POSE

The Programmable Object Simulation Engine version 1.0
An Object Simulation Environment for POV-Ray
by Bob Hood

Table of Contents

1.0 ***Introduction to POSE***

1.1 The Programmable Object Simulation Engine

POSE is an object simulation system built explicitly for POV-Ray 2.x (a freely-available, multi-platform ray tracing system). It is an animation-only system, providing no Graphical User Interface (GUI) modelling capabilities. As its name implies, POSE provides an environment where a subset of objects and constructs available from POV-Ray can be brought to "life" through the use of a built-in Object Behavior Language.

Although POSE is the result of more than a year's worth of work, POSE came into existence as the result of frustration on my part. I wanted something that I could do computer animation with, but I could not afford any of the professional animation systems that were on the market at the time (NewTek's Video Toaster, 3D Studio, etc.). POV-Ray itself provides an extremely crude mechanism for producing animation, but it simply wasn't enough. I looked all over the world (via the 'Net), and could find nothing to satisfy my needs.

So, to keep myself busy until the day when I could afford a better system, I began working on POSE. What you have now are the results of my busy-work.

1.2 What POSE Isn't

POSE is not a full-featured computer animation system. While you can accomplish some impressive computer animation with POSE (see the included example animation done with POSE called "ship"), it lacks a number of features that would make it rival software that is used for professional animation.

One of the most obvious differences is that POSE is a non-interactive system where its objects are concerned. With POSE, you create a "live" object, and then program its actions through the use of an Object Behavior Language source file. This file is compiled by POSE, and evaluated (executed) for its related object by POSE as each frame of animation is generated. This requires the object programmer to think ahead and consider the behavior of the object.

Although this method of generating computer animation might seem awkward, it can, in fact, generate some interesting effects.

1.3 Where To Get POSE

The Shareware distribution of POSE can be found at the following sites:

MS-DOS:

File name: **pose1dos.zip**
Anonymous FTP: ftp.netcom.com
directory /pub/bh/bhood/pose
WWW: ftp://ftp.netcom.com/pub/bh/bhood/pose

OS/2 (2.x/Warp):
File name: **pose1os2.zip**
Anonymous FTP: ftp.netcom.com
directory /pub/bh/bhood/pose
WWW: ftp://ftp.netcom.com/pub/bh/bhood/pose

HP-UX 9.x:
File name: **pose1hpux.tar.gz**
Anonymous FTP: ftp.netcom.com
directory /pub/bh/bhood/pose
WWW: ftp://ftp.netcom.com/pub/bh/bhood/pose

Please feel free to distribute these Shareware copies of POSE as far and as wide as you like. If you have distributed POSE to a well-known Internet site, please let me know so that I may update this location information. Thanks.

1.4 Where To Get POV-Ray

You can acquire a copy of POV-Ray 2.x from any of the following places:

CompuServe (GO GRAPHDEV)
Sections 8 POV Sources and 9 POV Images.

PC Graphics Area on America On-Line
Jump keyword "PCGRAPHICS"

"You Can Call Me Ray" BBS in Chicago
(708) 358-5611

"The Graphics Alternative" BBS in El Cerrito, CA
510-524-2780 (PM14400FXSA v.32bis 14.4k, Public)
510-524-2165 (USR DS v.32bis/HST 14.4k, Subscribers)

"PI Squared" BBS, Maryland
(301)-725-9080 (14.4K bps 24 hrs.)

Internet
alfred.ccs.carleton.ca (134.117.1.1).

WWW

ftp://alfred.ccs.carleton.ca/pub/pov-ray/POV-Ray2.2

1.5 Registering POSE

The Shareware version of POSE contains a number of limitations, and is designed simply as an evaluation of the complete system. The full list of limitations of the Shareware version are:

- Each project is limited to one scene
- Each scene is limited to three (3) POSE objects and/or constructs, not including the camera (any single scene can only have one camera in either the Shareware or registered versions of POSE).
- Creation/loading of project primitives is disabled.

The Registered version of POSE contains no such artificial limitations.

You should have received a file called REGISTER.FRM in your POSE distribution. Please refer to this document for information about ordering the registered version of POSE.

If this file was not provided in your POSE distribution, Appendix C contains a reprint of this file, including the POSE registration form.

2.0 ***Getting Started***

2.1 Installing POSE

POSE requires very little setup work before you can begin programming simulations. Regardless of the platform you are working on, the distribution archive you received should exist in a format that will create all subdirectories required by POSE.

POSE does not require any specific directory name or location from which to operate. Create a directory where you wish to install the POSE distribution, and change to that directory. A good place would be a sub-directory under your POV-Ray installation.

If you are working with a PKZIP archive, extract the archive with the "-d" switch, which instructs PKUNZIP to create the subdirectories contained within the archive.

If you are working with an archive for a supported UNIX platform, decompress the archive with the GNU ZIP utility, and then extract the resulting tar file with the command "tar xf <archive>."

2.1.1 POSE Environment Variables

POSE uses several environment variables during its operation. These variables are documented below.

2.1.1.1 POVRAYOPT

This variable is actually employed by POV-Ray to contain run-time options that you do not wish to type in each time you invoke the ray tracer. This is the only variable that POSE requires to exist in its environment to operate. At a minimum, this variable should contain an "include" declaration (+I) so that POSE can locate the directory or directories where POV-Ray include files reside.

The POVRAYOPT variable setting I use looks like:

```
+x +w1024 +h768 +d4 +p +Ic:\graphics\pov\include +b100
```

2.1.1.2 PROJECTS

This variable should contain a path to an existing directory where projects will be created when you issue the `project` command.

If it is not set in the POSE environment, project creation defaults to the directory where POSE is currently executing.

2.1.1.3 EDITOR

When editing Object Behavior Language source files, this variable is used to determine the name of the text editor to invoke on the file. A sample variable setting to use QEdit as your source file editor might look like:

```
SET EDITOR=c:\qe3\q
```

If this variable is not set, editing for source files cannot take place from within the POSE environment.

2.1.1.4 PREVIEW

POSE gives you the ability to preview generated frames of animation by invoking POV-Ray from within the POSE environment. This variable contains the command to invoke POV-Ray, and must include three replaceable parameters that represent

1. The scene number
2. The frame number of the current scene
3. The preview image width
4. The preview image height

These replaceable parameters appear in the PREVIEW setting as "%s", the tokenized representation of a string parameter in a C printf() statement. A suggested PREVIEW variable setting for UNIX might appear as (assumes that the POV-Ray executable appears in the PATH variable):

(ksh)
```
export PREVIEW="povray +is%df%d.pov -f +w%d +h%d +d -v"
```
(csh)
```
setenv PREVIEW "povray +is%df%d.pov -f +w%d +h%d +d -v"
```

For OS/2 or MS-DOS,

```
set preview=povray +is%df%d.pov -f +w%d +h%d +d -v
```

If you enter this command at an MS-DOS or OS/2 shell prompt, you will need to "escape" the percent sign (%) under OS/2 and MS-DOS by including an additional percent sign. This is because it is a special character to the command shell (just as a dollar sign [$] is special to the UNIX shell). For example:

```
set preview=povray +is%%df%%d.pov -f +w%%d +h%%d +d -v
```

Your PREVIEW setting must include four replaceable parameters.

2.1.1.5 RENDER

When POSE prepares your scene for rendering (see section 5.19 concerning the `prepare` command), it creates a one or more render files. Each line in this render file corresponds to the POV-Ray command line to create the frame of animation. As with the PREVIEW variable, you can create a character string that represents this command in the RENDER environment variable. In the same fashion as PREVIEW, the RENDER value should have a number of replaceable parameters that represent

1. The current scene and frame (appearing twice)
2. The image width
3. The image height

RENDER employs the "%d" token to represent its replaceable parameters. For example, the RENDER variable I use holds the following value:

(ksh)
```
export RENDER="povray +ft +is%df%d.pov +os%df%d.tga +w%d
+h%d -d -v +A +J +b100 -p > /dev/null 2>&1"
```

As with PREVIEW, entering commands at the shell prompt will require you to "escape" any percent signs (%) under OS/2 and MS-DOS.

Your RENDER setting must include six replaceable parameters.

2.2 Starting POSE

2.2.1 OS/2 and HP-UX

The OS/2 and HP-UX versions of POSE contain a single, stand-alone executable, requiring no additional files or data to run. Because of this, POSE can be started from anywhere, not just from the directory where it was installed. It is typically a good idea to start POSE in the directory where you will be housing simulation projects.

2.2.2. MS-DOS

The MS-DOS version of POSE runs in 80x86 protected mode. This mode of operation emulates OS/2 and UNIX in that it allows processes to execute in a flat-memory model.

To facilitate this mode of the Intel chip, two additional files are provided with the MS-DOS version. They are "32rtm.exe" and "dpmi32vm.ovl," and they can be found in the same directory as the POSE executable in the distribution (bin/). To execute POSE under MS-DOS, you simply need to ensure that both of these additional files can be found in one of the directories in your PATH variable. If you elect to add the POSE "bin" directory to your PATH, you need take no further action.

The protected-mode MS-DOS version of POSE will execute properly and has been tested under MS-DOS 5.x, Microsoft Windows 3.x Enhaced Mode in a DOS window, and under OS/2 Warp in a VDM window. This same system should function correctly under a Windows NT console window as well.

2.2.3 Character-based Interfaces

As was mentioned previously, release 1.0 of POSE does not make use of a Graphical User Interface (GUI). GUIs tend to require huge amounts of program development and maintenance, as they tend to be very platform-specific. For instance, Microsoft Windows only operates on Intel-based platforms, while X/Motif only functions under UNIX. Differences in these two APIs really turn one program into two different programs.

Using a text-based, cursor-addressable package moves the program's user interface closer to being portable across platforms, but even these packages (i.e., Curses) are not available on all platforms.

As a result, POSE presents a simple, line-based interface. Future releases of POSE might employ more-advanced, platform-specific user interfaces.

