DISPLACEMENT A triple of 64 bit floating point numbers representing a displacement vector in space. Adjusted by AutoCAD when the object is rotated, mirrored, or scaled.

DIRECTION A triple of 64 bit floating point numbers representing a unit length direction vector in space. Adjusted by AutoCAD when the object is rotated or mirrored.

***n* CHARACTERS** A character string with maximum length *n*−1. Not transformed by AutoCAD commands.

POINTER A character string containing an AutoCAD database handle. Transformed when the object is inserted into another drawing with handles.

A variable is declared simply by giving its variable type and the name of the variable. The variable type must be repeated for each variable you declare; it's not possible to declare a list of variables in a single statement. Here are some simple variable declarations.

```
POINT centre
DISTANCE radius
DIRECTION normal
82 CHARACTERS label
```

Instances of classes

You can declare variables which are instances of any previously loaded class definition. These instances are initialised by running the constructor of their class, and you can access any public field and run any public method of the variable's class. For example, if we've loaded the **POLY** and **DPOLY** classes shown above in the introduction, we could declare instances of them within another class as:

```
POLY wannacracker
DPOLY mydpoly
```

Access modes

Variables and methods can be declared with one of four access modes. The access mode of a variable is given by the most recent access mode specification, or private if none has been declared. The access mode specifiers and their interpretations are as follows.

PRIVATE: A private variable is accessible only within the class definition. A private variable is stored with the instance or class but can be examined and changed only by methods of the class. Private variables provide the data hiding which is an essential part of object oriented programming.

PUBLIC: A public variable is generally accessible. It can be referenced within the class definition, by classes derived from it, or from other classes that declare instances of the class. Public data are not hidden and the class definition must be wary of incorrect manipulation of public variables by other programs.

PROTECTED: A protected variable is accessible from within the class that defines it and by any class derived from that class, but not by other classes that declare instances of the class.

TEMPORARY: Temporary variables are private and are not stored with either the class or the instance. They retain their values only for the current AutoCAD drawing editor session and are normally used as a scratchpad for calculations within the methods of the class.

The following set of declarations illustrates the various access modes. Note that an access mode declaration applies to all variables (and methods) declared until the next access mode is specified.

```
PUBLIC:
    POINT centre
    DISTANCE radius
PROTECTED:
    DIRECTION normal
PRIVATE:
    82 CHARACTERS label
TEMPORARY:
    INTEGER numsides
    REAL tempang
```

Class variables

Most variables in class definitions are instance variables—one copy per object of the class. For some purposes, for example assigning unique sequence numbers to every object of the class, you want a variable common to all members of the class. Such a variable is called a “class variable,” and is declared by preceding its declaration with the word “**STATIC**.” The **STATIC** specification causes only the next variable declared to become a class variable; subsequent variables will be instance variables unless also preceded by **STATIC**. In the following declarations:

```
PUBLIC:
    POINT centre
    DISTANCE radius
    STATIC 20 CHARACTERS classid
PROTECTED:
    DIRECTION normal
PRIVATE:
    82 CHARACTERS label
    STATIC INTEGER seqnumber
TEMPORARY:
    INTEGER numsides
    REAL tempang
```

variables “*classid*” and “*seqnumber*” are class variables; all the rest are instance variables or temporaries.

Method definitions

After the variables, the methods of the class are defined. The methods contain the procedural code that operates upon objects of the class, implementing their repertoire of behaviour. Some methods, such as those that initialise the class and instance variables, draw the object when it is created or modified, and obtain the parameters for a new object from the user, have predefined names and functions. Most methods, however, are specific to the class, implementing operations peculiar to objects of that class.

Any number of classes may have methods with the same name. CLASSWAR automatically calls the correct method by examining the type of the object being operated upon and selecting the method with the specified name for objects of its class. Methods may be called from within CLASSWAR programs, given the object on which they are to operate on the stack or, if declared as *command methods*, invoked as AutoCAD commands that operate upon objects selected by AutoCAD’s entity selection mechanism. Methods can have *arguments* (parameters). A

method with arguments obtains its arguments from the ATLAST stack. When a command method with arguments is run, CLASSWAR obtains the arguments by prompting the AutoCAD user for them, combining requests for identical arguments requested by different methods referenced by objects in a selection set. Then, when the command method is run for a selected object, CLASSWAR places the arguments it expects on the stack. Thus, the same method can be used both within CLASSWAR programs and as an AutoCAD command; this makes methods readily reusable when new classes are built upon existing ones.

Methods are declared using the following syntax, where items in [brackets] are optional.

```
[COMMAND] METHOD  name [ (( arguments )) ]
{
    ... method implementation...
}
```

The *name* is the name of the method. If “**COMMAND**” precedes the declaration, this will be the name of an AutoCAD command that invokes the method as well as its internal name within CLASSWAR programs. If the method has arguments, they are declared within an argument list enclosed in double parentheses. Note that spaces must separate these tokens, as they are actually ATLAST definitions. If a method has no arguments, the argument list delimiters should be omitted.

The ATLAST code that implements the method is enclosed in braces. The implementation code is written as a regular ATLAST program, but has access to the variables declared in the class definition and all the additional geometric primitives of CLASSWAR.

Variables declared in the class definition are referenced by name; as with a normal ATLAST variable this places a pointer to the variable on the stack. You can load or store the value in the variable using that pointer. When you reference a variable, CLASSWAR determines whether it is an instance, class, or temporary variable and gives you a pointer to the correct area automatically. Since variable offsets are calculated at compilation time, access to variables of a class definition is very efficient.

You normally leave a method by executing off the end of the definition. If you want to exit from the method within its body (for example, to bail out if an error is detected), use the **RETURN** primitive. You must not use the ATLAST **EXIT** primitive; it does not perform essential cleanup required when leaving a method. For example, a method might respond to the user canceling a variable

editing dialogue box as follows.

```
THIS OBJECT.INSPECT
0= IF
    RETURN    ( User hit Cancel )
THEN
    Method code continues...
```

Method arguments

Method arguments are declared in an argument list surrounded by the delimiters “(” and “)”. To enable CLASSWAR to obtain arguments from the AutoCAD user when a command method is invoked, arguments must be declared in a precisely specified way. Methods may have as many arguments as desired.

Each argument declaration begins with the data type of the argument, either one of the simple data types previously described for use in variable declarations (INTEGER, REAL, POINT, etc.), or one of the following argument-only data types.

ANGLE A 64 bit floating point number representing a relative angle in radians.

ORIENTATION A 64 bit floating point number representing an absolute bearing angle in radians.

CORNER A triple of 64 bit floating point numbers representing the corner of a box in space.

KEYWORD A string containing the keyword entered by the user, chosen from a keyword list specified for the argument.

Next in the argument declaration is the prompt to be issued to the AutoCAD user when the argument is requested. The prompt is specified as a constant string which should consist just of the description of the datum requested. The default (if any), and the balance of the prompt is provided by CLASSWAR.

Following the prompt and preceding the word **ARG**, one or more options can be specified. We’ll discuss the options in more detail in a moment. First, consider the following simple class definition, which we’ll refer to as “SPROINK.”

```
PUBLIC:
    DISTANCE size
    INTEGER nsides
```

```
COMMAND METHOD modpoly ((
    DISTANCE "New size" ARG
    INTEGER "New number of sides" ARG ))
{
    nsides !
    size 2!
}
```

When this method is run as an AutoCAD command, CLASSWAR conducts the following dialogue with the user:

```
Command:  modpoly
Select objects: 1 selected, 1 found
Select objects:
New size:  3.2
New number of sides:  3
```

Command:

The arguments obtained are passed to the method’s implementation code on the stack, in the order the arguments were declared (the last argument in the list will be on the top of the stack). Integer and floating point arguments are placed directly on the stack; points and other co-ordinate triples and string arguments are passed as pointers to a temporary storage area containing the argument value.

When the method is invoked in CLASSWAR code rather than as an AutoCAD command, the argument declarations are ignored and arguments are obtained directly from the stack. This is how CLASSWAR allows the same methods to serve as programming language constructs and AutoCAD commands.

For example, here’s a class definition that includes an object of class SPROINK and uses its *modpoly* method from within one of its own methods.

```
PUBLIC:
    INTEGER icount
    SPROINK fiddle

COMMAND METHOD clearit
{
    0 icount !
    3.2 3 fiddle modpoly
}
```

In the *clearit* method we set our *icount* variable to zero, then use the *modpoly* method to set the *size* and

`nsides` variables of the `SPROINK` object named `fiddle` to the values we passed on the stack: 3.2 and 3.

Since the `size` and `nsides` variables were declared `PUBLIC`: in the definition of the `SPROINK` class, we could have set them directly in the `clearit` method, as follows:

```
COMMAND METHOD clearit
{
    0 icount !
    3.2 fiddle size 2!
    3 fiddle nsides !
}
```

When a public variable name is used as a method, it places the address of the selected instance or class variable on the stack, allowing access to it with normal ATLAST facilities.

As noted above, several options can be specified to control argument acquisition. The following paragraphs describe these options.

Default values. You can specify a default value for an argument by supplying a pointer to a temporary variable containing the default and the `DEFAULT` keyword. For example, if we'd declared the `MODPOLY` method of `SPROINK` as follows:

```
TEMPORARY:
    DISTANCE dsize
    INTEGER dnsides

1.0 dsize 2!
3 dnsides !

PUBLIC:

COMMAND METHOD modpoly ((
    DISTANCE "New size"
        dsize DEFAULT
    ARG
    INTEGER "New number of sides"
        dnsides DEFAULT
    ARG ))
{
    nsides !
    size 2!
}
```

the dialogue with the AutoCAD user would have been:

```
Command: modpoly
Select objects: 1 selected, 1 found
Select objects:
```

```
New size <1.0000>: 3.2
New number of sides <3>: 3
```

```
Command:
```

Had the user entered a blank line to either of these prompts, the specified default value would have been passed to the method. Defaults are specified as pointers to variables containing the default value to allow methods to change the default values where appropriate. For example, many AutoCAD commands use the last values entered as the defaults for the next invocation of the command. A `CLASSWAR` command method can implement this simply by storing its arguments into the corresponding default variables. Note that defaults must be stored in temporary variables; you cannot use the contents of an instance or class variable as a default. Since AutoCAD obtains all arguments needed to process objects in a selection set in advance, the specific variable settings of the individual objects are unavailable when arguments are obtained.

Acquisition modes. AutoCAD input processing provides a rich set of input validation checks. All of these can be used when obtaining `CLASSWAR` arguments. These modes are requested by computing the sum of the requested modes, then enabling them with the `ARGMODES` keyword. The available modes are:

<code>ARG_nonull</code>	Null input prohibited.
<code>ARG_nozero</code>	Zero input prohibited.
<code>ARG_noneg</code>	Negative input prohibited.
<code>ARG_nolim</code>	Don't check drawing limits.
<code>ARG_dashed</code>	Use dashed rubber band line.
<code>ARG_cronly</code>	End string input with Return only.

Here is a version of the `MODPOLY` method that rejects negative size specifications and zero or negative inputs for the number of sides.

```
COMMAND METHOD modpoly ((
    DISTANCE "New size"
        dsize DEFAULT
    ARG_noneg ARGMODES
    ARG
    INTEGER "New number of sides"
        dnsides DEFAULT
    ARG_noneg ARG_nozero + ARGMODES
```

```

        ARG ))
{
    nsides !
    size 2!
}

```

Now AutoCAD will enforce the required modes for the arguments, as illustrated below.

```

Command: modpoly
Select objects: 1 selected, 1 found
Select objects:

```

```

New size <1.0000>: --23
Value must be positive.
New size <1.0000>: 23
New number of sides <3>: 0
Value must be positive and nonzero.
New number of sides <3>: --11
Value must be positive and nonzero.
New number of sides <3>: 6
Command:

```

Base points. ANGLE, CORNER, DIRECTION, DISPLACEMENT, DISTANCE, ORIENTATION, POINT, and TRIPLE arguments may be specified either by typing the value on the keyboard or by graphically entering it with the pointing device. Graphical specifications are often relative to a base point known at argument acquisition time. For example, a distance argument used as the radius of a circle can be specified as a vector from the centre of the circle to a point on its circumference. If no base point is specified, the user would have to enter two points to define the distance; supplying the base point in the argument definition not only relieves the user of having to enter the first point, it permits AutoCAD to draw a rubber band line that dynamically shows the value being specified.

We can modify the acquisition of the polygon size in MODPOLY to use a base point of (0,0,0) as follows. This allows the user to draw the new size on the screen as a vector from the origin.

```

TEMPORARY:
    POINT bpoint
    0.0 0.0 0.0 bpoint POINT!

```

PUBLIC:

```

COMMAND METHOD modpoly ((
    DISTANCE "New size"
    dsize DEFAULT

```

```

    bpoint BASEPOINT
    ARG_noneg ARGMODES
    ARG
    INTEGER "New number of sides"
    dnsides DEFAULT
    ARG_noneg ARG_nozero + ARGMODES
    ARG ))
{
    nsides !
    size 2!
}

```

Keyword lists. Finally, you can supply a list of keywords that can be entered, either as the sole form of input (argument type KEYWORD), or as alternatives for one of the numeric or co-ordinate argument types. (You cannot request keywords as options on string input; any text may be entered as a string argument so keywords cannot be recognised there.)

Keyword lists are specified as strings, usually constant, naming the keywords recognised by the command. You can accept abbreviated forms of keywords by capitalising the letters of the minimum prefix recognised as the keyword or, alternatively, by suffixing the abbreviation to the keyword after a comma. When a keyword is declared for an argument, CLASSWAR passes the method either the numeric value followed by a zero integer or, if a keyword was entered, the full keyword given in the keyword string (even if the user entered an abbreviation) followed on the stack by a nonzero integer. The integer flag permits the method to determine whether the keyword was entered and take appropriate action.

Suppose, for example, we'd like to further extend our much hacked MODPOLY method to accept "Triangle" and "Square" as well as numeric input for the number of sides argument. To accomplish this, we can modify the declaration as follows.

```

COMMAND METHOD modpoly ((
    DISTANCE "New size"
    dsize DEFAULT
    ARG_noneg ARGMODES
    ARG
    INTEGER "New number of sides"
    dnsides DEFAULT
    ARG_noneg ARG_nozero + ARGMODES
    "Triangle Square" KEYWORDS
    ARG ))
{
    if
        "Triangle" strcmp 0= if

```

```

        3
    else
        4
    then
then
nsides !
size 2!
}

```

The first IF statement in the method tests the status word indicating whether a keyword was entered for the second argument. If so, it tests which one (since there are only two keywords, if it isn't "Triangle" it must be "Square." If the status is zero, the user entered a number which we use as before. We can use this method as follows:

```

Command: modpoly
Select objects: 1 selected, 1 found
Select objects:

New size <1.0000>: 2
New number of sides <3>: Squ
Command:

```

Since the "S" of "Square" was capitalised, we can abbreviate the keyword to as little as the initial letter.

Procedural arguments

The same syntax used to specify arguments within method argument lists can be used within the body of a method to obtain arguments from the AutoCAD user when the method runs. Although the syntax and semantics of the argument acquisition are identical, obtaining a *procedural argument* within a method is a very different operation, and it's important to understand the distinction.

When processing a collection of objects chosen with an AutoCAD selection set, CLASSWAR asks for the arguments requested by their methods, as declared in argument lists, only once. Identical argument requests, even if declared in argument lists of methods of entirely different classes, are merged into the minimum number of requests to the user. After the user enters all required arguments, the methods are applied to the objects in the selection set without user intervention.

When a procedural argument is requested within a method, the user is immediately asked for the argument. A procedural argument is never automatically provided on the stack—the user must always enter it. Procedural argu-

ments are consequently primarily useful in object acquisition methods where one is specifying the initial properties of a new object, or for handling error conditions that may arise while processing objects. We could replace the argument list of our MODPOLY method with procedural arguments as follows:

```

COMMAND METHOD modpoly
{
    DISTANCE "New size"
        dsize DEFAULT
        ARG_noneg ARGMODES
    ARG
    -1 = if return then
    INTEGER "New number of sides"
        dnsides DEFAULT
        ARG_noneg ARG_nozero + ARGMODES
        "Triangle Square" KEYWORDS
    ARG
    dup -1 = if return then
    1 = if
        "Triangle" strcmp 0= if
            3
        else
            4
        then
    then
    nsides !
    size 2!
}

```

We've added extra status checking because procedural argument acquisition always leaves a status word on the stack regardless of whether a keyword was specified or not. The status is 0 if a normal argument was entered, 1 if a keyword was entered, and -1 if the user canceled the input with Control C.

As long as we select a single entity as the object of the MODPOLY command, this method will behave as before. If, however, we select several objects of class SPROINK, the prompts for the new size and number of sides will appear for each individual object. This is rarely what's desired, so procedural arguments should be used only where appropriate.

Special purpose methods

The meaning of most methods is entirely up to the creator of the class, but CLASSWAR reserves the names of several methods for predefined purposes. If a class definition contains methods with these names, CLASSWAR

will automatically invoke it at the proper times and will assume it conforms to the guidelines given below.

DRAW method. A DRAW method, if present, is executed whenever an object is initially created or any of its instance variables change (regardless of how the change came about). The DRAW method is invoked with no arguments and is responsible for generating the geometrical representation of the object in the AutoCAD database. The DRAW method may create the geometry using the turtle, with SGLIB, or by assembling entities with the ADS bindings and adding them to the database with the ADS_ENTMAKE primitive. When the DRAW method receives control, the turtle has been RESET to its default parameters, then set to generate Polyline entities with “1 LEAVETRACKS.” The SGLIB current transformation is set to the identity transform upon entry to the DRAW method. The instance variables of the object being generated may be accessed simply by referring to their variable names. The object should be generated relative to the world coordinate system origin, (0,0,0); CLASSWAR automatically translates and rotates it to the correct orientation in space. If the DRAW method generates no geometry for the object, the user will not be able to pick it on the AutoCAD screen but the object will still be stored in the database.

ACQUIRE method. If an ACQUIRE method is defined within a class, an AutoCAD command with the same name as the class is defined to create new objects of that class. For example, if we’ve created a class named “SPIRAL,” entering SPIRAL at the AutoCAD command prompt will invoke the ACQUIRE method of that class. The task of ACQUIRE is simple: to initialise the instance variables for the new object. This is usually accomplished by conducting a dialogue with the user, often employing procedural argument requests, to obtain the values for each variable. The ACQUIRE method is expected to leave a status on the stack indicating whether it succeeded in obtaining the instance variable values. If it returns TRUE, the new object will be created. If FALSE is left on the stack (as it might do, for example, if the user canceled one of the input requests), no object is created.

An ACQUIRE method can present the user with a dialogue box for specifying the properties of the new object simply by storing whatever default values are desired in the instance variables and executing “THIS OBJECT.INSPECT.” This executes the OBJECT.INSPECT method (defined for all CLASSWAR objects as part of the superclass “object” inherited implicitly by all class definitions) of the current object (THIS), presenting an INSPECT dialogue and re-

turning TRUE if the user picks OK and FALSE if Cancel is selected. The MOUNTAIN.CLS class definition uses OBJECT.INSPECT in this manner.

NEW method. The NEW method is the *instance constructor* of the class. Whenever a new object of the class is created, the NEW method is run (if defined). Its job is usually to store default values into the instance variables appropriate to newly-created objects of the class. It may, however, do anything it wishes; it is a fully general method. Don’t confuse the NEW method with the ACQUIRE method. Whenever an object is created, whether by being declared as a component of another class or as an AutoCAD database object, the NEW method is run. It is non-interactive. The ACQUIRE method is invoked only when the class name is entered as an AutoCAD command, after running the NEW method (if any). An ACQUIRE method may rely on the NEW method to store default values for the instance variables, but its main job is to interactively query the user for the properties of this specific object. If no NEW method is defined, instance variables of newly created objects will be set to standard defaults: zero for numbers, the null string for strings.

NEWCLASS method. The NEWCLASS method is referred to as the *class constructor*. It is invoked just once; when the class is first defined within a drawing. The NEWCLASS method is responsible for setting class variables to their initial values. If no NEWCLASS method is defined, class variables will be set to standard defaults: zero for numbers, the null string for strings.

Derived classes: inheritance

You can create new classes by defining them from scratch, only using primitive CLASSWAR objects, by building upon existing classes by including them in the classes you define, or by *deriving* new classes from previously defined classes, allowing them to *inherit* the variables and methods of their *parent class*, adding whatever additional variables and classes are required and redefining any methods of the parent class that are unsuitable to the derived class.

To create a derived class, simply include the statement:

```
:parent [PUBLIC] DERIVED
```

as the first line of the derived class definition. The *parent* name specifies the name of the parent class. This name is preceded by a colon to denote the name of the parent class itself, as opposed to the declaring word used

to create instances of the class. If “PUBLIC” is specified before “DERIVED,” variables and methods declared **PROTECTED:** and **PUBLIC:** in the parent class will retain those attributes in the derived class. If **PUBLIC** derivation is not specified, all variables and methods inherited from the parent class are considered **PRIVATE:** in the derived class.

When a derived class is created, all **PROTECTED:** and **PUBLIC:** variables and methods of the parent class are included by reference in the derived class. They can be referenced in the same way as if declared within the derived class. If you declare a method that duplicates the name of a method of the parent class, it is *overloaded*; the method in the derived class supplants the parent class’ method.

Please examine the example in the introduction of a labeled polygon class defined by derivation from a simple polygon class for details on how a derived class is defined.

Constructors and derived classes

When classes are created by derivation from existing classes, variables in the parent class are incorporated into the derived class. The class and instance constructors (**NEWCLASS** and **NEW** methods) of the parent class are automatically invoked when new objects of the derived class are created. If a class is defined by several levels of derivation, the constructors are called with the topmost class constructor first, the constructor for the class derived from it second, and so on with the constructor for the bottom class last. Constructors in derived classes are free to override defaults set by parent class constructors as long as they have access to the variables they wish to initialise (in other words, the variables are **PUBLIC:** or **PROTECTED:**).

The virtualise operator: <-

When classes are built by derivation from existing classes or by incorporating instances of other classes, cases arise where names used in one class clash with the names of another. Variables of a class definition can be referenced within that definition just by giving their name; this compiles an efficient reference to the class, instance, or temporary variable. Similarly, methods defined within a class definition may be called by other methods of the same definition with the same speed as invoking any other ATLAST definition. There are cases where this action is inappropriate and CLASSWAR must use the general definition of the name as a method applicable to objects of a variety of classes. You can force the general interpretation of a

name by following it with the “<-” operator (pronounced “send to”).

This operator was used in the example of a derived class given at the start of this section. The **DRAW** method of the derived class wanted to use the **DRAW** method of its parent class to draw the outline of the polygon. Had the method simply performed:

```
THIS draw
```

which pushes the current object on the stack and calls the **DRAW** method, an infinite recursive call on the **DRAW** method of the derived class would have ensued. Instead, it specified:

```
THIS draw <-
```

Following “draw” with “<-” forces CLASSWAR to use the parent class **DRAW** method, accomplishing the desired result.

You can also use the “<-” operator to access variables of classes that duplicate the names of variables defined in the current class. You normally reference variables in the current class just by giving their name. This is convenient and efficient, but causes a problem if you want to access a variable with the same name from a component class. In the following class definition, including an instance of our **SPROINK** class:

```
PUBLIC:
    REAL size
    SPROINK fiddle

COMMAND METHOD clearit
{
    1.0 size 2!
    3.2 fiddle size <- 2!
    3 fiddle nsides !
}
```

we have a potential problem because the **size** variable in our class duplicates the name of the **size** variable in **SPROINK**. By following the reference to “size” with the “<-” operator, we inform CLASSWAR to use the most general, or *virtual* definition of **size**—a definition that examines the object on the top of the stack and returns the correct variable from it depending on which class it belongs to.

Drawing Facilities

Three separate drawing facilities are provided for **DRAW** methods to add geometry to the AutoCAD database. These facilities can be used separately or in conjunction with one another. The facilities consist, from the lowest to the highest level, of ATLAST bindings for ADS functions to assemble result buffer lists and create entities with `ads_entmake()`, a comprehensive ATLAST binding for SGLIB, and a complete turtle geometry package that can generate a variety of AutoCAD objects.

The ADS binding

The ADS binding primitives provide facilities that allow you to assemble result buffer lists representing AutoCAD entities and add them to the database. Key to understanding the ADS primitives is the *current item*. The ADS primitives are always working on one entity. This current item is implicitly referenced by all of the ADS binding primitives.

Item primitives

The item primitives operate upon entire items (lists of groups representing entire entities in the AutoCAD database).

ADS_ENTMAKE. The entity described by the current item is added to the AutoCAD database. The status from the operation is left on the stack; it will be positive if the object was added successfully, negative in case of error.

ADS_ENTMOD. The entity described by the current result item is modified in the AutoCAD database to reflect the values given in the result item. The ADS status is left on the stack: positive if the object was successfully modified, negative otherwise.

CLEARITEM. All groups of the current item are deleted. You'd only use this if you intended to build a new item from scratch using **ADDGROUP**. The stack is not affected.

PRINTITEM. All groups of the current item are printed on the AutoCAD text screen. For example, if an Arc is the current entity, the statement:

PRINTITEM

might generate the following output:

```
0:  "ARC"
8:  "0"
10: (3, 2, 0)
40: 1
50: 0
51: 90
```

Group primitives

The group primitives provide access to the individual data fields that make up an item. In the following descriptions of primitives, assume that the current item is a Line entity on layer 0, from co-ordinates (1,1,0) to (2,2,0). This item would be displayed by “**PRINTITEM**” as:

```
0:  "LINE"
8:  "0"
10: (1, 1, 0)
11: (2, 2, 0)
```

Groups within an item can be identified either by group code or by their position within the item. Regular AutoCAD item fields are always unique and may be identified simply by their group codes. Extended entity data, however, uses the same group code for all fields of a given type, so group codes are not necessarily unique. A positive number used to designate a group chooses the first occurrence of that group code in the current item. A negative number of the form $-(10000 + n)$, where n specifies the position of the group within the item (with the first group numbered zero), selects the n th group in the chain of groups composing the item and may be used to uniquely specify extended entity groups that appear more than once in an item.