2.2.4 The POSE command interface

When you start POSE by entering "pose" at the shell prompt, you will be greeted with the POSE start-up message, and then the POSE command prompt.

```
POSE v1.0 for OS/2
Copyright (C) 1995 Bob Hood
All Rights Reserved

Registered to: < Evaluation copy, distribute freely >

Command:
```

When POSE starts, it will automatically scan for both the POVRAYOPT and PROJECT environment variables. If POSE cannot locate the POVRAYOPT variable, it will issue a message and terminate:

```
POSE v1.0 for MS-DOS32
Copyright (C) 1995 Bob Hood
All Rights Reserved

Registered to: < Evaluation copy, distribute freely >

ERROR:  Cannot locate POVRAYOPT variable in current environment
            Please ensure that POV-Ray is installed on your system
            and that this variable contains the correct path to
            the POV-Ray include directory or directories.
```

If the PROJECT variable does not exist, a warning message will be issued by POSE:

```
POSE v1.0 for HP-UX
Copyright (C) 1995 Bob Hood
All Rights Reserved

Registered to: < Evaluation copy, distribute freely >

WARNING:  Environment variable PROJECT unavailable
            defaulting to current directory

Command:
```

When you select a project to work with, the POSE command prompt will change to include the project name (don't worry about understanding the commands being sent to POSE quite yet, they will be covered later):

```
Command: project test
[ no primitives found for current project ]
test.?:
```

Once you have selected a project, you must then select a scene within that project to work with. As you can see in the previous example, the command prompt "test.?" contains a question mark where the scene number will appear. You select a scene from the current project by issuing the "scene" command:

```
test.?: scene 1
test.1:
```

You are now ready to begin working with other POSE commands. But first, let's look at how POSE maintains its list of objects for generating animation.

3.0 ***POSE Object Layers***

3.1 How Objects Exist In POSE

POSE attempts to provide you with the most flexibility in designing your objects. At one end of the spectrum, objects in POSE can exist as "live" entities that have programmed behavior; full POSE citizens. While at the other end object and scene attributes can exist completely outside of POSE's awareness, yet still be included in each frame of animation. To add to this, there are even layers in between where objects can reside and still directly generate animation.

3.1.1 Live objects

Live objects exist at the top of the POSE object food chain. These objects exist within each scene, and are those displayed when you ask POSE to `list objects` for a scene. They are either incarnations of one of the object types listed in section 4.0 -- direct representations of their POV-Ray counterparts (i.e., Sphere) -- or they act as proxies for pre-existing objects (primitives) or composites (a Union object is a collection of other POSE objects or primitives). These proxy and composite representations are comprised of any POSE object.

These objects can have programmed behaviors through the use of an Object Behavior Language source file.

3.1.2 Live primitives

Live primitives are very much like live objects. The differences are:

1. Live primitives exist in their own area
2. Live primitives belong to the project, not to any individual scene (i.e., there is only one live primitive in an entire project as opposed to any number of Spheres across scenes).

Live primitives can be added to a project by simply issuing the command `new primitive` from any scene prompt.

As with live objects, live primitives can also have programmed behaviors by attaching an Object Behavior Language source file to the primitive. Why would a live primitive need to have programmed behavior? How about a communications satellite with a bunch of little moving parts? The satellite as a whole would be a composite object (Union), but each individual moving part would be a primitive with its own behavior independent of the whole.

3.1.3 Phantom primitives

Phantom primitives are only a slight step above "static" primitives (see the next section). These primitives exist outside of the POSE operating space, and should have the ability to be referenced by name. An example of such a primitive would be an item that has been declared within a POV-Ray source file:

```
#declare MyEar = union {
      ...
}
```

This declaration would have been pre-modelled outside of POSE (see the included example project called "ship" for an example of this mechanism). Both a live and primitive object could reference this declaration by name (i.e., `MyEar`), but the POSE object acting as a proxy would know nothing about the makeup of this object. Only a limited amount of behavior would be programmable for such an object.

3.1.4 Static primitives

Static primitives, or "scene files," are items that exist completely outside of the POSE environment. POSE knows nothing about these primitives, and takes no more action on them than to faithfully replicate them in every frame generated for a scene. Each scene in a project can have its own scene file.

These static primitives can be composed of any legal POV-Ray instruction or construct. A typical use of a scene file is to house all the objects in an animation that are unmoving and unchanging. Buildings, trees, pillars -- anything that does not require control or behavior by POSE is a likely candidate for inclusion in a scene file.

One drawback of placing objects in a scene file: because they are outside the scope of POSE, objects that exist within the POSE framework will be ignorant of the existence of these static items. As such, the possibility exists that a POSE object could "walk through" a static item during your animation. Unless you've designed it that way, this sort of thing looks very unprofessional.

4.0 ***The POSE Object List***

4.1 The Object Interface

As you create objects within the POSE environment using the `new` command, you will be required to enter default information for the object you are creating. Although some objects may share common attributes, others require specialized information in order to properly generate themselves for animation.

This section will cover each available POSE object, illustrating the response generated by each object to acquire user input.

When you invoke the command `new object` from the POSE command line, you will be given a list representing types which are valid as "live" objects:

```
 1) point_light      2) area_light       3) spot_light
 4) sphere           5) blob             6) blob_component
 7) box              8) cylinder         9) cone
10) fog             11) plane           12) character
13) torus           14) disc            15) bicubic_patch
16) union           17) difference      18) intersection
19) composite
```

Had you elected to create a project primitive by entering the command `new primitive`, the above list would have been the same with the exception of the last option, #19. Primitives cannot be composites; they are themselves the stuff of which composites are comprised.

It needs to be noted that this section makes no attempt to explain what each object is. It is assumed that you have read and have a working understanding of POV-Ray and its capabilities. Please refer to the documentation provided with POV-Ray for a detailed discussion of these capabilities (see section 1.5 earlier in this document for information on how to get a copy the POV-Ray archive).

In the examples given in the remaining sections, entries typed by the user are highlighted in bold.

4.1.1 Point Light

Selecting type #1 will invoke the dialog to create a Point Light object:

```
Type? 1
Enter <X,Y,Z> of location:
0 0 0
Enter Red|Green|Blue values of color, or color name:
White
```

As you can see, you can specify a color name (the `list colors` command will show you valid color entries), or you can specify the RGB value of the color by specifying these numeric values (from 0-255), separated by a vertical bar:

```
Enter Red|Green|Blue values of color, or color name:
197|255|13
```

Each light object in POSE (Point, Spot, and Area) prompt the object creator with the next series of dialog prompts. These prompts allow you to apply a "looks_like" condition to the light source. It accomplishes this by attaching an existing POSE object to the light source. In this, the POSE light objects become containers of other POSE objects.

```
Apply 'looks_like' setting from an existing object? (y/n) y

1.  Sphere @

Enter object # to use as 'looks_like' value:
object: 1
```

4.1.2 Spot Light

The Spot Light object dialog appears below:

```
Type? 3
Enter <X,Y,Z> of center location: 10 10 0
Enter Red|Green|Blue values of color, or color name:
1|1|0.5
Enter <X,Y,Z> of point at location: 0 1 0
Enter light radius: 11
Enter light falloff: 25
Enter light tightness: 50
```

4.1.3 Area Light

The Area Light object builds on top of the Point and Spot Light objects to add its own requirements:

```
Type? 2
Enter <X,Y,Z> of center location:
0 50 0
Enter Red|Green|Blue values of color, or color name:
White
Enter <X,Y,Z> of Area Light length:
40 0 0
Enter <X,Y,Z> of Area Light direction:
0 0 1
Enter first and second array dimensions of Point Lights:
100 1
Adaptive value:
4
Use jitter? (y/n) y
Include Spot Light settings? (y/n) y
```

```
Enter <X,Y,Z> of point at:
0 1 0
Enter light radius:
11
Enter light falloff:
25
Enter light tightness:
50
```

4.1.4 Sphere

The following dialog is invoked for the Sphere object:

```
Type? 4
Enter <X,Y,Z> coordinates:
0 0 0
Enter sphere radius:
1
```

4.1.5 Blob

The Blob object is a container for Blob Component objects (see section 4.1.6). When you create a Blob object, there must exist in the scene one or more Blob Components from which to construct the new object. If no such objects exist, POSE will issue an error message, and abort the creation of this new object:

```
Type? 5
[ cannot create Blob object without pre-existing Blob Components ]
blob.1:
```

As with any "live" POSE object, each Blob Component that you render over into the care of the Blob object can have its own, independent behavior.

```
Type? 5

1.  Blob Component, strength 1, radius .5, @ 0 0 0
2.  Blob Component, strength 1, radius .5, @ 0 0 0
3.  Blob Component, strength 1, radius .5, @ 0 0 0

Enter component # to include in blob (enter "done" when complete):
component: 1
component: 2
component: done

The following components will be added to the blob:

1.  Blob Component, strength 1, radius .5, @ 0 0 0
2.  Blob Component, strength 1, radius .5, @ 0 0 0

Are you sure?
(y/n): y

Enter <X,Y,Z> coordinates: 2 0 -1
Enter blob threshold: .25
```

4.1.6 Blob Component

The Blob Component object is an atomic piece of the Blob object (see section 4.1.5). At least one Blob Component object must exist in a scene before a Blob object can be created.

Blob Components, although they can be assigned behavior, are not evaluated, nor is their POV-Ray image information placed into a frame file, unless they are under the management of a Blob object.

```
Enter component strength: 1
Enter component radius: 1
Enter <X,Y,Z> of component center: -.375 -.64952 0
```

4.1.7 Box

The following dialog example is used to specify a Box object:

```
Type? 6
Enter <X,Y,Z> of lower-left corner:
-1 -1 -1
Enter <X,Y,Z> of upper-right corner:
1 1 1
```

4.1.8 Cylinder

The Cylinder object is initialized with the following dialog:

```
Type? 7
Enter <X,Y,Z> of first end:
0 0 0
Enter <X,Y,Z> of second end:
0 1 0
Enter cylinder radius:
.5
```

4.1.9 Cone

Creating a Cone object will initiate the following dialog:

```
Type? 8
Enter <X,Y,Z> of first end:
0 0 0
Enter first end radius:
1
Enter <X,Y,Z> of second end:
0 1 0
Enter second end radius:
.10
```

4.1.10 Fog

The following dialog example is used to specify a Fog object:

```
Type? 9
Enter fog distance: 200
Enter fog color: Gray70
```

Although POSE allows you to create more, you would typically want only one Fog object per scene.