PRINTGROUP. The group identified by the second item on the stack is printed on the AutoCAD text screen. For example:

```
Command: atlast
Atlast[0]-> 10 printgroup
10:  (1, 1, 0)
Atlast[0]-> -10001 printgroup
8:  "0"
Atlast[0]->
Command:
```

GROUPCOUNT. Places the number of groups in the current item on the top of the stack.

```
Atlast[0]-> groupcount .
4
```

GROUP?. If the group with group code given by the top of the stack is present in the item, -1 is placed on the top of the stack. If the group does not appear in the item, 0 is returned.

```
Atlast[0]-> 10 group? .
-1
Atlast[0]-> 40 group? .
0
```

DELGROUP. The group on the top of the stack is deleted from the item, if present. If the specified group is not present, DELGROUP is simply ignored.

```
Atlast[0]-> printitem
0: "LINE"
8: "0"
10: (1, 1, 0)
11: (2, 2, 0)
Atlast[0]-> 8 delgroup
Atlast[0]-> printitem
0: "LINE"
10: (1, 1, 0)
11: (2, 2, 0)
```

GROUP. The value of the specified group, in whatever form is appropriate for it, is placed on the top of the stack. Integers are stored as single stack items; real numbers and angles as pairs of stack items representing their floating point values; co-ordinates as triples of pairs, each giving a floating co-ordinate with Z at the top of the stack, Y next, and then X ; strings as the address of a temporary string buffer containing the text; and binary chunks as a length, in bytes, on the top of the stack and the address of the chunk data, stored in a temporary string buffer, next on the stack.

```
Atlast[0]-> 10 group f. f. f.
0 1 1
```

ADDGROUP. A group with the type given by the top of the stack is added to the end of the item. The value

field of the group is cleared to zero, and may be then set with SETGROUP.

```
Atlast[0]-> 62 addgroup
Atlast[0]-> printitem
0: "LINE"
10: (1, 1, 0)
11: (2, 2, 0)
62: 0
```

SETGROUP. Sets the value of the group specified by the top of the stack to the values below it (in the same form as the results returned by GROUP). Removes the group specification and the values from the stack.

```
Atlast[0]-> 3 62 setgroup
Atlast[0]-> 3.0 4.0 5.0 10 setgroup
Atlast[0]-> printitem
0: "LINE"
10: (3, 4, 5)
11: (2, 2, 0)
62: 3
```

The SGLIB binding

The SGLIB-based drawing facilities are implemented as a collection of ATLAST primitives that provide access to the linear algebra, geometric, and object creation facilities of SGLIB, the Simple Graphics Library.

Co-ordinate systems

All geometric objects generated by the library are multiplied by a 4×4 transformation matrix before being output. This matrix is initialised to the identity matrix, defining the world co-ordinate system. At any point there is a current co-ordinate system in effect and all geometry is transformed by it. Co-ordinate systems may be pushed and popped on a co-ordinate system stack, limited only by the amount of memory available to save them.

When a transformation is specified, it is composed with the current transformation by premultiplying it with the current transformation. If C is the current transformation matrix, then the new matrix is given by $C \leftarrow T_{new}C$. This concatenates the transform into the viewing pipeline chain. If you wish to restore the earlier transform, push it before concatenating the new transformation, then pop the transformation stack when you want it back.

Drawing functions

Several functions may be used to draw objects. All co-ordinates passed to these functions are passed through the current co-ordinate system transform before being emitted to AutoCAD. Typical models do not use these functions very frequently—most models are built from the graphics primitives described in the next section, but the drawing functions are the most fundamental operations in the library, and are therefore important to understand.

Draw vector

p1 p2 DRAWVEC

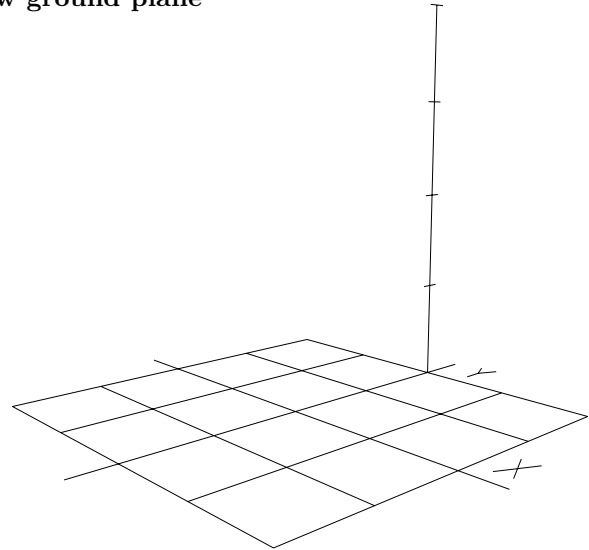
Draws a vector from *p1* to *p2*, both POINTs. The line is created with the colour given by the variable `OBJECT.DRAWCOL`. The default colour is “by block,” which causes geometry created to take on the overall colour of the object that contains it.

Draw polygonal face

n p1 p2 p3 p4 visbit DRAWFACE

Draws an AutoCAD `3Dface` object with *n* sides (which must be either 3 or 4) bounded by the four vertices, which must be specified in the order one would encounter them walking around the face. If *n* is 3, the *p4* argument must still be specified, but may simply be a duplicate of *p3*. This primitive generates a `3Dpoly` entity if `OBJECT.WIREFRAME` has been set to `TRUE`. The *visbit* argument specifies visibility of the edges of the generated face. If the 2^i bit in *visbit* is nonzero, the edge beginning with vertex p_{i+1} will be invisible. All objects are created with the AutoCAD colour currently set in the variable `OBJECT.DRAWCOL`, which defaults to “by block.”

Draw ground plane



To draw a ground plane (co-ordinate system reference), use:

fancy GPLANE

If *fancy* is zero a ground plane of lines is drawn. If *fancy* is nonzero you’ll get a snazzy ground plane made up of faces which will appear in shaded images. The regular ground plane is very handy, as it provides a co-ordinate system reference within AutoCAD, then disappears from the shaded image when you render the model.

Draw cube

CUBE

Draws a cube of edge length 2, centred around the origin. Why 2? Because that makes the vertex co-ordinates $(\pm 1, \pm 1, \pm 1)$, which lends itself nicely to clean examples. Note that one uses the transformations to move, scale, and rotate this primitive cube as needed to get the figure you want in the model. This is how one works with this package, and it’s a very expressive way to build models when you get used to it.

Draw sphere

SPHERE

Draws a unit radius sphere centred around the origin. The sphere is normally drawn as a mesh of 64 faces, but if the variable `OBJECT.WIREFRAME` is set nonzero, the sphere

is represented as three mutually orthogonal circles. Note that the wireframe presentation of a sphere will disappear in shaded renderings. You can set the number of faces in the mesh representation of the sphere by changing the variable `OBJECT.CNSEGS`. The default value of 8 specifies 8 longitudinal and 8 latitudinal tabulations, or a total of 64 faces.

Draw tetrahedron

TETRAHEDRON

A tetrahedron with edge size 1 is drawn at the origin of the current co-ordinate system. This operation modifies the point database.

Draw octahedron

OCTAHEDRON

An octahedron with edge size 1 is drawn at the origin of the current co-ordinate system. This operation modifies the point database.

Draw dodecahedron

DODECAHEDRON

A dodecahedron with edge size 1 is drawn at the origin of the current co-ordinate system. This operation modifies the point database.

Draw icosahedron

ICOSAHEDRON

An icosahedron with edge size 1 is drawn at the origin of the current co-ordinate system. This operation modifies the point database.

Draw polygon

A polygon is defined by storing the vertices in a “point database,” and then enumerating the vertex indices for one or more polygons. This form of specification, used in many of the classics of computer graphics such as the teapot and the Volkswagen, is compact, fast, and easy to use. Since one tends to use points over and over again in

describing a complex surface as polygons, this representation tends to reduce the bulk of such definitions. Geometry defined this way may be output as AutoCAD rat nest mesh entities, minimising the database space required to store the object.

Points are placed in the point database with:

`x y z n PNT`

where n is the point number, starting with 1, and x , y , and z are the floating point co-ordinates of the point. Points need not be specified in sequential order, nor need they form a contiguous sequence of point numbers. They remain in the point database until a subsequent point is stored with the same point index. Since the point database is an array, very large point numbers should be avoided. The point database array is automatically acquired on the first reference, and is expanded automatically as needed in a tasteful and efficient manner.

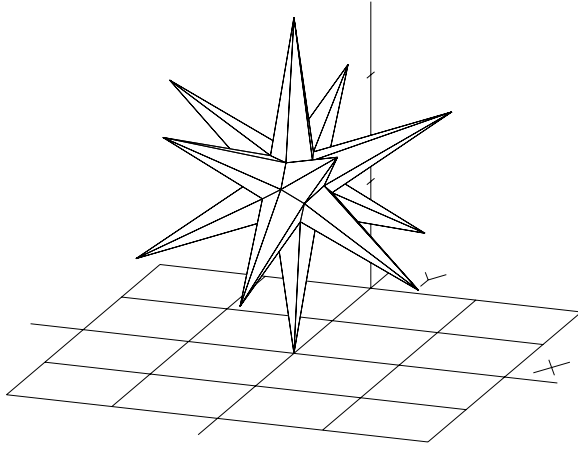
Once points have been stored in the point database, you may draw polygons which use them as vertices with calls on:

`0 v1 v2 ... vn POLY`

This primitive accepts a variable number of arguments, with the list terminated by the initial zero (this is why point numbers must start with 1). The $v1$, $v2$, through v_n arguments are point numbers which reference the points previously stored in the point database. Using points in a call on `POLY` leaves them unchanged in the point database so they may be used to define any number of polygons.

Polygons may have “invisible edges.” If a vertex index is negative then the edge that begins with the point in the point database with index equal to the absolute value of the specified index will not appear in the drawing. If the last index is negative, the segment that closes the polygon by connecting the last vertex to the first will be invisible.

Normally, each `POLY` reference generates an individual AutoCAD 3DFace entity. You can collect all the point and polygon references into a single AutoCAD rat nest mesh by invoking the `RATON` primitive before the first `PNT` of the object, and the `RATOFF` primitive after its last `POLY`.



If the floating point variable `OBJECT.STELLATION` is set nonzero, polygons are generated with “stellated faces.” These faces are defined by taking the arithmetic mean of all of the co-ordinates of the vertices, then measuring a distance equal to the value of `OBJECT.STELLATION` multiplied by the length of the first edge of the polygon in a direction along a vector normal to the plane defined by the vertices bounding the first two edges of the polygon (if `OBJECT.STELLATION` is negative, the distance is measured in the direction opposite the normal), then drawing triangles whose bases are the edges of the polygon and whose third vertices are the point arrived at by the process just described.

Setting `OBJECT.STELLATION` nonzero is primarily useful in conjunction with regular polyhedra. It may, however, be used when any polygon is generated as a “special effect”. Be sure to remember to reset `OBJECT.STELLATION` to zero after generating a stellated object to avoid unanticipated results in subsequent calls on `poly()`.

Transformation functions

These functions compose transformations with the current transformation matrix and provide facilities to save and restore transformation matrices.

Translation

`tz ty tx XTRANS`

Composes a translation by the specified displacements.

This is accomplished by:

$$C \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{bmatrix} C$$

Scaling

`sx sy sz XSCAL`

Composes a scale factor transformation with the current transformation. Note that the scale factor may be nonuniform (this permits transforming the `CUBE` primitive into an arbitrary box, and the `SPHERE` into a general ellipsoid, and permits mirroring by specification of negative scale factors). Thus:

$$C \leftarrow \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} C$$

Rotation

You may rotate about the *X*, *Y*, or *Z* axes with the call:

theta axis `XROT`

where *theta* is the angle to rotate, in radians, and *axis* is the axis to rotate about: 0 for the *X* axis, 1 for the *Y* axis, 2 for the *Z* axis. Specification of a positive direction generates clockwise rotation when viewed in the direction of the designated co-ordinate axis.

This composes one of three matrices with the current transformation. If we define:

$$\begin{aligned} s &= \sin \theta \\ c &= \cos \theta \end{aligned}$$

then, if *axis* is 0 (*X*),

$$C \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} C$$

If *axis* is 1 (*Y*),

$$C \leftarrow \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} C$$

and if *axis* is 2 (*Z*),

$$C \leftarrow \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} C$$

Perspective

You can introduce perspective distortion with the call:

alpha zn zf **XPERS**

Where *alpha* specifies the field of view in radians (to simulate the eye, use **26.0 180.0 f/ PI f***), *zn* specifies the distance from the eye to the near clipping plane (i.e. the projection plane), and *zf* specifies the distance from the eye to the back clipping plane. Note that the clipping plane specifications are used only to calculate the projection; no clipping is done, and if objects outside this volume are projected, strange output will occur. The transformation assumes that the eye is at the origin, looking down the positive *Z* axis—to achieve this, you should compose the required rotations and translations after calling **PERS** to establish the perspective transformation.

The perspective transformation is composed by defining:

$$\begin{aligned} s &= \sin \alpha/2 \\ c &= \cos \alpha/2 \\ q &= \frac{s}{1 - zn/zf} \end{aligned}$$

Then,

$$C \leftarrow \begin{bmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & q & s \\ 0 & 0 & -q \times zn & 0 \end{bmatrix} C$$

Note: the perspective transformation is included for completeness and because it belongs in any transformation library. In building models for use with AutoCAD and AutoShade, you will rarely need it, as you normally build a model in world co-ordinates and use the viewing facilities of AutoCAD and AutoShade to examine the model. Once a model has been processed by a perspective transformation, it is distorted to simulate viewing, but it cannot be correctly processed by code that counts on three dimensional spatial relationships, such as AutoCAD's hidden line code or AutoShade's obscuration tests. Therefore, you can use the perspective transformation to make models that correctly represent perspective when viewed in plan view by AutoCAD, but if you try to **HIDE** them, you'll get grossly incorrect results.

Arbitrary orientation

You can compose an arbitrary orientation matrix with the current transformation with the call:

a b c d e f p q r **XORIE**

This function is most often used to specify an arbitrary rotation, but can be used to specify skew transformations, as there's no checking of the values supplied.

$$C \leftarrow \begin{bmatrix} a & d & p & 0 \\ b & e & q & 0 \\ c & f & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} C$$

Saving a transformation

XPUSH

Pushes the current transformation on the transformation stack.

Restoring a transformation

XPOP

Restores the most recently pushed transformation from the transformation stack. The current transformation in effect at the time of the **XPOP** is lost.