4.1.11 Plane

Creating a Plane object will initiate the following dialog:

```
Type? 10
Enter <X,Y,Z> of surface:
0 1 0
Enter offset:
-4
```

4.1.12 Character

The Character object employs those POV-Ray objects declared in the file "chars.inc" (included with the POV-Ray distribution). This file (as of version 2.x of POV-Ray) declares the following characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-+!@#$%^&*()[]

Your POSE Character object must be created as one of these supported characters. (You can employ your own characters, perhaps converted from Postscript using an external utility, by creating a Composite object. See section 4.1.19 for information about Composite objects).

```
Type? 11
Enter <X,Y,Z> of character:
0 0 0
Enter character:
W
```

4.1.13 Torus

The following dialog supports the creation of Torus object:

```
Type? 12
Enter <X,Y,Z> coordinates: 12 15 20
Enter torus Major radius: 6.4
Enter torus Minor radius: 3.5
```

4.1.14 Disc

The following dialog example is used to specify a Disc object:

```
Type? 14
Enter <X,Y,Z> of disc center: -2 -0.5 0
Enter <X,Y,Z> of disc normal vector: 0 1 0
Enter disc radius: 2
Enter disc hole radius (-1 for no hole): -1
```

4.1.15 Bicubic Patch

The Bicubic Patch object of POSE has a very limited dialog. The reason for this is because the Bicubic Patch object reads patch data from a disk file, instead of prompting the creator for a large number of numeric parameters (20 items).

The Bicubic Patch object expects this disk file to be in the following format:

<patch type>
<flatness>
<usteps>
<vsteps>
<x1> <y1> <z1>
...
<x16> <y16> <z16>

It is from a file of this format that the following dialog will initialize the Bicubic Patch object:

```
Type? 13
Enter filename containing bicubic patch data:
c:\patch1.dat
```

4.1.16 Union

Like the Blob object, the POSE Union object is a container. Unlike the Blob, however, the Union object can contain *any* form of POSE object, such as Blobs and even other Union objects. The only exception to this is the Camera object. The Camera object cannot be contained by or associated with any other object in any way.

This object is used to create a new, single object comprised of the objects it contains, and it corresponds to its CSG equivalent in POV-Ray.

An example of creating a POSE Union object might be:

```
Type? 14
```

```
1. PointLight @ <-20,20,0>, color White
2. PointLight @ <0,20,20>, color White
3. PointLight @ <20,20,0>, color White
4. PointLight @ <0,5,-10>, color White
5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Enter object # to include in union
(enter "done" when complete):
object: 5
object: 6
object: 7
object: 8
object: done

The following objects will be added to the union:

5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Are you sure?
(y/n): y
```

4.1.17 Difference

Again, like the POSE Union object, the Difference object is a container. As in the case of the Union object, the Difference object can also contain *any* form of POSE object, including other Difference objects. Of course, the one exception is the Camera object.

This object is used to create a Boolean difference of the objects it contains, and it corresponds to its CSG equivalent in POV-Ray.

An example of creating a POSE Difference object might be:

```
Type? 15

1. PointLight @ <-20,20,0>, color White
2. PointLight @ <0,20,20>, color White
3. PointLight @ <20,20,0>, color White
4. PointLight @ <0,5,-10>, color White
5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Enter object # to include in difference
(enter "done" when complete):
object: 5
object: 6
object: 7
object: 8
```

```
object: done

The following objects will be added to the difference:

5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Are you sure?
(y/n): y
```

4.1.18 Intersection

Another container object, the Intersection object, as with the Difference object, can also contain *any* form of POSE object, including other Intersection objects. The Camera object is excluded.

This object is used to create a Boolean intersection of the objects it contains, and it corresponds to its CSG equivalent in POV-Ray.

An example of creating a POSE Intersection object might be:

```
Type? 16

1. PointLight @ <-20,20,0>, color White
2. PointLight @ <0,20,20>, color White
3. PointLight @ <20,20,0>, color White
4. PointLight @ <0,5,-10>, color White
5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Enter object # to include in intersection
(enter "done" when complete):
object: 5
object: 6
object: 7
object: 8
object: done

The following objects will be added to the intersection:

5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>

Are you sure?
(y/n): y
```

4.1.19 Composite

The Composite object gives you the ability to bring "primtive" objects into the

realm of the scene's "live" objects. It is essentially a proxy, acting on the behalf of these primitives so that they may exist in the current scene.

There are two variations of the Composite object available in POSE: the first is created from items in the project's "primitive" area; the other associates with static, non-POSE objects.

In the following example dialog, a component called "LeftFrontFin", potentially modelled in another utility, has been added to the scene file with the following declaration:

```
#declare LeftFrontFin = object {
      ...
}
```

POSE would allow you to proxy this object in the following fashion:

```
Type? 17
Enter <X,Y,Z> coordinates:
0 0 0

1. PointLight @ <-20,20,0>, color White
2. PointLight @ <0,20,20>, color White
3. PointLight @ <20,20,0>, color White
4. PointLight @ <0,5,-10>, color White
5. Composite::LetterS @ <0,0,0>
6. Composite::LetterH @ <0,0,0>
7. Composite::LetterI @ <0,0,0>
8. Composite::LetterP @ <0,0,0>
9. Static Primitive

Enter primitive # to associate with composite (0=abort): 9

Enter Static Primitive identifier exactly as it
is declared in the scene file: LeftFrontFin
```

Once created, your new POSE Composite object can be assigned to an Object Behavior Language source file. Please note that the commands in this OBL source file will only effect the Composite object as a whole, not any individual component of the object for which it serves as proxy.

4.2 Object Identities

Each object described previously shares the same two closing prompts. These prompts are generated by POSE, not by the object you are creating.

4.2.1 Object Name

Objects in POSE can have an optional name. This name is not simply meant to foster affection for your objects, but it serves as a tag that POSE objects can

use to access one another from within the Object Behavior Language environment. See section 7.7 later in this document for an explanation of this feature. This name can be of any length, but cannot contain spaces.

The dialog prompt appears as:

```
Object name: ship
```

4.2.2 Object Behavior

Attaching an Object Behavior Language file to an object gives you the ability to bring an object to "life" within the animation. The next prompt allows you to specify the name of the Object Behavior Language source file.

Whenever you specify the name of an Object Behavior Language file within POSE, you should only use the root name of the file. POSE will automatically tack on an extension to the file name (".sim") and then look for the full file name.

```
Object behavior filename: ship
```

In the above example, POSE would expect to find a file named "ship.sim" in the directory for the current scene. If this file does not exist, you can have POSE create the file for you by issuing the command `edit ship`.

You should ensure that this file exists before issuing a `compile` command, or you will receive error messages.

5.0 ***The POSE Command List***

The POSE environment provides a number of commands to aid the simulation programmer in developing computer animation. The following sections document the commands that POSE understands as of release 1.0 of the engine.

It is important to note that commands listed in this section appear in random order. For instance, the `generate` command appears before the `new` command in this section, however you generally would not want to generate a frame of animation until you have first created objects and behaviors for the current scene.

5.1 quit

As the name implies, this command is used to terminate POSE and return to the shell prompt.

If you have modified the current scene in any fashion (i.e., added/deleted objects), POSE will issue a warning so that you do not unknowingly lose work:

```
test.1: quit
[ scene 1 of project "test" has been modified ]
[ please issue a "save" before exiting or ]
[ type "quit" again to discard changes ]
test.1:
```

No parameters are accepted by this command.

5.2 generate [#]

This command instructs POSE to generate a frame file (an input file to POV-Ray) by invoking each "live" object in the current scene. Before generating each frame file, POSE walks its list of objects and evaluates each one that contains a behavior.

The `generate` command can be invoked without parameters to create a single frame file in the next sequence. Frame files are managed as sequences beginning at frame #1 when a new project and scene is selected within POSE. As each frame is generated, the frame sequence increments by one. Scenes that have had frames generated previously can be "caught up" by using the `resume` command (see section 5.13).

You can also specify a number of frames to be generated by following the

command with a number. For instance, to generate 200 frames of sequential animation in the current scene, you would enter:

```
test.1: generate 200
```

5.3 pause

This command is typically used only in conjunction with the `debug` command (section 5.17). It instructs POSE to pause between the evaluation of each instruction of an object's Object Behavior Language file.

It is initially off, and no parameters are accepted by this command.

5.4 static [<file>]

Objects in POSE can exist at various levels (see section 3.0). This command associates "static" objects (and other miscellaneous POV-Ray settings) with a scene.

A static file contains nothing more than default POV-Ray data that will be included with each frame file generated by POSE. With one exception, any legitimate POV-Ray data structure or command can be included in a static file. The sole exception to this is the camera object. POSE *always* generates the camera object.

If this command is invoked without options, any existing static file for the current scene will be deleted.

The command can be invoked with a single parameter which is the path to a file that is to be used as the static file. A copy is made of the specified file, and any existing static file for the scene is overwritten.

5.5 new [object|primitive]

The `new` command is used to add a new object to the POSE environment. Objects can be added either to the current scene's "live" list (using the `object` parameter), or to the project's "primitive" list. Primitives can have defined behavior, and "live" objects can be constructed from these primitives.

If this command is invoked without parameters, POSE will assume that you wish to create a "live" object instead of a "primitive."

See section 3.0 earlier in this document for a discussion of how, where and why objects can live within POSE.

5.6 delete

Just as objects can be added to a scene, they can also be deleted. Once deleted from within POSE, unless an object has been saved to disk, it is permanently removed. If the object has been saved previously, you can restore the object by re-loading the current project (ignoring the warnings about losing modifications).