Reset transformation

XRESET

Resets the current transformation to the identity transformation and discards all transformations saved on the transformation stack.

Beginning a new transformation

When building three dimensional models, many transformations are built, used for a subassembly, then discarded. To make code that generates such models more readable, the primitive:

XTHEN

is provided. This is identical in effect to the sequence:

XPOP X PUSH

and allows one to define a new co-ordinate system based on the previously active co-ordinate system.

Vector operations

These functions operate on 4-element row vectors, which are usually interpreted as points in homogeneous co-ordinates.

Declare vector

A 4 element vector is declared with the statement:

4VECTOR *name*

4VECTORS are defined only as temporary objects; they cannot be saved as instance or class variables.

Get vector

x y z v VECGET

Creates a homogeneous co-ordinate vector

$\begin{bmatrix} x & y & z & 1 \end{bmatrix}$
and stores it into 4VECTOR *v*.

Put vector

v VECPUT $\rightarrow x y z$

Stores rescaled co-ordinates from a homogeneous vector *v* on the stack as *x*, *y*, and *z*. If *v* is $\begin{bmatrix} X_v & Y_v & Z_v & W_v \end{bmatrix}$ then:

$$\begin{aligned} x &\leftarrow X_v/W_v \\ y &\leftarrow Y_v/W_v \\ z &\leftarrow Z_v/W_v \end{aligned}$$

Copy vector

v vo VECCOPY

Copies 4VECTOR *v* to *vo*.

Transform vector

v m vo VEXCMAT

The 4VECTOR *v* is transformed by multiplying it by the matrix *m*, and the resulting 4VECTOR is stored into *vo*.

$$vo \leftarrow vm$$

Print vector

v VECPRINT

Prints the 4VECTOR *v* on the AutoCAD text screen.

Vector algebra

These functions operate on 3-element row vectors, stored in the data type POINT. You can pass data of type 4VECTOR to these functions, but only the first three elements will be processed.

Get point

x y z p POINTGET

Sets the co-ordinates of point *p* to (*x*, *y*, *z*).

Copy point

p po POINTCOPY

Copies point *p* to point *po*.

Dot (inner) product

a b VECDOT $\rightarrow p$

Computes the dot product of the 3-vectors *a* and *b*, and returns the result as a REAL *p*.

$$p \leftarrow a \cdot b$$

Cross (vector) product

a b o VECCROSS

The vector product of the 3-vectors a and b is computed and stored into o . The result vector may be the same as either of the input vectors.

$$o \leftarrow a \times b$$

Add vectors

$a \ b \ o \ \text{VECADD}$

Vectors a and b are added and the sum is stored in vector o . The result vector may be the same as one of the vectors added.

$$o \leftarrow a + b$$

Subtract vectors

$a \ b \ o \ \text{VECSUB}$

Vector b is subtracted from vector a and the difference vector is stored into o . The result vector may be the same as one of the vectors subtracted.

$$o \leftarrow a - b$$

Scalar product

$a \ s \ o \ \text{VECSCAL}$

Vector a is multiplied by the scalar value s and the result is stored in vector o . Vectors a and o may be the same vector.

$$o \leftarrow as$$

Magnitude of vector

$a \ \text{VECMAG} \rightarrow m$

The magnitude (absolute length) of vector a is returned on the stack as a floating point value m .

$$m \leftarrow |a|$$

Normalise vector

$a \ o \ \text{VECNORM}$

The vector a is normalised by scaling it so that its magnitude is 1. The resulting vector is stored in o . Vectors a

and o may be the same vector.

$$o \leftarrow a/|a|$$

General matrix operations

These functions implement operations on 4×4 matrices.

Declare matrix

A 4×4 matrix is declared with the statement:

MATRIX *name*

MATRIX variables are defined only as temporary objects; they cannot be saved as instance or class variables.

Multiply matrices

$a \ b \ o \ \text{MATMUL}$

Matrix a is multiplied by matrix b and the result is stored in matrix o .

$$o \leftarrow ab$$

Identity matrix

$m \ \text{MATIDENT}$

Sets m to the identity matrix:

$$m \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Copy matrix

$a \ o \ \text{MATCOPY}$

Copies matrix a to matrix o .

Print matrix

$m \ \text{MATPRINT}$

Prints the matrix on the AutoCAD text screen.

Transformation matrix functions

These functions construct matrices which perform the various geometric transformations. These functions generate the primitive matrices which the transformation functions compose with the current co-ordinate transformation, but simply store the matrix for the transformation into a matrix given on the stack. The transformation parameters are identical to those of the corresponding transformation function, which description you should examine for additional information.

Translation matrix

tx ty tz m MATTRAN

$$m \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

Scaling matrix

sx sy sz m MATSCAL

$$m \leftarrow \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation matrix

You may create a matrix to rotate about the *X*, *Y*, or *Z* axes with the call:

theta axis m MATROT

where *theta* is the angle to rotate, in radians and *axis* is the axis to rotate with 0 denoting the *X* axis, 1 the *Y* axis, and 2 the *Z* axis.

$$\begin{aligned} s &= \sin theta \\ c &= \cos theta \end{aligned}$$

If *axis* is *X*:

$$m \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If *axis* is *Y*:

$$m \leftarrow \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If *axis* is *Z*:

$$m \leftarrow \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective matrix

alpha zn zf m MATPERS

Where *alpha* specifies the field of view in radians (to simulate the eye, use **26.0 180.0 f/ PI f***), *zn* specifies the distance from the eye to the near clipping plane (i.e. the projection plane), and *zf* specifies the distance from the eye to the back clipping plane.

$$\begin{aligned} s &= \sin alpha/2 \\ c &= \cos alpha/2 \\ q &= \frac{s}{1 - zn/zf} \end{aligned}$$

Then,

$$m \leftarrow \begin{bmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & q & s \\ 0 & 0 & -q \times zn & 0 \end{bmatrix}$$

Arbitrary orientation

a b c d e f p q r MATORIE

$$m \leftarrow \begin{bmatrix} a & d & p & 0 \\ b & e & q & 0 \\ c & f & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

CLASSWAR Software Drawing Turtles

The turtle graphics package implements a superset of the Turtle Procedure Notation given in the book “*Turtle Geometry*.” The extensions allow turtles to move in three-dimensional space and to generate both linear paths and closed planar polygons with edges traced out by the turtle’s motion. The turtle graphics package can be used either interactively or within compiled class definitions. When used interactively (after entering the ATLAST interpreter with the ATLAST command), a turtle icon is drawn on the screen at the active turtle’s present position (unless suppressed with HIDETURTLE).

Moving the turtle

To move the turtle forward, use:

distance **FORWARD**

where *distance* is the floating point distance you want the turtle to travel. Note that the distance must be specified as a floating point number. A common error is specifying an integer constant; this will result in a stack underflow when **FORWARD** attempts to remove a two item floating point value from the stack. The *distance* can be positive or negative; if negative the turtle backs up.

You can also make the turtle back up with:

distance **BACK**

This is precisely the same as **FORWARD** of $-distance$.

You can make the turtle “jump” to an absolute position with:

x y z **SETPOSITION**

where *x*, *y*, and *z* are the floating point co-ordinates to which the turtle should move.

The pen

As the turtle moves, it can “leave tracks” in a variety of forms. The turtle has a pen which, if lowered, traces out the path taken by the turtle. When the turtle icon is visible, the pen is drawn as a short vector normal to the plane of the turtle at its current position. (The pen will appear as a dot when the turtle is seen in plan view.)

The pen is initially down, so turtle movement draws lines.

To raise the pen, use:

PENUP

To lower it:

PENDOWN

Turning the turtle

You can change the turtle’s pointing direction (orientation) in a variety of ways. The turtle is a general three dimensional object and can assume any orientation in space. Because navigation in the plane is much simpler than arbitrary three dimensional positioning, distinct pointing primitives are provided which are suited to both dimensionalities.

To turn the turtle to the left (counterclockwise), use:

degrees **LEFT**

where *degrees* is the floating point angle, in degrees, you wish the turtle to turn. Note that the angle is always specified in degrees, unlike other angles in ATLAST (angles in degrees are used by tradition in turtle geometry; requiring conversion to radians would make existing algorithms much harder to convert and more difficult to read). The angle must be specified as a floating point number, with a trailing “.0” even if it’s an integral number of degrees. Specifying an integer will result in a stack underflow.

You can turn the turtle to the right (clockwise) either by using a negative angle with **LEFT** or, equivalently:

degrees **RIGHT**

The absolute orientation (bearing) of the turtle in the plane can be set with:

degrees **SETHEADING**

where 0 *degrees* denotes the positive *X* axis and angles increase counterclockwise (hence 90 would point the turtle along the positive *Y* axis). **SETHEADING** is meaningless if the turtle is not parallel to the *X–Y* plane.

The turtle can be moved out of the *X–Y* plane with the **PITCH**, **ROLL**, and **YAW** primitives. To make the turtle pitch up (rotate around its local *X* axis) use:

degrees **PITCH**

To roll the turtle about its local *Y* axis,

degrees ROL

(This primitive is named “ROL” to avoid conflict with the ATLAST stack manipulation primitive “ROLL.”) Finally, you can rotate the turtle within its current plane, turning about its local *Z* axis with:

degrees YAW

This is a synonym for **LEFT**, provided for consistency. The *degrees* specified in any of these rotation primitives may be negative in which case the direction of rotation is reversed.

Turtle tracks

When the turtle moves with the pen down, “tracks” are left on the drawing screen. The nature of these tracks is set by:

type LEAVETRACKS

where *type* is an integer from 0 to 3. If 0 (the default), ephemeral vectors are drawn using **ads_grdraw()** which will disappear the next time the screen is redrawn. If 1, a three dimensional Polyline is generated, with a new Polyline automatically begun whenever raising and lowering the pen creates a break in the turtle’s path. If 2, planar faces (expressed as AutoCAD rat nest mesh entities) are generated with the edges traced out by the turtle. If 3, individual Line entities are generated for each motion made by the turtle.

For any setting of **LEAVETRACKS**, the colour of the objects drawn may be set by storing the desired AutoCAD colour number in the variable **OBJECT.DRAWCOL**. The default colour is “by block,” which causes objects generated within **DRAW** methods to take on the overall colour of the object created and change if its colour is subsequently modified. If Polyline or Line entities are being generated, their line type can be set by storing the name of a pre-loaded AutoCAD line type into the string variable **OBJECT.DRAWLTYPE**. The default line type is “BYBLOCK.”

Disappearing turtles

When you’re at the interactive ATLAST command prompt, the turtle is shown on the screen at its present position and orientation as a triangular icon with a normal vector representing the pen extending upward from the current location. To make this icon disappear, use:

HIDETURTLE

To restore it:

SHOWTURTLE

You can adjust the size of the turtle icon by changing the settings of the floating point variables **TURTLE.HEIGHT**, **TURTLE.WIDTH**, and **TURTLE.DEPTH** which control, respectively, the height of the triangle (default 0.5), the width of its base (default 0.4), and the length of the normal vector (default 0.25).

Multiple turtles

Most turtle programs use only the single predefined turtle (like all turtles, it has a name, “KELVIN”), but you can declare additional turtles if you have need of them, or simply grow fond of the little critters. To declare a turtle, use:

TURTLE *name*

where *name* is the name of the new turtle. When a turtle is declared, its position is set to (0,0,0) in the *X–Y* plane, and its initial direction is along the positive *Y* axis. Declaring a turtle does not *activate* it. At any moment only one turtle is active; the active turtle responds to turtle commands such as **FORWARD**, **LEFT**, etc. To activate a turtle, use:

name **LISTEN**:

where *name* is the name you gave the turtle in its declaration. Turtles retain the entire state of the turtle graphics system, so any operations you do with the active turtle will not affect other turtles. You can place the active turtle on the stack with:

ME

You could use this, for example, to interrupt a drawing process with a turtle, draw a unit square at the origin, and resume where the original turtle left off with:

TURTLE delbert

ME

delbert **LISTEN**:

4 0 DO 1.0 FORWARD 90.0 RIGHT LOOP

LISTEN:

The mouse and the turtle

In some turtle graphics programs (but almost never in a CLASSWAR class definition), it's useful to sense the current position of the pointing device. You can accomplish this with:

MOUSE

which reads the position of the AutoCAD pointing device immediately—without waiting for the button to be pressed—and places the floating point X , Y , and Z co-ordinates of its position on the stack. If an error occurs, zeroes are returned for all three co-ordinates.

If you'd rather wait for the pick button, allowing the user to use any of AutoCAD's methods of specifying co-ordinates, you can use:

prompt PICKPOINT

where *prompt* is a string that is presented to the user to request the point. As with **MOUSE**, the co-ordinates entered are left on the stack. The general **ARG** acquisition facilities of CLASSWAR provide much more flexibility in obtaining input from the user; **PICKPOINT** and **MOUSE** are provided primarily for simple stand-alone turtle programs.

Cleaning up after turtles

To reset the active turtle to the defaults of a newborn hatchling, use:

RESET

This places the turtle in the X - Y plane at $(0, 0, 0)$ pointed along the positive Y axis. Its height is set to 0.5, its width to 0.4, and its depth to 0.25. The pen is put down, the icon becomes visible, and the tracks are set as ephemeral vectors.

You can redraw the AutoCAD screen and thereby discard all ephemeral vectors left by turtles with:

REDRAW

Where's that turtle?

You can retrieve the current co-ordinates of the active turtle with the **XCOR**, **YCOR**, and **ZCOR** primitives. Each places the respective floating point co-ordinate on the stack.

Loading turtle programs

You can load a file containing turtle programs or any other ATLAST code with:

filename LOAD

where *filename* is the name of the file to be loaded, with a default extension of ".tur" if none is given. Unlike **CLASSDEF** and **CLASSFILE**, programs loaded this way are not stored in the drawing. Consequently, **LOAD** is primarily useful for loading small utilities used only during one drawing session.

Transforming turtle tracks

Co-ordinates traced out by the turtle are transformed by the SGLIB current transformation before being inserted into the AutoCAD database. This allows you to combine the ease of turtle definition of geometric figures with the power of SGLIB in assembling transformations to translate, rotate, and scale the figure as desired. The SGLIB transformation is reset to the identity transform before a **DRAW** method is invoked, so turtle-generated geometry will not be transformed unless you explicitly define a transformation with the SGLIB primitives within the **DRAW** method.

External Methods

By declaring a method "EXTERNAL," you can implement it in another ADS application, coresident with CLASSWAR. When that method is invoked, whether from the AutoCAD command line or within another class definition, CLASSWAR collects the class and instance variables along with the arguments to the method (if any) and sends them to the other application for processing. Since the other application can be written in any programming language with an ADS binding and may be shipped in compiled binary form to the customer, external methods provide both maximum execution speed and protection of proprietary components of user applications.

The task of creating an external method program is simplified by the **CLASSTOC** command provided by CLASSWAR. This command generates a ready-to-use C language header (.h) file which, in conjunction with the source code module **CLASSAPP.C** supplied with CLASSWAR, manages all communication between the external method and CLASSWAR, leaving the application devel-

oper only the job of actually implementing the actions taken by the method. /*

Declaring external methods

All classes must have CLASSWAR class definitions, but for classes with entirely external methods these are just short “stubs.” Here is a simplified version of the class definition MOUNTAIN.CLS, the fractal mountain sample class supplied with CLASSWAR.

```
\ Fractal mountain class definition
```

```
PUBLIC:
```

```
integer mesh_size
real fractal_dimension
real power_factor
integer colour_mode
integer random_seed
```

```
PRIVATE:
```

```
static integer interface_level
```

```
method newclass
```

```
{
    1 interface_level !
}
```

```
external command method draw
```

```
{
}
```

```
method acquire
```

```
{
    8 mesh_size !
    1.75 fractal_dimension 2!
    1.0 power_factor 2!
    0 colour_mode !
    0 random_seed !
    this object.inspect
}
```

This class definition contains conventional declarations for its instance and class variables and a normal ACQUIRE method which simply sets all instance variables to their defaults and uses OBJECT.INSPECT to display an INSPECT box to allow the user to change them.

The DRAW method, however, is declared EXTERNAL and, being externally implemented, contains no code. When this class is loaded into AutoCAD and a header file generated with the CLASSTOC command, the result is as follows (edited slightly to fit on the page):

```
External method interface
definition for class MOUNTAIN
```

```
*/
```

```
typedef struct {
```

```
/* Class variables */
```

```
long interface_level;
```

```
/* Instance variables */
```

```
long mesh_size;
ads_real fractal_dimension;
ads_real power_factor;
long colour_mode;
long random_seed;
```

```
} s_mountain;
```

```
static s_mountain mountain;
```

```
/* Message protocol description */
```

```
#define Gv(x) ((char *) &(mountain.x))
```

```
#define Gp(x) ((char *) (mountain.x))
```

```
#ifndef m_Protocol_defined
```

```
#define m_Protocol_defined 1
```

```
typedef struct {
```

```
int p_type;
char *p_field;
```

```
} m_Protocol;
```

```
typedef struct {
```

```
char *methname;
void (*methfunc)();
```

```
} method_Item;
```

```
extern int beg_method(), end_method();
```

```
extern void define_class();
```

```
#endif
```

```
static m_Protocol mP_mountain[] = {
```

```
{ 71, Gv(interface_level) },
{ 71, Gv(mesh_size) },
{ 40, Gv(fractal_dimension) },
{ 40, Gv(power_factor) },
{ 71, Gv(colour_mode) },
{ 71, Gv(random_seed) },
{ -1, NULL}
```

```
};
```

```
#undef Gv
```

```
#undef Gp
```

```
/* Protocol for DRAW method */
```

```
static m_Protocol aP_draw[] = {
```

```
{ -1, NULL}
```

```

};

extern void M_draw_mountain();
#define draw_mountain void \
    M_draw_mountain() { if \
    (beg_method(mP_mountain, \
    aP_draw) != RTNORM) return; {
#define end_draw_mountain } end_method(); }

/* Method definition table */

method_Item MOUNTAIN[] = {
    {"MOUNTAIN.DRAW", M_draw_mountain},
    {NULL, 0}
};
#define mountain_methods void main(c,v) \
    int c;char *v[];{main_method(c,v,MOUNTAIN);}

```

Implementing external methods

Using the definitions in this file, coding methods in the external application is straightforward. This is a simplified version of the DRAW method from the sample external method application MTNAPP.C.

```

#include <stdio.h>
#include "adslib.h"
#include "mountain.h"

    mountain_methods

/* MOUNTAIN -- Make a mountain. */

draw_mountain
    float *a;
    int i, j, n, urs;

    if (mountain.interface_level > 1) {
        ads_printf("Interface level %d.\n",
            mountain.interface_level);
    }

    n = mountain.mesh_size;
    fracdim = mountain.fractal_dimension;
    powscale = mountain.power_factor;
    wtype = mountain.colour_mode;
    urs = mountain.random_seed;

    if (urs == 0) {
        initseed();
        urs = rseed;
    }
    /* Return random seed used */
    mountain.random_seed = urs;

    initgauss(urs);

```

```

    spectralsynth(&a, n, 3.0 - fracdim);

    free((char *) a);
end_draw_mountain

```

The external method application first includes the header file, `mountain.h`, created with the `CLASSTOC` command. After declaring any regular C variables and functions needed in the application (I've elided these from this listing for brevity), the application declares the start of the method implementations with the statement:

```
classname_methods
```

where *classname* is the name the class had when loaded in AutoCAD. Each method function is delimited by the sequence:

```
methodname_classname
end_methodname_classname
```

where *classname* is again the name of the class, and *methodname* is the name of the method being declared. These macros generate the entire function header, initialisation, and termination code. You simply place the body of the function between them.

Referencing variables

Within an external method function, you access the instance and class variables of the object being manipulated with:

```
classname.varname
```

where *classname* is the AutoCAD name of the class and *varname* is the name of the variable as declared in the original class definition (make sure you choose a variable name that's acceptable as an identifier in C!).

If you change any of the instance and/or class variables, the changes are automatically transmitted back to CLASS-WAR and stored in the AutoCAD database.

Methods with arguments

External methods may have arguments, just as built-in methods do. The arguments to external methods are transmitted to them along with the instance and class variables and are referenced in a similar manner. Suppose we add another method to our `MOUNTAIN.CLS` definition, as follows:


```

2variable two 2.0 two 2!

external command method rougher ((
    real "How much rougher" two default
    ARG_nozero ARG_noneg + argmodes
    "Little Lots Smooth" keywords
    arg ))
{
}

```

This method allows us to adjust the fractal dimension of the mountains, making them rougher or smoother. Adding this method will result in the following definitions in the `mountain.h` file written by CLASSTOC.

```

/* Arguments for ROUGHER method */

typedef struct {
    struct {
        int kwflag;
        union {
            char *kwtext;
            ads_real value;
        } kw;
    } arg1;
} aS_rougher;

static aS_rougher rougher;

/* Protocol for ROUGHER method */

#define Gv(x) ((char *) &(rougher.x))
#define Gp(x) ((char *) (rougher.x))

static m_Protocol aP_rougher[] = {
    { -40, Gv(arg1) },
    { -1, NULL }
};

#undef Gv
#undef Gp

extern void M_rougher_mountain();
#define rougher_mountain void
M_rougher_mountain() {
    if (beg_method(mP_mountain,
        aP_rougher) != RTNORM) return; {
#define end_rougher_mountain } end_method(); }

```

Using this definition, we can code the method function as follows:

```

/* ROUGHER -- Make a mountain
    rougher by increasing its
    fractal dimension. */

```

```

rougher_mountain
    if (rougher.arg1.kwflag) {
        ads_printf("\nKeyword: %s\n",
            rougher.arg1.kw.kwtext);
    } else {
        mountain.fractal_dimension +=
            rougher.arg1.kw.value;
    }
end_rougher_mountain

```

Here we access the argument, which can either be a number or an optional keyword string, as:

classname.argn

where *classname* is the AutoCAD class name and *n* is the argument number, with the first numbered 1. If the argument has no optional keywords, this is the full name of the argument. If keywords were declared for the argument with the KEYWORDS specification, the argument is a structure containing the following subfields:

kwflag An int which is nonzero if a keyword was entered and zero if a normal value was furnished for the argument.

kw.kwtext If kwflag is nonzero, a string containing the keyword entered.

kw.value If kwflag is zero, the value of the argument.

Whether referenced directly as *argn* or as *argn.kw.value*, the argument will be declared with the C data type corresponding to the type of argument being passed.

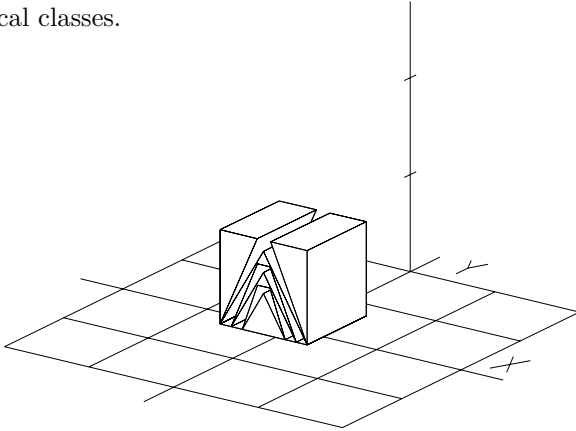
Linking and running

An external method application written as described above is prepared for use by compiling the source code module CLASSAPP.C with the same C compiler and modes used by the rest of the application, then linking all application modules, the object from CLASSAPP.C, and the ADS library file for the host machine into an ADS application. You can split the method functions across as many C source files as you like, but only one file may contain the *classname_methods* statement which generates the main program for the application.

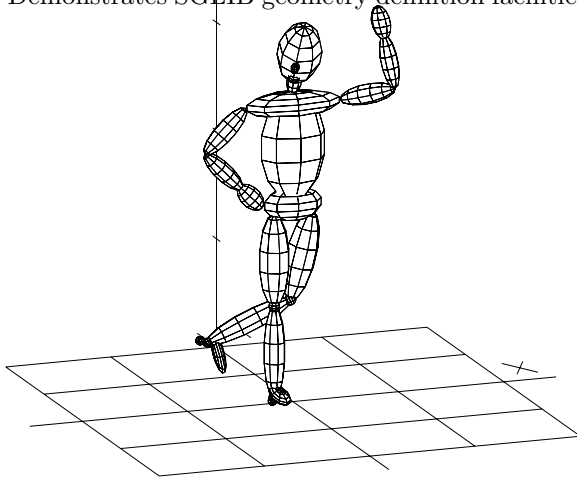
The application should be loaded into AutoCAD before any of the methods it implements are used. You can load it either explicitly with the "(xload *filename*)" statement or by naming it in the `acad.ads` file. If an external method is invoked and the application that implements it is not loaded, CLASSWAR will report an error and terminate the command.

Sample Classes

A collection of sample classes are supplied with CLASSWAR. These classes are intended to illustrate the various features of CLASSWAR more than be useful applications, but examination of them should help in defining your own practical classes.



AILOGO. The Autodesk logo, realised as an extruded solid. Demonstrates SGLIB geometry definition facilities.

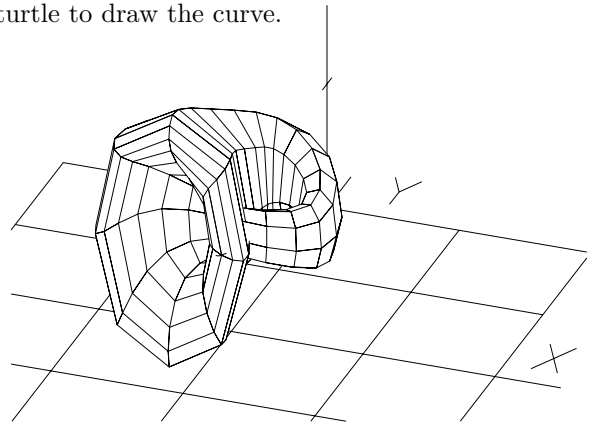


BLOBMAN. The Blobby Man. You can make s/he/it stand at attention with the **ATTENTION** message or wave at you with the **WAVE** message. Use **INSPECT** to create your own postures. Demonstrates nested transformations and the SGLIB geometry definition facilities. This class definition is large; you'll have to increase **CLASHEAP** well above the default of 5000 to load it.

DPOLY. This is a labeled polygon class derived from **POLY**. It uses `ads_entmake` to label each polygon with

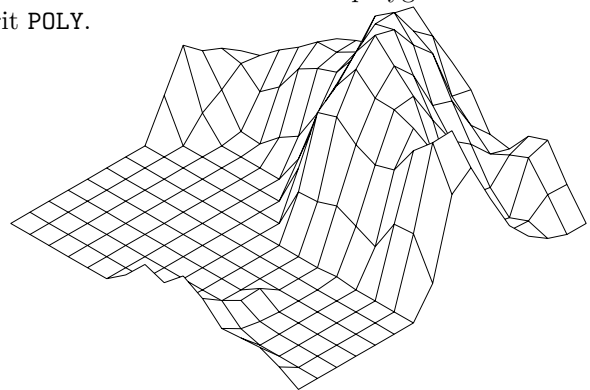
the number of sides as a text item.