It is important to note that "primitives" within POSE are available to all scenes within a project. In other words, "live" objects belong to the scene; "primitives" belong to the project. As such, it is forbidden for any one scene to delete a primitive, as this may damage one or more other scenes in the project.

To avoid mass destruction of your work, you can only delete one object for each invocation of the `delete` command.

```
flight.1: delete

      1. PointLight @ <-20,20,0>, color White
      2. PointLight @ <0,20,20>, color White
      3. PointLight @ <20,20,0>, color White
      4. PointLight @ <0,5,-10>, color White
      5. Composite::LetterS @ <0,0,0>
      6. Composite::LetterH @ <0,0,0>
      7. Composite::LetterI @ <0,0,0>
      8. Composite::LetterP @ <0,0,0>

delete which object?
```

Because every scene must have a camera, this object is not subject to user management. The only action the user can take on a camera is to give it a behavior.

5.7 list [objects|primitives|colors|textures]

The list command is used to display a list of objects in either the scene's "live" area or the project's "primitives" area. It can also be used to display a list of colors and textures that are valid in the current installation of POV-Ray on your system.

5.7.1 objects

Using the `objects` parameter to the `list` command will display a directory of your main scene, or "live", objects. A sample listing might appear as:

```
flight.1: list objects

      Camera @ <0,0.5,-15>
      PointLight @ <-20,20,0>, color White
```

```
PointLight @ <0,20,20>, color White
PointLight @ <20,20,0>, color White
PointLight @ <0,5,-10>, color White
Composite::LetterS @ <0,0,0>
Composite::LetterH @ <0,0,0>
Composite::LetterI @ <0,0,0>
Composite::LetterP @ <0,0,0>
```

`flight.1:`

5.7.2 primitives

The output of the `list primitives` command is identical in format to that of `list objects`.

5.7.3 colors

This command will display all the colors that POSE determined to be valid in the current installation of POV-Ray.

```
Command: list colors
Yellow            Cyan              Magenta     Black
Aquamarine        BlueViolet        Brown       CadetBlue
Coral             CornflowerBlue    DarkGreen   DarkOliveGreen
...
Command:
```

5.7.4 textures

This command will display all the textures that POSE determined to be valid in the current installation of POV-Ray.

```
Command: list textures
Jade              Red_Marble        White_Marble      LBlue
Vein              Blood_Marble      Blue_Agate        Blue_Sky
Clouds            Rosewood          Glass2            Rust
...
Command:
```

5.8 dir

This command is an alternative to issuing the command `list objects`.

No parameters are accepted by this command.

5.9 save

You will want to issue this command to save the state of your current scene. This command saves both the "live" and "primitive" objects to disk so that they can be reloaded again.

No parameters are accepted by this command.

5.10 project <name>

This command allows you to specify the simulation project with which you wish to work. This command will also create a project if one cannot be found by the specified name:

```
Command: project test

Create new project called "test"?
(y/n):
```

See section 6.0 later in this document for a detailed discussion of how to create a new POSE project.

5.11 scene <number>

Projects within POSE are comprised of scenes (and scenes are comprised of objects that generate frames of animation). This command allows you to either select an existing project scene to work with, or create a new scene in the current project.

A scene must be created or selected before new POSE objects can be added.

This command takes a numeric parameter that specifies the scene number to which all future commands are applied.

5.12 merge

The `merge` command takes all scene map files for the current project (created by the `prepare` command, section 5.19) and merges them into a single file called "project.map."

No parameters are accepted by this command.

5.13 resume

You may not always wish to generate all frames of animation in one session with POSE. The resume command, when issued in a scene, will restore the scene to the next frame of animation based upon what was generated previously. For instance, if you had generated 50 frames of animation in a previous session, you might reset the state of a scene so that the next generate command would create frame number 51 by doing the following:

```
flight.1: resume
frame 1: evaluating camera
...
frame 50: evaluating camera
restored to frame #51
flight.1:
```

The `compile` command (5.16) must be issued on the current scene before the `resume` command is used.

No parameters are accepted by this command.

5.14 preview <frame number>

POSE gives you the ability to preview, through the use of POV-Ray, a frame of the current scene. This frame must have already been generated for `preview` to work.

The `preview` command relies on the PREVIEW environment variable to function (see section 2.1.1.4). If this variable has not been set, the preview command will not operate:

```
flight.1: preview 50
[ cannot preview frame 50:  PREVIEW variable not set ]
flight.1:
```

If everything is correct, the `preview` command will prompt you for the width and height of the preview window that POV-Ray will generate:

```
flight.1: preview 50
Preview width: 160
Preview height: 100
```

5.15 edit <object name|behavior file|"scene">

The `edit` command invokes the editor you specified in the EDITOR environment variable (see section 2.1.1.3). You can specify either a named POSE object (the object must have been both named and provided a behavior file name previously), or the name of a behavior file in the current scene (sans file name extension).

You may also use the literal term "scene" when issuing the `edit` command. This parameter causes POSE to invoked the editor command on the scene file for the current scene. This file is called "static.sim," and resides in the scenes working directory. Refer to section 5.4 for more information about the `static` command and creating scene files.

5.16 compile [<object name>]

The Object Behavior Language of POSE is an interpreted, C-like language. When an object is associated with a behavior source file, the source file must be compiled into an interpretable form before POSE can evaluate an object.

The `compile` command is used to invoke this source file compilation. It can be invoked without parameters, causing all objects in the current scene, either "live" or "primitive", to compile their associated behavior files. Invoking the command with the name of an object will cause only that object to compile its associated source file.

This command can be invoked at any time to force a re-compilation of an object's behavior file.

5.17 debug

Because the POSE Object Behavior Language is interpreted, each behavior is evaluated once for each frame of animation produced. The `debug` command toggles the interpreter to either be silent when evaluating behaviors (the default setting), or to be verbose, detailing what it is doing at each step as a compiled behavior file is executed.

Because `debug` has global effect (i.e., all behaviors are either silent or verbose), it is best to use the `debug` command on a single object/behavior that you suspect is not executing properly. This is accomplished by compiling only the single object you wish to observe. Once compiled, issue the `debug` command (and, optionally, the `pause` command) and generate single frames at a time.

It is initially off, and no parameters are accepted by this command.

5.18 populate

[the populate command is under construction. Its capabilities will be broadened in future updates of POSE]

There will likely come a time when you will want to generate a number of objects that are randomly dispersed within a given region of space. An example of this might be generating a series of stars in a space scene (an animation I'm constructed using POSE called "flight" used this technique to generate 3,000 sphere entries that appear in the scene files for several scenes).

The `populate` command offers this capability. It also goes a step further and

gives you the option to specify a exclusion region within the populate region (ever notice that planets don't have stars within their proximity during most space shots?).

When you populate a region, you will be prompted for the coordinates for the region to populate. These coordinates will correspond to a box-shaped region of the rendering space:

```
flight.1: populate
Enter lower-left coordinates of region: -100 -100 -100
Enter upper-right coordinates of region: 100 100 100
```

To this, the `populate` command will add the question of an area of exclusion:

```
Specify exclusion region? (y/n): y
Enter lower-left coordinates of region: -10 -10 -10
Enter upper-right coordinates of region: 10 10 10
```

You will then be prompted for type of object with which to populate the region. In the current release of POSE, the only object valid for `populate` is "sphere." Other object types will be made available in future releases of POSE.

```
1) sphere

Object type? 1
Number of objects to generate (0=abort): 3000
```

You should be aware that generating large numbers of objects within the POSE environment will require a large amount of RAM. It is likely that you could only accomplish such a generation of objects comfortably on a UNIX platform (the HP-UX system on which I generated 3,000 objects above caused the POSE process to consume over 25MB of RAM to accommodate them). It can also be done under OS/2, but with more noticeable effort.

Each object you generate can be attached to a single Object Behavior Language module. You would typically want to do this to give each object a common texture and/or color, or some behavior that each object would exhibit:

```
Attach a behavior to each object? (y/n): y
Object behavior filename: star
```

The name you enter should correspond to an Object Behavior Language file for the scene. In the example above, a file called "star.sim" should exist in the current scene's directory before the generated objects are compiled.

```
Populating region...
Object 1 of 3000
```

Objects generated with this command are created in the "live" are of the current scene (i.e., they belong to the scene, not the project).

5.19 prep[are]

The `prepare` command tells POSE that you wish to generate a number of files that can be useful in rendering a scene. This command will prompt you for a number of options.

```
flight.3: prep
Number of processors:
6
```

For those lucky few who have access to SMPs (symmetrical multiprocessors), POSE will attempt to accommodate you by grouping POV-Ray frame rendering commands into a series of files. Each batch file can then be invoked, one per available processor. These files will be created in the same directory as the project's frame files. The name of these files will be "s#p#" under UNIX, and "s#p#.cmd" under OS/2. The first number (#) in the file name corresponds to the current scene number, while the second numeric value corresponds to the processor that the file is intended to use.

`prepare` will look for the RENDER evironment variable (section 2.1.1.5). This variable, if it exists, should contain the POV-Ray command line to use to render each frame of animation. There must be a number of replaceable parameters within this value for POSE to properly generate the render files. If the RENDER variable does not exist in the environment, POSE will generate a generic POV-Ray command line for each frame.

```
Animation width:
160
Animation height:
100
```

These two parameters will be used when creating the commands that are placed into the rendering file mentioned previously.

Additionally, this command will create a file in this same directory called "scene#.map", where # represents the scene number to which the file belongs. This map file contains a listing of anticipated POV-Ray output file names for the current platform. The names in this file assume that the frame files will be created with the rendering batch files mentioned previously. Most utilities for converting still-frame Targa files into FLI, FLC or MPEG animation files will accept a file containing a list of input file names for processing. This is the purpose of the map files.

No parameters are accepted by this command.

5.20 reset

The reset command is used to restore a scene to its initial load state. This command will reset the frame count to one (1), remove all files from the project's frame working directory belonging to this scene, and recompile all objects that belong to the scene.