HILBERT. The Hilbert curve, recursively defined. Uses the turtle to draw the curve.



KLEIN. The Klein bottle, defined with SGLIB.

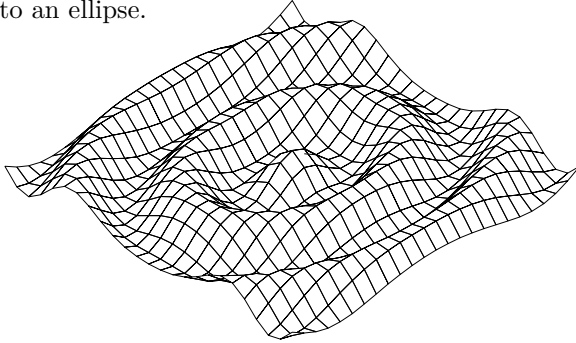
LPOLY. A stand-alone labeled polygon that doesn't inherit **POLY**.



MOUNTAIN. A fractal mountain application that demonstrates **EXTERNAL** methods. To use this class, you must remake the contents of the "mtnapp" subdirectory of your

CLASSWAR directory, then load the application into AutoCAD with (`xload "mtnapp"`) before using the commands defined by the MOUNTAIN class.

PARAM. Three-dimensional parametric curves. Defaults to an ellipse.

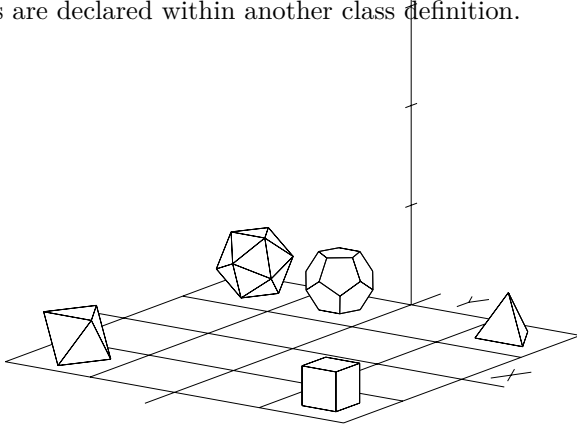


PLOT3D. Three dimensional plot of a function of X and Y . Demonstrates `ads_entmake` object creation.

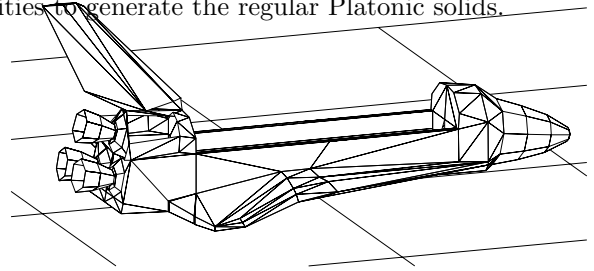
POLAR. Two-dimensional parametric polar co-ordinate curves. Defaults to the Nephroid of Freeth.

POLY. A simple polygon object. Uses the turtle for drawing.

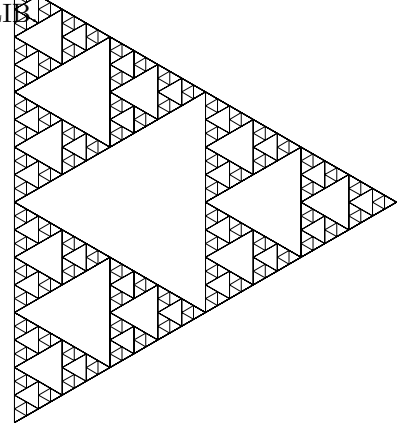
4POLY. A class that uses four instances of LPOLY to create an object containing four distinct labeled polygons. Demonstrates instances of classes within a new class, message passing to constituent classes, and the behaviour of `PRIVATE:` and `PROTECTED:` variables when instances of classes are declared within another class definition.



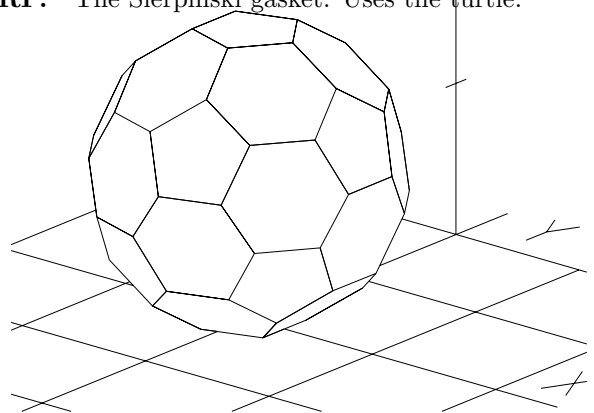
POLYHEDRON. Uses the SGLIB geometry creation facilities to generate the regular Platonic solids.



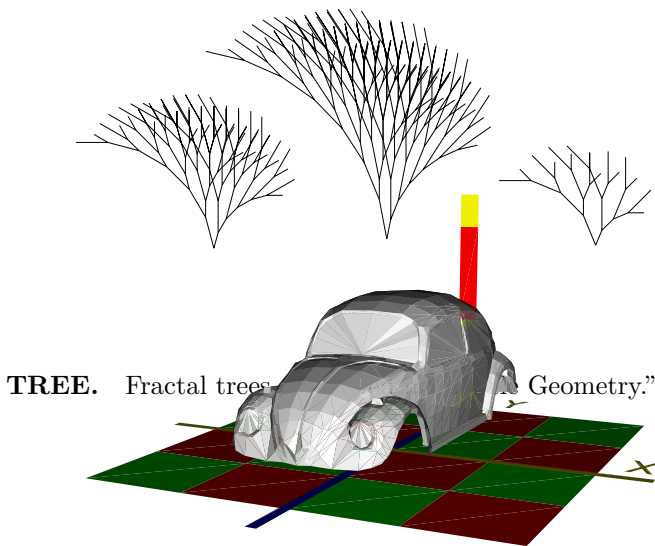
SHUTTLE. The space shuttle orbiter. A rat nest mesh defined with SGLIB.



SIERP. The Sierpiński gasket. Uses the turtle.



SOCBAL. The soccer ball / implosion bomb. Uses SGLIB.



VW. The VW bug defined using SGLIB.

CLASSWAR Today and Tomorrow

Since CLASSWAR implements a general object oriented programming environment within a showroom stock AutoCAD Release 11, there has been some confusion among those who've examined it so far about how CLASSWAR will interact with our plans to restructure AutoCAD in Release 12 around an object-oriented database in a manner that allows entities and commands to be implemented in an identical fashion and with equivalent performance regardless of whether implemented within the AutoCAD core or in an external application. In this section I'd like to dispel as much of this confusion as I can by explaining, in a question and answer format, how CLASSWAR fits with Releases 11 and 12 and how future developments in AutoCAD will affect the evolution of CLASSWAR.

Question: Does the advent of CLASSWAR make the Release 12 object oriented database project unnecessary?

Answer: Not at all. CLASSWAR is a user-level object oriented programming interface which presently maps its operations into the existing set of AutoCAD facilities available through ADS. Because the internals of AutoCAD are not structured in an object-oriented manner, several desirable facilities are missing from CLASSWAR. The most

vexing of these limitations is the inability for user-defined objects to supply methods that overload built-in AutoCAD commands such as **MOVE**, **ERASE**, and **TRIM**. In AutoCAD Release 12, these built-in commands will be methods of a system-supplied entity superclass, inherited by all user-defined entities but capable of being redefined by any entity that wishes different treatment by a command.

Q: Does the inability to redefine built-in commands make CLASSWAR unusable for practical applications?

A: No. AutoCAD's automatic transformation of the extended entity data that stores a CLASSWAR object's parameters allows most user defined objects to behave properly when edited with standard AutoCAD commands. The AME project has demonstrated that complex application-defined objects can be introduced this way and integrate smoothly with AutoCAD.

Q: Will CLASSWAR be compatible with Release 12? Will I be able to use class definitions I create on Release 11 with Release 12?

A: Hey, that's two questions! First, CLASSWAR will be completely compatible with Release 12. As far as AutoCAD is concerned, CLASSWAR is simply an ADS application that uses standard AutoCAD facilities, albeit in an unorthodox fashion. Since Autodesk is committed to maintaining 100% upward compatibility for ADS applications, CLASSWAR should run with Release 12 with at most a recompilation and relink with the Release 12 ADS library. If we maintain binary compatibility of ADS applications in Release 12, the existing CLASSWAR executable application should work without difficulties. Since CLASSWAR is compatible, all class definitions developed with it will also work in Release 12 without modification.

Q: How will CLASSWAR benefit from Release 12?

A: While the Release 11 CLASSWAR will run on Release 12, I anticipate we'll develop an extended, upward-compatible version of CLASSWAR for Release 12. Internally, the program will be restructured to replace the current ADS, entity attribute, and anonymous block architecture with direct calls to the Release 12 object oriented database manager. Externally, class definitions will now be able to overload existing AutoCAD commands (simply by defining methods named "ERASE," "TRIM," etc.) and receive control when overloaded commands are invoked on their objects. In addition, a rich set of internal AutoCAD facilities, including all built-in commands, will become available to CLASSWAR methods by being defined as methods of the entity superclass.

Q: I'm developing a big application in C. Why should I use CLASSWAR as its "wrapper"?

A: By packaging your application as external methods of a stub CLASSWAR definition you receive several benefits. First, you can let CLASSWAR worry about all the low-level details of working with the AutoCAD database and interacting with ADS. Not only does this simplify your job as an application developer, it allows your application to automatically benefit when more powerful and efficient database access mechanisms for applications are introduced in Release 12. If your application were peppered with ADS calls, you would have a major programming task on your hand to convert it to the new interface and reap the extra performance it will offer. Second, by specifying your application objects as CLASSWAR class definitions, you publish the external interface of your application in a form that encourages your customers and other developers to build upon your application. By allowing your application to be "designed into" other applications, you increase its usefulness and expand your potential market as well as making your product more useful. Since the actual algorithms of your application remain written in C and shipped in binary form, you disclose no proprietary information by doing this.

Q: Isn't CLASSWAR really just a sugar-coating of the existing AutoCAD database and not object oriented at all?

A: Yes, CLASSWAR is an interface to our existing Release 11 database facilities. But those facilities, particularly extended entity data, provide the foundations required to implement a true object oriented programming environment. A language is generally deemed object oriented if it provides *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*. CLASSWAR provides all of these mechanisms, in the manner of much-vaunted object oriented languages such as C++ and Smalltalk.

Q: Isn't this way too complicated and arcane for our developers and users to digest?

A: Complicated and arcane compared to what? Yes, mastering the basic concepts of CLASSWAR does take some time and effort, but once accomplished you can create new user-defined entities with minimal effort. If you don't use CLASSWAR, you're forced to roll your own code to manage extended entity data, ADS command definition, entity creation, etc. I expect most users will prefer to let CLASSWAR worry about these details rather than figuring them out for themselves. Besides, our customers are among the most intelligent and resourceful software users in the world; they have repeatedly confounded predictions that they couldn't master Lisp programming, 3D,

shaded rendering, and virtually every advanced feature we've given them. I'm confident that if we place CLASSWAR in their hands, they'll accurately evaluate its merits and demerits, apply it where it makes sense, and let us know how we can improve it to better meet their needs.

Q: Suppose CLASSWAR doesn't prove out in the market. Won't we end up stuck maintaining it forever to satisfy a splinter group of users?

A: No. Since CLASSWAR is a pure ADS application, if we decide we don't want to go on supporting it we can simply place its source code in the public domain and let its dedicated supporters do with it as they wish.


Q: Why rush this out with Release 11? Why not retain it as a mid-life kicker, or else package it with Release 12 when we'll be making a joyful noise about our object oriented architecture?

A: With Release 11 we're putting unprecedented power to create user-defined objects into our customers' hands, but we are providing very little guidance about how to proceed with the tools we're making available. As we've groped our own way through building applications with these tools, we've discovered that many of the techniques that work are subtle and that many pitfalls await the unwary. CLASSWAR provides a clearly marked path through the minefield. Applications developed within its guidelines will almost certainly work on any AutoCAD platform, be upwardly compatible with new releases, interoperate with other applications, and be extensible by users. By releasing CLASSWAR now, we can avert a painful learning curve and difficult period as each developer struggles to figure out how to build robust applications from the low-level ADS tools.

Summary and Conclusions

The development of ADS and the introduction of extended entity data has rendered Release 11 able to support general user defined entities. The Eagle/AME project has demonstrated that the facilities work, and possess the speed and generality needed to support real applications. The user defined entity capability is implicit in the new mechanisms, however, not a coherent package explicitly documented and promoted to that end. In addition, the techniques required to implement user defined entities are highly specialised and somewhat subtle. Consequently, we run the risk that much of the true power of Release 11, power which if properly applied may transform the world's view of the capabilities of AutoCAD and its fu-

ture extensibility, may lie latent—undiscovered by the users and developers whom it might benefit the most. Or, even worse, we may find ourselves deluged by a torrent of unreliable, inextensible, and mutually incompatible applications.

CLASSWAR can act as unifying force as the scope of AutoCAD applications broadens beyond anything we've seen to date. Requiring no material sacrifices of efficiency or proprietary protection of applications that adhere to its standards, it offers faster implementation, guaranteed correct interface with ADS and AutoCAD, and an open object definition architecture that encourages users and developers to incorporate applications as components of larger software systems. CLASSWAR provides Autodesk a head start in clearly positioning AutoCAD as an object oriented CAD system, one that supports a wide variety of applications that can be assembled, like building blocks, into solutions that benefit our customers. 