6.0 ***Creating A Simulation Project***

6.1 The `project` Command

When you need to start a new simulation project, you need to use the `project` command from within POSE. If you refer back to section 5.10, you will recall that the `project` command takes a single parameter, specifically the name to be used to identify the project. The name you enter will be used to name the directory created for the project:

```
Command: project test

Create new project called "test"?
(y/n): y
```

It is important to be aware of the environment in which you are creating the project. Under the MS-DOS and OS/2 FAT file systems, your project name (and all file names you specify, for that matter) should follow the 8-character limitation convention. Under UNIX, the length of the project name is typically not an issue. Under both environments, certain characters are not valid in file/directory names (e.g., "%" under OS/2 and MS-DOS, "&" under UNIX, space characters under both).

POSE does not enforce these conventions.

6.1.1 Directories created

When you direct POSE to create a new project, POSE creates a subdirectory under the PROJECTS path (or in the current directory if PROJECTS is unset). Under this directory, there are a number of subdirectories created initially.

6.1.1.1 frames

The "frames" directory is used by the project to house all the generated animation frame files from each scene.

Frame files use a naming convention that identifies them with their scene. The format of the file name is:

```
sSSSfFFF.pov
```

The `SSS` portion of the file name identifies the scene number of the frame file. The `FFF` section indicates the frame number that the file represents.

Also housed in this directory are the scene map and rendering script files (see section 5.19 concerning the `prepare` command for a description of these files).

6.1.1.2 misc

This directory is not used directly by POSE, and is created as a holding place for project special files (GIF image files, POV-Ray files, etc).

6.1.1.3 prim

Scene directories (discussed shortly) house objects that belong to scenes. This directory is used by POSE to contain project-owned primitive objects and their related Object Behavior Language source files. When you add a new primitive to a project, it is maintained here.

6.2 Working With Scenes

Projects in POSE are organized into a series of scenes. A project can have as few as one scene; as many as 999. When you first start POSE, you will need to specify a project with which to work, and then a scene within that project:

```
Command: project test
test.?: scene 1
```

If the scene you have specified does not exist, POSE will silently create the scene directory for you, and will establish the scene as the current working environment for POSE.

```
test.?: scene 1
test.1:
```

6.2.1 The "objects" file

A number of files reside in the scene directory. Among the most important of these is the "objects" file. This file is the repository of your scene's objects (those objects created with the `new` command).

If a default objects file exists for the project (see section 6.3 later in this section), POSE will automatically create the first "objects" file for your new scene from these objects.

6.2.2 The "static.sim" file

Another file that resides in the scene directory is the scene file. This scene file is called "static.sim" (note the POSE extension on the file). This is the file that

is created when you issue the `static` command (section 5.4) from within the POSE environment.

6.2.3 Object Behavior Language source files

As you assign and create Object Behavior Language (OBL) source files for your objects, they get deposited into the scene directory. OBL source files appear as the name given to the object when prompted for an OBL source file name (section 4.2.2), with the POSE file extension ".sim"

6.3 Default Scene Objects

As mentioned previously in this section, POSE supports a feature called default scene objects. Default scene objects are those created automatically for a scene when it is first created. For instance, you might want a default series of light sources, or project-wide "atmosphere" objects that span scenes.

Default scene objects are housed in a file called "default.sim," and this file resides in the top level of the project directory. A "default.sim" file is shipped with the POSE distribution that contains a camera object and four light sources. When you create a new project, POSE will take this application-level default file and make a copy of it for the new project.

If POSE cannot locate this file within the project directory when it creates a new scene, it will simply create the default camera object and proceed without complaint.

To create a new "default.sim" file, merely create a "dummy" project, populate the first scene of that project with the objects you wish to use as your defaults, and then copy the "objects" file from the scene directory over your "default.sim" file. You can them remove the project directory and all files.

6.4 Portability

POSE project and scene files have been designed to be completely portable across all the platforms on which POSE executes. All data is stored in ASCII text format, and converted back to binary by POSE when it is reloaded.

If you are transporting POSE data files between OS/2 and UNIX, you will need to use a utility to alter the format of the files for the target platform. For instance, OS/2 text files terminate each line with carriage return and line feed characters. UNIX text files only contain a line feed at the end of each line. Utilities are widely available (and relatively easy to write) for performing this conversion. POSE for UNIX will not read OS/2-formatted data files properly

without conversion.

7.0 ***The Object Behavior Language***

We now come to the heart of the Programmable Object Simulation Engine: the Object Behavior Language (or OBL for short). It is through the use of this built-in script language that POSE objects can be given a life of their own.

7.1 Thinking In Frames

As with any computer animation, all action is distilled down to the individual frame. The animation frame is the lowest level of operation that we need to be concerned with. This is especially true in POSE, where objects are evaluated during the generation of each frame file.

Because generating individual frames of animation is our goal with POSE, the structure of OBL source file needs to match this iterative process.

7.2 Behavior File Structure

The OBL source file is separated into two sections. This structure is designed to facilitate the idea of sub-function calls. Each OBL source file is essentially a sub-function that is invoked by POSE as each frame of animation is generated.

As with most sub-functions, there is a declaration, or initialization, section and a section containing code that is to be executed. Each of these sections is optional (i.e., one can exist without the other), but at least one must exist in each OBL source file.

7.2.1 The "initialize" section

The first section of an OBL source file is the "initialization" section. It has the follow structure:

```
initialize {
      ...
}
```

This section is evaluated only once, directly following each compilation of the OBL source file. It is used to declare, and optionally initialize, variables that will be local to the executable portion of the OBL file. Exported object variables (see Appendix B for a list) can also be initialized in this section (an object's <X,Y,Z> location vector, the camera's look_at vector, etc.).

The identifier for this section ("initialize") can be abbreviated to "init" if you wish.

7.2.2 The "evaluate" section

POSE looks for the "evaluate" section of the OBL source file for an object each time it generates a frame of animation. It has the following format:

```
evaluate {
      ...
}
```

As with the "initialize" section, the identifier for this section ("evaluate") can also be abbreviated to "eval."

7.3 Object Behavior Language Strucuture

7.3.1 Language keywords

The following table shows the keywords in the OBL. Most of them will be familiar if you have done C or C++ programming. Although there are comparatively few, teaching you the usage of each keyword goes beyond the current scope of this document. The best way to learn their usage to examine the OBL source files provided in the sample projects.

initialize	evaluate	if	else
while	const	continue	return
True/true	False/false	Nil/nil	=
+	-	*	/
==	!=	<=	>=
+=	-=	*=	/=
++	--	||	&&

7.3.2 Comments in OBL

You can embed comments into your OBL source files using standard C++ syntax. Single lines can be commented with the C++ double-slash:

```
init {
      scale(1,1,1.5);  // make ourselves larger along Z axis
}
```

7.3.3 Using pragma directives

A pragma directive is an instruction you give to a compiler to modify or affect its behavior. In OBL, each pragma directive only applies to the OBL source file in which it appears. OBL provides three different pragma options to the object programmer. They are: Nesting, Symbols, and Code. Each directive controls an aspect of the OBL compiler.

The Nesting directive sets the level of loop nesting to be allowed in the OBL

source file. Loop nesting involves `while` loops. The default value is ten (10) levels deep. For example, to set the nesting level to five (5) in the current OBL source file, you would place the following command somewhere before the `initialize` section:

```
#pragma nesting 5

init {
      ...
```

The Symbols directive sets the number of symbol table entries that the current OBL source file can accommodate. The default value is 200 entries. This number is quite high for a single source file, and would more than likely be lowered by an OBL source file to conserve memory.

```
#pragma symbols 30

init {
      ...
```

The Code pragma controls the number of instruction entries an OBL source file can consume. The default value for code instructions is 500 entries per source file. As with Symbols, this number is set high from the onset, and would more than likely be lowered to conserve memory. If you compile a behavior file with the `debug` flag on, POSE will inform you of exactly how many symbol table and code instruction entries an OBL source file requires. You can then adjust these values accordingly.

```
#pragma code 100

init {
      ...
```

Multiple pragma directives can be placed in an OBL source file:

```
#pragma nesting 3
#pragma code 100

init {
      ...
```

7.4 Variables

7.4.1 Declaring Local Variables

Let's declare some simple variables for an object (for now, we don't care what type of object it is):

```
init {
```

```
        xangle;
        tilt = 15;
}
```

We've just created two variables that will be used by the remainder of the OBL source file. Note that only one variable was initialized. The other was left uninitialized, and consequently its value is undefined.

Variables in the OBL are typeless; a single variable instance can hold a value that is of any of the supported data types. See section 7.4.2 later in this section for a discussion of the types of data supported in the OBL. We could have just as easily assigned a string value instead of a numeric value:

```
init {
        xangle;
        tilt = "Hello!";
}
```

Along with initializations with literals, we can invoke any of the built-in functions that are provided by the OBL (section 7.6.2 provides a complete list of these functions):

```
init {
        trigger = random(1,10);
        rotate(50,0,-30);
}
```

7.4.1.1 Temporary Variables

Occasionally, you might find the need to use a variable in the "initialize" section of an OBL source file for holding temporary values. These declared variables will never be used by the "evaluate" section, and would simply continue to exist, consuming memory.

You can avoid this situation by using temporary variables. These variables are only valid in the "initialize" section of the OBL source file, and are identified by a dollar sign ($) as their first character. These variables may be used as you would any other variable, yet they will no longer exist once the "initialize" section has been evaluated.

In the following example, only the variable `newAngle` will survive to be accessible by the "evaluate" section:

```
init {
        $temp1 = random(1,10);
        $temp2 = $temp1 * sin(45);
        newAngle = $temp2;
}
```

7.4.2 Object Behavior Language Data Types

The OBL supports three different types of data. These types are discussed in the next three sections.