*John Walker
Muir Beach, California
February 25–May 6, 1990
16079 lines of code*

CLASSWAR Programming Language Syntax

The following syntax, given in a form resembling that used in the Algol 68 Revised Report, describes the overall structure of a CLASSWAR class definition. Each statement in the metalanguage begins with the name of the element being defined followed by a colon. Nonterminal symbols are written in normal roman letters and may consist of any number of words. Terminal symbols appear in teletype font and are quoted. A comma between symbols indicates concatenation; a semicolon delimits alternative forms. Each definition is terminated with a period. Since ATLAST code can be embedded anywhere in a class definition this syntax is not complete but it does specify the essential structure of a valid program. Upper and lower case letters are treated identically by CLASSWAR except within quoted string constants.

Program

Program: Derivation part, Declaration part, Method part.

Derivation

Derivation part: Parent class name, “DERIVED”;
Parent class name, “PUBLIC”, “DERIVED”;
Empty.

Parent class name: “:”, Symbol.

Declarations

Declaration part: Declaration, Declaration part; Empty.

Declaration: Access specifier, Variable specifier.

Access specifier: “PRIVATE:”; “PUBLIC:”; “PROTECTED:”; “TEMPORARY:”; Empty.

Variable specifier: Class variable indicator, Variable type, Variable name.

Class variable indicator: “STATIC”; Empty.

Variable type: Simple type; String length, “CHARACTERS”.

Simple type: “INTEGER”; “REAL”; “SCALEFACTOR”; “DISTANCE”; “POINT”; “TRIPLE”; “DISPLACEMENT”;
“DIRECTION”; “POINTER”.

Variable name: Symbol.

Methods

Method part: Method, Method part; Empty.

Method: Access specifier, Method declaration.

Method declaration: External option, Command option, Method definition.

CLASSWAR Programming Language Syntax

External option: “EXTERNAL”; Empty.

Command option: “COMMAND”; Empty.

Method definition: “METHOD”, Method name, Argument list, “{”, Method body, “}”.

Method name: Symbol.

Argument list: “(”, Argument sequence, “)”;

Argument sequence: Argument declaration; Argument declaration, Argument sequence.

Argument declaration: Argument type, Argument prompt, Argument option list, “ARG”.

Argument type: Simple type; “CHARACTERS”; “ANGLE”; “ORIENTATION”; “CORNER”; “KEYWORD”.

Argument prompt: Quoted string.

Argument option list: Argument option, Argument option list; Empty.

Argument option: Default value pointer, “DEFAULT”;
Base point pointer, “BASEPOINT”;
Acquisition initget modes, “ARGMODES”;
Keyword string, “KEYWORDS”.

Default value pointer: Symbol.

Base point pointer: Symbol.

Acquisition initget modes: Positive integer.

Keyword string: Quoted string.

Syntactic Elements

Variable name: Symbol.

String length: Positive integer.

Positive integer: Digit; Digit, Positive integer.

Symbol: Letter, Symbol tail.

Symbol tail: Alphanumeric, Symbol tail; Empty.

Quoted string: “”, *Any sequence of characters. Backslash forces a quote or backslash*, “”.

Alphanumeric: Digit; Letter.

Digit: “0”; “1”; “2”; “3”; “4”; “5”; “6”; “7”; “8”; “9”.

Letter: “A”; “B”; “C”; “D”; “E”; “F”; “G”; “H”; “I”; “J”; “K”; “L”; “M”; “N”; “O”; “P”; “Q”; “R”; “S”; “T”;
“U”; “V”; “W”; “X”; “Y”; “Z”; “_”.

Empty: “”.

CLASSWAR Primitives: Alphabetical Reference

{	→	Begin method OBJECT Marks the beginning of the executable code that implements a method.
}	→	End method OBJECT Marks the end of the executable code of a method.
((→	Begin argument list OBJECT Delimits the start of the optional argument list for a COMMAND METHOD.
))	→	End argument list OBJECT Marks the end of an argument list for a COMMAND METHOD.
<-	→	Send virtual OBJECT The preceding message is treated as a virtual message. This allows access to methods of the parent class which may have been overloaded by methods defined in a derived class, and references to fields of other classes with the same names as fields of the current class.
ADS_ENTGET	name →	Load entity ADS The AutoCAD entity specified by 2VARIABLE <i>name</i> is loaded into the current result buffer chain. If an error occurs, the result buffer chain will be void. (This condition can be detected by inquiring the presence of the 0 (entity name) group with the “GROUP?” primitive.)
ADS_ENTLAST	name → status	Last entity ADS The name of the last (most recently created) entity in the AutoCAD database is stored in the 2VARIABLE given by <i>name</i> . The ADS <i>status</i> is left on the stack.
ADS_ENTMAKE	→ status	Create entity ADS The entity defined by the current result buffer chain is added to the AutoCAD database. The <i>status</i> returned by AutoCAD is left on the stack. This status is positive if the entity was successfully added, negative in case of error.
ADS_ENTMOD	→ status	Modify entity ADS The entity defined by the current result buffer chain is modified in the AutoCAD database to conform to the values in the result buffer chain. The ADS <i>status</i> is left on the stack.
ADS_ENTNEXT	name resname → status	Next entity ADS Given the address of an entity <i>name</i> (stored in a 2VARIABLE), stores the name of the next entity in the database in 2VARIABLE <i>resname</i> and leaves the ADS <i>status</i> from the operation on the stack. If <i>name</i> is zero rather than a pointer to an entity name, the name of the first entity in the database is stored in <i>resname</i> .

CLASSWAR Primitives: Alphabetical Reference

ADDGROUP	<i>gcode</i> →	Add group to item ADS Adds a new group of type <i>gcode</i> to the end of the current item.
ANGLE	→	Declare angle argument OBJECT A method argument representing a relative angle in radians is declared.
ARG	→	Declare argument OBJECT Declares an argument, either within a “(”–“)” argument list, or a procedural argument requested within a method.
ARGMODES	<i>modes</i> →	Argument modes OBJECT The argument modes specified on the stack are used in the next argument acquisition.
BACK	<i>fdist</i> →	Turtle back TURTLE The active turtle backs up the distance given by <i>fdist</i> , drawing if the pen is down.
BASEPOINT	<i>p1</i> →	Base point OBJECT The specified point is used as the base point in the next argument acquisition.
CHARACTERS <i>x</i>	<i>n</i> →	Declare string OBJECT A character string variable or method argument is declared. If used to declare a variable <i>x</i> , the maximum length string the variable can store is given by <i>n</i> –1 and the variable assumes the current storage class and modes. The string length, <i>n</i> , is <i>not</i> specified on the stack when declaring a string method argument.
CLASSNAME	<i>class s1</i> →	Class name OBJECT Stores the name of the <i>class</i> into string <i>s1</i> .
CLEARITEM	→	Clear current item ADS All groups of the current item are deleted.
COMMAND	→	Declare command method OBJECT Used before “METHOD” to declare a method as an AutoCAD command as well as a CLASSWAR method.
CORNER	→	Declare corner argument OBJECT A floating point triple method argument representing the corner of a box in three-dimensional space is declared.
CUBE	→	Draw cube SGLIB A cube with unit vertex co-ordinates is drawn at the origin.
DEFAULT	<i>ptr</i> →	Declare default value OBJECT The variable pointed to by the pointer at the top of the stack is used as the default value in the next argument acquisition.
DELGROUP	<i>group</i> →	Delete group ADS The group selected by <i>group</i> is deleted from the current item.

CLASSWAR Primitives: Alphabetical Reference

DERIVED	parent →	Declare derived class OBJECT Used at the head of a class declaration to define a derived class. Expects the name of the parent class in the form “: <i>classname</i> ” on the stack.
DIRECTION x	→	Declare direction OBJECT A floating point triple variable or method argument x , representing a direction vector in three-dimensional space, is declared using the current storage class and modes.
DISPLACEMENT x	→	Declare displacement OBJECT A floating point triple variable or method argument x , representing a displacement vector, is declared using the current storage class and modes.
DISTANCE x	→	Declare distance OBJECT A floating point distance variable or method argument x is declared using the current storage class and modes.
DODECAHEDRON	→	Draw dodecahedron SGLIB A dodecahedron with unit edge length is drawn at the origin.
DRAWFACE	n $p1$ $p2$ $p3$ $p4$ $visbit$ →	Draw face SGLIB Draws a planar face with n vertices given by the 3 element vectors $p1$ through $p4$. If the face has only 3 vertices, the last vertex should be specified twice. If the 2^i bit is set in $visbit$, the edge starting with p_{i+1} is invisible.
DRAWVEC	$p1$ $p2$ →	Draw vector SGLIB A vector is drawn between the points given by the 3 element vectors $p1$ and $p2$.
EXTERNAL	→	Declare external method OBJECT Used before “METHOD” to declare a method as external. External methods are executed by sending messages to other applications with <code>ads_invoke()</code> .
FORWARD	$fdist$ →	Turtle forward TURTLE The active turtle advances the distance given by $fdist$, drawing if the pen is down.
GPLANE	$fancy$ →	Ground plane SGLIB A ground plane is drawn at the origin. If $fancy$ is nonzero, the ground plane is composed of solid faces; otherwise it’s a wire frame image.
GROUP	group → value	Group value ADS The value of the group in the current item selected by <i>group</i> is placed on the top of the stack. The value is stored as an integer, a floating point value, a triple of floating point values for co-ordinates, the address of a temporary string buffer, or the address of a temporary string buffer with a binary chunk length on the top of the stack depending on the group’s data type.

CLASSWAR Primitives: Alphabetical Reference

GROUP?	group → flag	Test group present ADS If the designated <i>group</i> is present in the current item −1 is placed on the top of the stack. If no such group appears in the current item, 0 is returned.
GROUPCOUNT	→ n	Number of groups in item ADS The number of groups in the current item is placed on the top of the stack. This number can be used in conjunction with the $-(10000 + n)$ group specification to scan streams of extended entity data groups with identical group codes.
HIDETURTLE	→	Hide turtle icon TURTLE The icon representing the active turtle is removed from the screen. All turtle operations continue to function normally.
ICOSAHEDRON	→	Draw icosahedron SGLIB An icosahedron with unit edge size is drawn at the origin.
INTEGER <i>x</i>	→	Declare integer OBJECT A 32 bit integer variable or method argument <i>x</i> is declared using the current storage class and modes.
KEYWORD	→	Declare keyword argument OBJECT A method argument representing a keyword string is declared.
KEYWORDS	s1 →	Specify argument keywords OBJECT The string <i>s1</i> is used as the alternative keyword list for the next argument acquisition. The argument string is specified in the standard manner used by <code>ads_getkeyword()</code> .
LEAVETRACKS	type →	Set objects created by turtle TURTLE The type of “tracks” left by the turtle when it moves with the pen is down is set to <i>type</i> . If 0, ephemeral vectors that disappear on the next REDRAW are generated. If 1, Polyline entities are created, with a new Polyline started whenever a break appears in the turtle’s path due to the pen being raised and lowered. If 2, planar faces (represented as rat nest meshes) are generated from closed paths traced by the turtle. If 3, individual Line entities are created for each step taken by the turtle.
LEFT	degrees →	Turn turtle left TURTLE The turtle turns <i>degrees</i> counterclockwise about its local Z axis.
LISTEN:	turtle →	Activate turtle TURTLE The specified <i>turtle</i> is activated and will respond to subsequent turtle commands. The previously active turtle becomes inactive (but remembers its position, orientation, and state).

CLASSWAR Primitives: Alphabetical Reference

LOAD	filename →	Load turtle program TURTLE A turtle program (or for that matter, any <u>ATLAST</u> program you like) is loaded and executed from the file given by <i>filename</i> . An extension of “.tur” is assumed if none is given.
MATCOPY	m1 m2 →	m2 = m1 SGLIB Matrix <i>m1</i> is copied to <i>m2</i> .
MATMUL	m1 m2 m3 →	m3 = m1 × m2 SGLIB The matrices <i>m1</i> and <i>m2</i> are multiplied and the product is stored in <i>m3</i> .
MATIDENT	m1 →	Identity matrix SGLIB The matrix <i>m1</i> is set to the identity matrix.
MATORIE	a b c d e f p q r m1 →	Orientation matrix SGLIB The matrix <i>m1</i> is set to an arbitrary orientation matrix with the values given by <i>a</i> through <i>r</i> placed in the first three rows and columns.
MATPERS	alpha zn zf m1 →	Perspective matrix SGLIB The matrix <i>m1</i> is set to a perspective transformation matrix with a field of view of <i>alpha</i> radians, a near clipping plane at a distance <i>zn</i> from the eye, and a far clipping plane at a distance <i>zf</i> .
MATPRINT	m1 →	Print matrix SGLIB The matrix <i>m1</i> is printed on the AutoCAD text screen.
MATRIX <i>x</i>	→	Declare matrix SGLIB A new 4 × 4 transformation matrix named <i>x</i> is declared and initialised to the identity matrix.
MATROT	theta axis m1 →	Rotation matrix SGLIB The matrix <i>m1</i> is set to a rotation transformation matrix that rotates <i>theta</i> radians about the designated <i>axis</i> (0 for the <i>X</i> axis, 1 for the <i>Y</i> axis, and 2 for the <i>Z</i> axis).
MATSCAL	fx fy fz m1 →	Scaling matrix SGLIB The matrix <i>m1</i> is set to a scaling transformation matrix with scale factors given by <i>fx</i> , <i>fy</i> , and <i>fz</i> .
MATTRAN	fx fy fz m1 →	Translation matrix SGLIB The matrix <i>m1</i> is set to a translation matrix with displacements given by <i>fx</i> , <i>fy</i> , and <i>fz</i> .
ME	→ turtle	Current turtle TURTLE The currently active turtle is placed on the stack.
METHOD <i>x</i>	→	Begin method OBJECT Begins compilation of a method named <i>x</i> .
MOUSE	→ fx fy fz	Current mouse position TURTLE The current co-ordinates of the AutoCAD pointing device are placed on the stack. The co-ordinates are returned immediately without waiting for a button to be pressed.

CLASSWAR Primitives: Alphabetical Reference

OBJECT.INSPECT	<i>o1</i> → <i>stat</i>	Inspect object OBJECT A dialogue box allowing examination and modification of the public variables of object <i>o1</i> is displayed. If the OK box is selected <i>stat</i> will be -1 , otherwise <i>stat</i> will be zero.
OBJECT.SPY	<i>o1</i> → <i>stat</i>	Spy on object OBJECT A dialogue box allowing examination and modification of all variables of object <i>o1</i> is displayed. If the OK box is selected <i>stat</i> will be -1 , otherwise <i>stat</i> will be zero.
OBJ!	<i>o1 o2</i> →	Copy object OBJECT The object <i>o1</i> is copied to <i>o2</i> . The objects must be of the same class.
OCTAHEDRON	→	Draw octahedron SGLIB An octahedron with unit edge size is drawn at the origin.
ORIENTATION	→	Declare orientation argument OBJECT A method argument representing an absolute (bearing) angle in radians is declared.
PENDOWN	→	Turtle pen down TURTLE The pen of the active turtle is lowered. Turtle motion while the pen is down leaves tracks in the form set by LEAVETRACKS.
PENUP	→	Turtle pen up TURTLE The pen of the active turtle is raised. Turtle motion while the pen is up does not leave tracks.
PICKPOINT	<i>prompt</i> → <i>fx fy fz</i>	Pick point TURTLE The co-ordinates of a point are obtained from the user after issuing the <i>prompt</i> specified by the string on the stack. If <i>prompt</i> is 0, no prompt is issued.
PITCH	<i>degrees</i> →	Turtle pitch up TURTLE The active turtle pitches up <i>degrees</i> (or down if <i>degrees</i> is negative) by rotating that amount about its local X axis.
PNT	<i>fx fy fz n</i> →	Define point SGLIB The point specified by <i>fx</i> , <i>fy</i> , and <i>fz</i> is added to the point database as point <i>n</i> . The lowest numbered point is 1.
POINT <i>x</i>	→	Declare point OBJECT A floating point triple variable or method argument <i>x</i> , representing a location in three-dimensional space, is declared using the current storage class and modes.
POINT@	<i>p1</i> → <i>fx fy fz</i>	Load point OBJECT The co-ordinates of point <i>p1</i> are placed on the stack.
POINT!	<i>fx fy fz p1</i> →	Store point OBJECT The co-ordinates <i>fx</i> , <i>fy</i> , and <i>fz</i> , are stored into point <i>p1</i> .

CLASSWAR Primitives: Alphabetical Reference

POINT?	$p1 \rightarrow$	Print point OBJECT The co-ordinates of point $p1$ are printed on AutoCAD's text screen.
POINTER x	\rightarrow	Declare pointer OBJECT A database pointer (handle) variable or method argument x is declared using the current storage class and modes.
POINTGET	$fx\ fy\ fz\ p1 \rightarrow$	Set point SGLIB The 3 element vector $p1$ is initialised to (fx, fy, fz) .
POINTCOPY	$p1\ p2 \rightarrow$	p2 = p1 SGLIB The 3 element vector $p1$ is copied to $p2$.
POLY	$0\ v1\ v2\ \dots v_n \rightarrow$	Draw polygon SGLIB An n -sided polygon consisting of the vertices $v1, v2, \dots$ through v_n is added to the database, either as a member of a rat nest mesh if enclosed in a RATON–RATOFF sequence, or 3DFace entities otherwise.
PRINTGROUP	$group \rightarrow$	Print group ADS The value of the specified <i>group</i> of the current item is printed on the AutoCAD text screen.
PRINTITEM	\rightarrow	Print current item ADS All groups of the current item are printed on the AutoCAD text screen.
PRIVATE:	\rightarrow	Set private access OBJECT Sets the accessibility of subsequently declared variables and methods as private. Private components can be accessed only within the class definition in which they appear.
PROTECTED:	\rightarrow	Set protected access OBJECT Sets the accessibility of subsequently declared variables and methods as protected. Protected components can be accessed within the class definition in which they appear and by classes derived from it, but not by other classes that declare instances of the class.
PUBLIC	\rightarrow	Set public inheritance OBJECT Used before DERIVED, causes all variables and methods of the parent class to be available as methods of the derived class.
PUBLIC:	\rightarrow	Set public access OBJECT Sets the accessibility of subsequently declared variables and methods as public. Public components can be accessed within the class definition, in classes derived from it, and by other classes that declare instances of the class.
RATON	\rightarrow	Begin rat nest mesh SGLIB Begins definition of a rat nest mesh with the PNT and POLY primitives. RATON must be issued before the first POLY primitive of the mesh.

CLASSWAR Primitives: Alphabetical Reference

RATOFF	→	End rat nest mesh SGLIB Closes a rat nest mesh begun with the last RATON. Call this after generating the last POLY of the mesh.
REAL x	→	Declare real OBJECT A floating point variable or method argument x is declared using the current storage class and modes.
REDRAW	→	Redraw screen TURTLE The AutoCAD screen is redrawn.
RESET	→	Reset turtle TURTLE The active turtle is reset to the default settings: position (0,0,0), heading (0,1,0), normal (0,0,1), pen down, turtle icon visible, ephemeral tracks, and icon 0.5 units high. In addition, the current SGLIB transformation is set to identity.
RETURN	→	Return from method OBJECT Return immediately from a method. EXIT cannot be used within a method; use RETURN instead.
RIGHT	degrees →	Turn turtle right TURTLE The turtle turns <i>degrees</i> clockwise about its local Z axis.
ROL	degrees →	Turtle roll right TURTLE The active turtle rolls <i>degrees</i> to the right (or left if <i>degrees</i> is negative) by rotating that amount about its local Y axis. The primitive is named “ROL” to avoid conflict with the <u>ATLAST</u> “ROLL” primitive.
SCALEFACTOR x	→	Declare scale factor OBJECT A floating point scale factor variable or method argument x is declared using the current storage class and modes.
SCRATCHPAD	class → scratch	Get scratchpad OBJECT Given a <i>:class</i> name, returns the scratchpad instance used to read and write the object to the AutoCAD database.
SETGROUP	value group →	Set group value ADS The selected <i>group</i> is set to the <i>value</i> that precedes it on the stack. The form of the <i>value</i> depends on the group’s data type; see the GROUP primitive for details.
SETHEADING	degrees →	Set turtle absolute heading TURTLE The active turtle’s heading is set to the bearing given in <i>degrees</i> , with 0 representing the X axis and angles increasing counterclockwise.
SETPOSITION	fx fy fz →	Set turtle absolute position TURTLE The active turtle is moved to the absolute three-dimensional location given by <i>fx</i> , <i>fy</i> , and <i>fz</i> , drawing if the pen is down.
SHOWTURTLE	→	Show turtle icon TURTLE The icon representing the active turtle is displayed on the screen.

CLASSWAR Primitives: Alphabetical Reference

SPHERE	→	Draw sphere SGLIB An approximation of a unit sphere is drawn at the origin. If <code>OBJECT.WIREFRAME</code> is nonzero, the sphere is represented by three mutually perpendicular circles. Otherwise, the sphere is drawn as a mesh with <code>OBJECT.CNSEGS</code> latitudinal and longitudinal tiles.
STATIC	→	Declare class variable OBJECT Causes the next variable declared to be a class, as opposed to an instance, variable. Note that the <code>STATIC</code> declaration affects only the next variable declared, not subsequent declarations.
TEMPORARY:	→	Set temporary storage OBJECT Sets the storage type of subsequent variable declarations as temporary. Temporary variables retain their values only during the execution of a method and are not stored with the instance or class.
TETRAHEDRON	→	Draw tetrahedron SGLIB A tetrahedron of unit edge length is drawn at the origin.
THIS	→ obj	Current instance SGLIB The instance being operated on by the current method is placed on the top of the stack.
TRIPLE <i>x</i>	→	Declare point OBJECT A floating point triple variable or method argument <i>x</i> is declared using the current storage class and modes. A <code>TRIPLE</code> is not modified by AutoCAD transformations of the object containing it.
TURTLE <i>x</i>	→	Declare turtle TURTLE A new turtle named <i>x</i> is created and given the default settings described under <code>RESET</code> . The new turtle is not automatically activated; use <code>LISTEN:</code> to activate it when desired. If you only need one turtle, you don't need to explicitly declare one. A turtle named "KELVIN" is automatically declared for you by CLASSWAR.
VECADD	<i>p1 p2 p3</i> →	<i>p1</i> = <i>p1</i> + <i>p2</i> SGLIB The vector sum of the 3 element vectors <i>p1</i> and <i>p2</i> is stored in <i>p3</i> .
VECCOPY	<i>v1 v2</i> →	<i>v2</i> = <i>v1</i> SGLIB The 4 element vector <i>v1</i> is copied to <i>v2</i> .
VECCROSS	<i>p1 p2 p3</i> →	<i>p3</i> = <i>p1</i> × <i>p2</i> SGLIB The cross (vector) product of the two 3 element vectors is calculated and stored in <i>p3</i> .
VECDOT	<i>p1 p2</i> → <i>f1</i>	<i>f1</i> = <i>p1</i> · <i>p2</i> SGLIB The dot (inner) product of the two 3 element vectors is calculated and placed on the top of the stack.
VECGET	<i>fx fy fz v1</i> →	Set vector SGLIB The 4 element vector <i>v1</i> is initialised to (<i>fx</i> , <i>fy</i> , <i>fz</i> , 1).

CLASSWAR Primitives: Alphabetical Reference

VECMAG	$p1 \rightarrow f1$	$f1 = p1$ SGLIB The magnitude of the 3 element vector $p1$ is placed on the stack.
VECNORM	$p1 \ p2 \rightarrow$	$p2 = p1/ p1$ SGLIB A unit vector in the same direction as the 3 element vector $p1$ is stored in $p2$.
VECPRINT	$v1 \rightarrow$	Print vector SGLIB The 4 element vector $v1$ is printed on AutoCAD's text screen.
VECPUT	$v1 \rightarrow fx \ fy \ fz$	Put vector SGLIB The components of the 4 element vector $v1$ are stored on the stack. They are normalised by dividing them by the fourth element of the vector.
VECSCAL	$p1 \ fs \ p2 \rightarrow$	$p2 = p1 \times fs$ SGLIB The scalar product of the 3 element vector $p1$ and the scale factor fs is stored into the 3 element vector $p2$.
VECSUB	$p1 \ p2 \ p3 \rightarrow$	$p3 = p1 - p2$ SGLIB The vector difference of the 3 element vectors $p1$ and $p2$ is stored in $p3$.
4VECTOR x	\rightarrow	Declare 4 element vector SGLIB A new 4 element homogeneous co-ordinate vector named x is declared and initialised to $(0, 0, 0, 1)$.
VECMAT	$v1 \ m1 \ v2 \rightarrow$	$v2 = v1 \times m1$ SGLIB The 4 element vector $v1$ is multiplied by matrix $m1$ and the result is stored in the vector $v2$.
XCOR	$\rightarrow fx$	Turtle X co-ordinate TURTLE The X co-ordinate of the present location of the active turtle is placed on the stack.
XORIE	$a \ b \ c \ d \ e \ f \ p \ q \ r \rightarrow$	Orientation transform SGLIB An arbitrary orientation matrix with the values given by a through r placed in the first three rows and columns is composed with the current transformation.
XPERS	$alpha \ zn \ zf \rightarrow$	Perspective transform SGLIB A perspective transformation with a field of view of $alpha$ radians, a near clipping plane at a distance zn from the eye, and a far clipping plane at a distance zf is composed with the current transformation.
XPOP	\rightarrow	Pop transformation SGLIB The topmost transformation on the transformation stack is removed and replaces the current transformation.
XPUSH	\rightarrow	Push transformation SGLIB The current transformation is saved on the transformation stack.
XRESET	\rightarrow	Reset transformation SGLIB The current transformation is set to the identity transform and all transformations pushed on the transformation stack are discarded.

CLASSWAR Primitives: Alphabetical Reference

XROT	theta axis →	Rotation transformation SGLIB A rotation transformation that rotates <i>theta</i> radians about the designated <i>axis</i> (0 for the X axis, 1 for the Y axis, and 2 for the Z axis) is composed with the current transformation.
XSCAL	fx fy fz →	Scaling transformation SGLIB A scaling transformation matrix with scale factors given by <i>fx</i> , <i>fy</i> , and <i>fz</i> is composed with the current transformation.
XTHEN	→	New transformation SGLIB Equivalent to XPOP XPUSH. Used for readability when assembling objects with many nested transformations.
XTRANS	fx fy fz →	Translation transformation SGLIB A translation with displacements given by <i>fx</i> , <i>fy</i> , and <i>fz</i> is composed with the current transformation.
YAW	degrees →	Turtle yaw left TURTLE The active turtle yaws <i>degrees</i> to the left (or right if <i>degrees</i> is negative) by rotating that amount about its local Z axis. This is a synonym for the LEFT primitive, provided to make three-dimensional turtle navigation more intuitive.
YCOR	→ fy	Turtle Y co-ordinate TURTLE The Y co-ordinate of the present location of the active turtle is placed on the stack.
ZCOR	→ fz	Turtle Z co-ordinate TURTLE The Z co-ordinate of the present location of the active turtle is placed on the stack.