7.4.2.1 number

Numeric values in the OBL are represented as floating point. In C, this type would be `double`. You can write numbers in your OBL source file in integer format, but all numeric values are converted to floating point with double precision. Numbers can be negative or positive. OBL also supports scientific notation.

```
init {
      integer = 15;
      real = 0.3;
      negative = -35.356;
      scientific = 7.5E+13;
}
```

7.4.2.2 string

String literals are represented by enclosing characters in double quote marks at the beginning and end of the character sequence:

```
init {
      astring = "Here is a test string";
}
```

You can embed literal quote marks within a string value by escaping the quote character with a back slash (\):

```
init {
      astring = "I think \"Phil\" was his name.";
}
```

7.4.2.3 array

OBL supports arrays of single dimensions. Array elements can be either numeric or string. Each array can contain any mixture of these data types.

Arrays can be created in two different fashions. The first format simply declares the array, indicating the number of elements the array contains:

```
init {
      check[15];
}
```

You can also declare and initialize an array from entries in a disk file. This disk file should contain one array element per line, and should be in ASCII format. The OBL interpreter will attempt to determine the data type as it reads in each entry.

Initialization of an array from a disk file takes place when you use a string literal value in place of the numeric parameter that would indicate the size of the array:

```
init {
      // the following initializer is formatted for a
      // UNIX file system

      points["/home/rbh/points.txt"];

      // an OS/2 system would look like
      // points["\home\rbh\points.txt"]

      index = 1;
}
```

With the `debug` command active, you can see each array element as it is read and converted.

Once arrays are declared, individual elements can be accessed by referencing the array element directly:

```
eval {
      x = points[index];
      y = points[index + 1];
      z = points[index + 2];

      index += 3;
}
```

Array elements are accessed in the OBL starting at offset one (1), as in Pascal, instead of zero (0), as in C or C++. Each OBL source file can declare a maximum of ten (10) arrays.

7.5 Inherited Objects

Each POSE object you create inherits a number of other objects. These inherited objects can only be accessed from within an OBL source file. As such, to take advantage of these objects, you must assign and create an OBL source file for your POSE object.

These inherited objects give you access to attributes and settings for your POSE object, such as texture, pigment, and finish. Each inherited object provides attributes as well as methods.

Some attributes and methods in the following sections may not contain descriptions detailed enough to allow you to begin using them immediately. For more detailed information on an attribute or method, please refer to the section of the POV-Ray documentation that further describes the object.

7.5.1 texture

The `texture` object provides access to POV-Ray texture attributes. The attributes available through this object are:

`map`	A character string attribute that contains a valid POV-Ray texture name

This object also provides methods that can be invoked:

`scale(#,#,#)`	A method to scale the texture map of the object
`rotate(#,#,#)`	A method to rotate the texture map of the object

A sample usage of the texture object might be:

```
init {
      ...
      texture.map = "Chrome_Metal";
      texture.scale(1,1,2.5);   // increase z axis of texture
      ...
}
```

7.5.2 pigment

The `pigment` object provides access to POV-Ray pigment settings for the current object. The attributes used to access these settings are:

`userType`	A string that specifies a predefined pigment setting (this setting should have been created using a POV-Ray `#declare` directive). This attribute differs from the color attribute only in that no "color" keyword is used to preface the value in the POV-Ray input file.
`color`	A string containing a valid POV-Ray color name
`quickColor`	A string containing a valid POV-Ray color name
`gradient`	A string value indicating the axis along which the gradient will be applied (i.e., "x", "y", "z")
`red`	A numeric value between 0 and 255
`green`	A numeric value between 0 and 255
`blue`	A numeric value between 0 and 255
`quickRed`	A numeric value between 0 and 255
`quickGreen`	A numeric value between 0 and 255

`quickBlue`	A numeric value between 0 and 255
`turbulence_x`	A numeric value indicating the amount of turbulence along the x axis
`turbulence_y`	A numeric value indicating the amount of turbulence along the y axis
`turbulence_z`	A numeric value indicating the amount of turbulence along the z axis
`agate_turbulence`	A numeric value
`omega`	A numeric value
`lambda`	A numeric value
`octaves`	A numeric value
`frequency`	A numeric value
`phase`	A numeric value
`radial`	A boolean value (true or false)
`bozo`	A boolean value (true or false)
`filter`	A numeric value

Methods for the pigment object consist of:

`scale(#,#,#)`	A method used to scale the pigment settings
`rotate(#,#,#)`	A method used to rotate the pigment settings
`checker(S,S)`	This method selects a checker pattern for the object's pigment. It takes two color names.
`hexagon(S,S,S)`	This method selects a hexagonal pattern for the object's pigment. It takes three color names.

A sample usage of the pigment object might be:

```
init {
      ...
      pigment.checker("Blue","Grey");
      ...
}
```

7.5.3 normal

The `normal` object provides access to POV-Ray normal settings for the current object. The attributes used to access these settings are:

`bumps`	A numeric value
`dents`	A numeric value
`ripples`	A numeric value
`waves`	A numeric value
`wrinkles`	A numeric value
`frequency`	A numeric value

`phase`	A numeric value
`turbulence_x`	A numeric value
`turbulence_y`	A numeric value
`turbulence_z`	A numeric value

In addition, these attributes of the `normal` object are provided to support Bump Maps:

`file`	A string value indicating the file to use as the bump map
`file_type`	A string value indicating the type of file, one of "tga," "gif," "iff," or "dump"
`bump_size`	A numeric value other than zero (0)
`map_type`	A numeric value, one of 0, 1, 2, or 5
`interpolate`	A numeric value of either 2 or 4
`use_index`	A boolean value (true or false)
`use_color`	A boolean value (true or false)
`once`	A boolean value (true or false)

The methods used to access these settings are:

`scale(#,#,#)`	A method to scale the normal settings of the current object
`rotate(#,#,#)`	A method to rotate the normal settings of the current object

An example usage of the `normal` object might be:

```
init {
      ...
      normal.waves = 0.05;
      normal.frequency = 5000;
      normal.scale(300,1000,300);
      ...
      step = 0.05;
      normal.phase = -0.05;
      ...
}

eval {
      // cycle the waves in the ocean

      normal.phase += step;

      if(normal.phase == 1.0)
            step = -0.05;

      if(normal.phase == 0.0)
            step = 0.05;
}
```

7.5.4 finish

The `finish` object provides access to POV-Ray finish settings for the current object. The attributes used to access these settings are:

`crand`	A numeric value
`phong`	A numeric value
`phong_size`	A numeric value
`diffuse`	A numeric value
`brilliance`	A numeric value
`ambient`	A numeric value
`reflection`	A numeric value
`specular`	A numeric value
`roughness`	A numeric value
`refraction`	A numeric value
`ior`	A numeric value
`shiny`	A boolean value (true or false)
`metallic`	A boolean value (true or false)

Methods for modifying `finish` are:

`scale(#,#,#)`	A method to scale finish settings
`rotate(#,#,#)`	A method to rotate finish settings

An example usage of the `finish` object might be:

```
init {
      ...
      finish.phong = .8;
      ...
}
```

7.5.5 image

The image object allows you to access image-specific POV-Ray settings for your object. Attributes of this object that help you are:

`once`	A boolean value (true or false)
`map_type`	A numeric value
`interpolate`	A numeric value of either 2 or 4
`file`	A string value indicating an image file to use
`file_type`	A string value indicating the type of `file`, one of "tga", "gif", "iff", or "dump"

Methods for the `image` object consist of:

`scale(#,#,#)`	A method to scale the image settings
`rotate(#,#,#)`	A method to rotate the image settings
`filter(#,#)`	A method to set the filter values of the image, where the first numeric value is the palette number and the second is the transparency value

You might use the `image` object like this:

```
init {
      ...
      image.file = "plasma.gif";
      image.file_type  = "gif";
      image.map_type = 0;
      image.filter(0,0.5);  // make color 0 50% transparent
      ...
}
```

7.6 Built-In Features

Along with inheriting other objects, each OBL source file can access "built-in" features and functions as well. These built-ins consist of both variables and functions.

7.6.1 Variables

A number of variables can be access from within each OBJ source file. These variables fall into two classes: global, belonging to the POSE environment; and local, belonging to the POSE object.

7.6.1.1 Global variables

7.6.1.1.1 frame

The `frame` variable holds the number of the current frame of animation being generated by POSE. This value can be accessed directly, and contains a numeric value.

A sample use of this variable might look like:

```
eval {
      ...
      if(frame > 200)
      {
            ...
      }
      ...
}
```

7.6.1.1.2 no_shadow

This variable is a global that most POSE objects inherit. However, if you examine the listings in Appendix B, you'll notice that not all objects export this variable for modification. The reason for this is that there are some POSE objects where the use of `no_shadow` would make not sense. This is typically restricted to the POSE Light objects. CSG constructs where you might include a light source can utilitize the `no_shadow` effect.

The `no_shadow` variable is a boolean, accepting either a true or false setting:

```
initialize {
      ...
      no_shadow = true;
      ...
}
```

7.6.1.1.3 inverse

As with `no_shadow`, the `inverse` variable is also a global inherited by all POSE objects. However, only those objects wherein the use of the inverse function of POV-Ray would make sense will export it for use.

`inverse` is also a boolean variable, requiring a setting of either true or false:

```
initialize {
      ...
      inverse = true;
      ...
}
```

7.6.1.2 Local object variables

Each object type that you create within POSE has a set of variables that it exports to an OBL source file. It is typically a good idea to assign to a POSE object an OBL source file that has been created specifically for an object of that type. Assigning an OBL file to an object for which it was not designed will likely cause a number of errors when you attempt to access local object variables that have not been exported by the POSE object.

Appendix B documents the variables that are exported by each POSE object type. In the same fashion as the `frame` variable discussed previously, these variables can be accessed directly from within an OBL source file without having to declare them.

You do not have to worry about having this document handy when you are editing an OBL source file. If POSE creates the OBL source file when you issue and `edit` command, POSE will instruct the particular object type to document its exported variables, and these descriptions (virtually identical to

those presented in Appendix B) will be embedded in the OBL source file for reference during programming.

7.6.2 Functions

POSE makes available to the object programmer a number of built-in functions. These functions are accessed from the OBL source file, and both their calling syntax and usage are documented in the following sections.

7.6.2.1 sqrt()

This function calculates the square root of a number. It accepts a numeric value, and returns the square root of that value.

You might invoke this function in the following manner:

```
init
{
      ...
      angle = 180;    // 180 degrees
      varSquare = sqrt(angle);  // calc square root of angle
      ...
}
```

7.6.2.2 exp()

This function calculates the exponent of a value. It accepts a numeric value, and returns the exponent of that value.

An example of its usage might be:

```
init
{
      ...
      v1 = exp(27.5);
      ...
}
```

7.6.2.3 log()

This function calculates the logarithmic value of a number. It accepts a number, and returns the logarithmic value of that number

An example of its usage might be:

```
init
{
      ...
      v1 = log(347.495);
```

```
        ...
}
```

7.6.2.4 sin()/asin()

These two functions calculate the sine and arc sine, respectively, of a number. They accept a numeric value, and return the sine or arc sine of that value.

An example of their usage might be:

```
init
{
        ...
        sin1 = sin(45);
        ...
        asin1 = asin(276);
        ...
}
```

7.6.2.5 cos()/acos()

These two functions calculate the cosine and arc cosine, respectively, of a number. They accept a numeric value, and return the cosine or arc cosine of that value.

An example of their usage might be:

```
init
{
        ...
        cos1 = cos(879);
        ...
        acos1 = acos(15);
        ...
}
```

7.6.2.6 tan()/atan()

These two functions calculate the tangent and arc tangent, respectively, of a number. They accept a numeric value, and return the tangent or arc tangent of that value.

An example of their usage might be:

```
init
{
        ...
        tan1 = tan(665);
        ...
        atan1 = atan(45);
        ...
```

```
}
```

7.6.2.7 hide()/show()

This pair of functions control whether or not an object includes its POV-Ray information in one or more frame files. All POSE objects default to the "show" state when they are created. They have to be explicitly hidden through the use of the `hide()` function.

It is important to note that, even though an object has removed itself from the animation by a call to `hide()` function, it continues to be evaluated by POSE through each frame generation. In this fashion, the object can continue to "live," updating its variables and state, even though it is not appearing in any scene.

Both functions accept no parameters, nor do they return any values. An example of their usage might be:

```
init {
      showing = true;
}

eval {
      ...
      if(showing == true && frame == 100)
      {
            hide();
            showing = false;
            return;
      }
      ...
      if(showing == false && frame == 350)
      {
            show();
            showing = true;
      }
}
```

7.6.2.8 disable()/enable()

This pair of functions control whether or not an object is evaluated as POSE generates each frame of animation. All POSE objects default to the "enable" state when they are created. Evaluation of an objects behavior has to be explicitly disabled through a call to the `disable()` function.

It is important to note that, even though an object has disabled itself by using the `disable()` function, its POV-Ray information continues to be included in the frame files generated by POSE. If the object was within view of the camera when it disabled itself, it will appear to the viewer as though the object simply froze.

Further, you should be aware of the ramifications of calling `disable()` function from within an OBL source file. Once disabled, an object cannot re-enable itself; its behavior is no longer evaluated, so the opportunity will never arise within the object itself. Only another POSE object with behavior can re-activate a disabled object by invoking that objects `enable()` method. See section 7.7 later in this document for a discussion of POSE objects accessing one another during frame generation.

Both functions accept no parameters, nor do they return any values. An example of their usage might be:

```
eval
{
      ...
      if(frame == 100)
      {
            disable();
            return;
      }
}
```

7.6.2.9 random()

This function offers an OBL source file the ability to generate random numbers. It accepts two numeric values, and returns a random number that lies between the two numbers (inclusive).

An example of its usage might be:

```
init
{
      ...
      triggerFrame = random(20,50);
      ...
}
```

7.6.2.10 rotate()

This function allows an object to rotate itself in three-dimensional space. It accepts three numeric values, each one representing a degree along the X, Y, or Z axis, respectively.

If you do not intend to modify the rotation factor along any particular axis, you should place a value of zero (0) in that position. A value of zero causes the object to maintain its original rotation relative to that axis.

Rotations are not queued. In other words, no matter how many `rotate()` commands you issue in a single execution of the OBL program, only one rotate

statement will be issued for the object.

An example of its usage might be:

```
init
{
      ...
      zangle = 0;
      ...
}

eval {
      ...
      rotate(0,0,zangle);
      zangle += 10;
      ...
}
```

7.6.2.11 scale()

This function allows an object to scale itself in three-dimensional space. It accepts three numeric values, each one representing a scaling factor to effect the X, Y, or Z axis, respectively.

If you do not intend to modify the scaling factor along any particular axis, you should place a value of one (1) in that position. A value of one causes the object to retain its shape in that direction without modification.

Scales are not queued. In other words, no matter how many `scale()` commands you issue in a single execution of the OBL program, only one scale statement will be issued for the object.

An example of its usage might be:

```
init
{
      ...
      xscale = 1;
      scaleStep = .25;
      ...
}

eval {
      ...
      scale(xscale,1,1);
      xscale += scaleStep;
      if(xscale > 2)
            scaleStep = -.25;
      if(xscale < -2)
            scaleStep = .25;
      ...
}
```

7.6.2.12 sound()

One of the most interesting functions POSE provides is the `sound()` function. This function affords every POSE object with a behavior the ability to synchronize a sound file with its behavior in the current frame. Of course, in order to take advantage of this capability, you will need an animation viewer that will read the resulting sound file and its format.

This function accepts two parameters: a string and a number. The string parameter is a path/file name to the sound file that is to be played on behalf of the object when the current frame is displayed by the viewer. The second numeric parameter represents the sampling rate at which the sound file is to be played back.

Each time an object generates a sound effect using the `sound()` function, a file in the project's frame directory is updated. The name of this file is "scene#.snd", where the numeric value (#) represents the scene number. This file contains a series of single entries, in the following format:

<frame_number>: <sound_file>(sampling_rate)

An example usage might be:

```
eval {
      ...
      if(y == 0)  // we've hit bottom
            sound("splat.wav",8000);  // play splat at this frame
      ...
}
```

POSE doesn't care what format your sound file is in; it simply stores the name and sampling rate where another process can read it. As such, you can specify a sound file of any format to POSE. It will be your animation playback engine that will have to deal with the actual format of the sound file.

7.7 Accessing Other POSE Objects

Mentioned in section 4.2.1 was the fact that you could bestow names upon your POSE objects. That section made a vague reference to the practical use of such a name, and in this section, we define exactly what that means.

In the Object Behavior Language of POSE, one object can access the attributes and, in some cases, methods of other POSE objects. In order for this mechanism to function between two objects, the object being referenced must have an identifier (the object making the reference is under no such requirement itself). Giving a POSE object a name satisfies this requirement,

allowing other POSE objects to access it directly.

For example, let's say that one POSE object, a Sphere, needed to be aware of the whereabouts of another POSE object, also a Sphere, in the three-dimensional space of the animation. Let's further assume that the first object knows the second object (the one it must access) by the name "Harry" (case is important). If we glance at Appendix B, we can determine that "Harry," because he is a Sphere, exports the variables "x", "y", "z", and "radius." If the first object wanted to check on the location of "Harry" in the current frame, it would look something like this:

```
eval {
      ...
      if(Harry.x > 0 && Harry.z < -2)
      {
            ...
      }
      ...
}
```

Further, section 7.6.2.11 tells us that every POSE object inherits the function scale(), and as such, it becomes a public method of that object, accessible to the world. The first object would then be able to further access Harry in the following way:

```
eval {
      ...
      if(Harry.x > 0 && Harry.z < -2)
      {
            Harry.scale(1,1,4);
      }
      ...
}
```

Here's another example of this access mechanism. Let's assume that Harry has invoked the `disable()` method on himself. He is now sitting there like a log, simply existing. The first object, perhaps after detecting a collision between itself and Harry, might flip Harry's "on" switch and bring him back to the living:

```
eval {
      ...
      if(<some test on Harry>)
            Harry.enable();
      ...
}
```

POSE objects have methods and attributes, some are "public" (exported variables and inherited methods) and some are "private" (declared variables). You must observe this design when programming POSE objects to interact.

8.0 ***Postscript***

As I mentioned at the beginning of this document, I wrote POSE out of frustration. When I started development a little more than a year ago as of this writing, I wanted to produce computer animation, not spend all my time writing the tools. Well, on that score, I lost.

However, there were a number of other goals I was driven to attain with the writing of POSE, and, to be fair, the animation system offered me a "playground" where I could experiment with and attain them. For one, I designed and implemented a script language and its necessary interpreter using lex, yacc, Alan Holub's Compiler Design in C and the Dragon Book. For another, I did heavy OO development with POSE; it is largely driven by objects. Granted, graphics programming lends itself nicely to object-oriented development, and POV-Ray had already modelled the objects themselves, but I did bring them to "life."

POSE in and of itself is not a "this-is-all-you-need" solution. It is merely another tool, another utility among many. It has no facilities for creating spline-based paths. It cannot create or allow you to view GIF images. You can't use it to create POV-Ray objects out of Postscript fonts. What I've attempted to do with POSE is create an environment where the output of other high-quality utilities written by other authors can be merged together with the objects provided by POV-Ray. It is within this environment that I hope the aspiring computer animator (myself included) will not be limited by the balance of a checkbook.

Granted, you can't do dancing soup cans with POSE, but you can do computer animation, and for the computer-animator-on-a-budget, it does enough to satisfy that animation craving.

I'd really love to see the animations you can come up with using POSE. Drop me a line and let me what you're doing.

Virtually Yours,

Bob Hood
May 1st, 1995

Appendix A **Contacting The Author**

- US Mail

Bob Hood
1217 South Independence Street
Lakewood, CO 80232 USA

- Internet

bhood@netcom.com

- Reporting bugs and making suggestions

I'm always interested in hearing about potential problems with POSE, or suggestions concerning enhancements and modifications to the POSE environment. However, because I am only a single person, I cannot always guarantee a response to every contact I receive. I will only promise to do my best.

Whether you are contacting me to report a bug or suggest a change, it would help me out greatly if you would provide specific examples of where you think a bug is occurring, or where a modification/enhancement should take place. This could include anything from screen captures to OBL source code examples. Completed animations showing changes or errors are really not necessary, unless you have not provided me with sufficient information/data to create my own.

If you plan to send me US Mail concerning any topic, please add the line "ATTN: POSE" somewhere on the outside of the envelope. If you plan to send me E-Mail (highly preferred), please ensure that the name "POSE" appears somewhere in the subject line of the message, i.e.:

```
Subject:  POSE: Fantastic program!   But...
```

While I will not send corrected versions of POSE to specific individuals, I will try to make frequent patch releases at the sites specified in section 1.3 that will incorporate the bug fixes. Enhancements and modifications will be queued for subsequent release in the next major revision of the software.

Appendix B **Exported Object Variables**

- Camera

```
x ............ --
y ............   |-- Vector to center of Camera location
z ............ --
lookat_x ..... --
lookat_y .....   |-- Vector to look_at of Camera
lookat_z ..... --
up_x ......... --
up_y .........   |-- Vector to up of Camera
up_z ......... --
direction_x .. --
direction_y ..   |-- Vector to direction of Camera
direction_z .. --
sky_x ........ --
sky_y ........   |-- Vector to sky of Camera
sky_z ........ --
right_x ...... --
right_y ......   |-- Vector to right of Camera
right_z ...... --
```

- Point Light

```
x ......... --
y .........   |-- Vector to center of Point Light
z ......... --
color ..... A valid POV-Ray color name string
red ....... Numeric value indicating intensity of red color (0-255)
green ..... Numeric value indicating intensity of green color (0-255)
blue ...... Numeric value indicating intensity of blue color (0-255)
```

- Area Light

```
x ......... --
y .........   |-- Vector to center of Area Light
z ......... --
length_x .. --
length_y ..   |-- Vector of Area Light length
length_z .. --
dist_x .... --
dist_y ....   |-- Vector of Area Light direction
dist_z .... --
n1 ........ Numeric indicating first dimension size of point light array
n2 ........ Numeric indicating second dimension size of point light array
adaptive .. Numeric that sets adaptive sampling value of light source
jitter .... A boolean (true or false) for random jittering to eliminate any shadow banding
color ..... A valid POV-Ray color name string
red ....... Numeric value indicating intensity of red color (0-255)
green ..... Numeric value indicating intensity of green color (0-255)
blue ...... Numeric value indicating intensity of blue color (0-255)
-- optional Spot Light settings --
pointAt_x . --
pointAt_y .   |-- Vector to point_at of optional Spot Light settings
pointAt_z . --
radius .... Numeric indicating degrees of the bright circular hotspot
            at the center of the Spot Light's area of affect
falloff ... Numeric indicating degrees of falloff angle of the radius
            of the total Spot Light area
tightness . Numeric indicating the speed with which the light dims in
            the region between the radius cone and the falloff cone (1-100)
```

- Spot Light

```
center_x .. --
center_y ..   |-- Vector to center of Spot Light
center_z .. --
```

```
pointat_x . --
pointat_y .   |-- Vector to point_at of Spot Light
pointat_z . --
color ..... A valid POV-Ray color name string
red ....... Numeric value indicating intensity of red color (0-255)
green ..... Numeric value indicating intensity of green color (0-255)
blue ...... Numeric value indicating intensity of blue color (0-255)
radius .... Numeric indicating degrees of the bright circular hotspot
            at the center of the Spot Light's area of affect
falloff ... Numeric indicating degrees of falloff angle of the radius
            of the total Spot Light area
tightness . Numeric indicating the speed with which the light dims in
            the region between the radius cone and the falloff cone (1-100)
```

- Sphere

```
x ......... --
y .........   |-- Vector to center of Sphere
z ......... --
radius .... Numeric size of Sphere as half the diameter
no_shadow . A boolean (true or false) indicating whether this object
            casts a shadow in the scene
inverse ... A boolean (true or false) indicating whether this object
            should be inverted
```

- Blob

```
x ............ --
y ............   |-- Vector to center of Blob
z ............ --
threshold .... Numeric indicating the total density value (> 0)
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Blob Component

```
x ......... --
y .........   |-- Vector to center of Blob Component
z ......... --
strength .. Numeric value indicating field strength at component center
radius .... Numeric size of component as half the diameter of the sphere
```

- Box

```
x1 ........ --
y1 ........   |-- Vector to coordinate of lower-left corner of Box
z1 ........ --
x2 ........ --
y2 ........   |-- Vector to coordinate of upper-right corner of Box
z2 ........ --
no_shadow . A boolean (true or false) indicating whether this object
            casts a shadow in the scene
inverse ... A boolean (true or false) indicating whether this object
            should be inverted
```

- Cylinder

```
x1 ........... --
y1 ...........   |-- Vector to center of first end of Cylinder
z1 ........... --
x2 ........... --
y2 ...........   |-- Vector to center of second end of Cylinder
z2 ........... --
radius ....... Numeric size of Cylinder as half the diameter
open ......... A boolean (true or false) indicating whether the Cylinder has open ends
```

```
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Cone

```
x1 ........... --
y1 ...........   |-- Vector to center of first end of Cone
z1 ........... --
radius1 ...... Numeric size of first end as half the diameter
x2 ........... --
y2 ...........   |-- Vector to center of second end of Cone
z2 ........... --
radius2 ...... Numeric size of second end as half the diameter
open ......... A boolean (true or false) indicating whether the Cone has open ends
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Plane

```
x ............ --
y ............   |-- Vector to normal of Plane (only one should be set)
z ............ --
offset ....... Numeric distance of displacement from the normal from origin
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Character

```
x ............ --
y ............   |-- Vector to center of Character
z ............ --
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Torus

```
x ............. --
y .............   |-- Vector to center of Torus
z ............. --
major_radius .. Numeric indicating major radius of Torus
minor_radius .. Numeric indicating minor radius of Torus
sturm ......... A boolean (true or false) used to select Sturmian root
                solving to render object
no_shadow ..... A boolean (true or false) indicating whether this object
                casts a shadow in the scene
inverse ....... A boolean (true or false) indicating whether this object
                should be inverted
```

- Fog

```
distance ..... Numeric distance for 100% fog color
color ........ A valid POV-Ray color name string
```

- Disc

```
x ............ --
y ............   |-- Vector to center of Disc
z ............ --
normal_x ..... --
normal_y .....   |-- Vector to orientation of Disc
```

```
normal_z ..... --
radius ....... Numeric size of Disc as half the diameter
hole ......... Numeric to indicate radius of Disc hole
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Bicubic Patch

```
x ......... --
y .........   |-- Vector to center of Bicubic Patch
z ......... --
type ...... Numeric indicating patch type (must be 0 or 1)
flatness .. Numeric controlling the amount of "splitting" that takes place
u_steps ... Numeric indicating minimum rows of triangles to use
v_steps ... Numeric indicating minimum columns of triangles to use
no_shadow . A boolean (true or false) indicating whether this object
            casts a shadow in the scene
inverse ... A boolean (true or false) indicating whether this object
            should be inverted
```

- Union

```
x ............ --
y ............   |-- Vector to center of Union
z ............ --
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Difference

```
x ............ --
y ............   |-- Vector to center of Difference
z ............ --
no_shadow .... A boolean (true or false) indicating whether this object
               casts a shadow in the scene
inverse ...... A boolean (true or false) indicating whether this object
               should be inverted
```

- Intersection

```
x ......... --
y .........   |-- Vector to center of Intersection
z ......... --
no_shadow . A boolean (true or false) indicating whether this object
            casts a shadow in the scene
inverse ... A boolean (true or false) indicating whether this object
            should be inverted
```

- Composite

```
x ......... --
y .........   |-- Vector to center of Composite
z ......... --
no_shadow . A boolean (true or false) indicating whether this object
            casts a shadow in the scene
inverse ... A boolean (true or false) indicating whether this object
            should be inverted
```

Appendix C POSE Registration Form

POSE v1.0
Copyright (C) 1995 Bob Hood

R E G I S T R A T I O N I N S T R U C T I O N S

This copyrighted material may be distributed freely for evaluation. It is not, nor has it ever been, free. The decision to keep this software after an evaluation period of 14 days implies a decision on your part to buy a registered copy of POSE.

Benefits of registering POSE:

- Full functionality of POSE (no limitiations on objects, primitives, or scenes).
- Updates to the current release of POSE will be provided to registered users free of charge.
- New releases of POSE are offered to registered users at reduced rates.
- Registered users receive priority in requests for improvements and enhancements.

How to register POSE 1.0

A single-user license for POSE, regardless of platform, is US$30.
To register, send a US$ check (personal/company) or money order payable to "Bob Hood" with the completed registration form (found below) to the following address:

Bob Hood
ATTN: POSE Registration
1217 South Independence Street
Lakewood, CO 80232 U.S.A.

COLORAOD RESIDENCES MUST INCLUDE STATE SALES TAX AT 3.8% PER COPY OF POSE REGISTERED. Your registration will be returned to you without processing if this amount is not included.

After your registration has been processed, you will receive, either by E-Mail or US mail, instructions and required information to upgrade your Shareware copy of POSE 1.0 to the full, unlimited registered version.

Please, when mailing your registration form and payment, be aware of the following:

- Do not send cash through the mail.
- Only personal/company checks and money orders are acceptable payment for registration. COD and credit card orders cannot be accepted.
- Personal/company checks may take up to ten (10) business days to clear.
- The price of registration is subject to change without prior notice. If the price has changed since this writing, you will be notified of the difference at the same time you receive your registration information.
- Unfortunately, support for the UNIX platforms on which POSE is currently available is not guaranteed in future releases.
- GUI versions of POSE (i.e., Microsoft Windows, OS/2 Presentation Manager versions, Motif) are currently only being considered. Their release is NOT guaranteed (I would like to do *some* animation work, you know =|^).
- If you're *really* that interested, site licensing is available for POSE. Contact the author to arrange a pricing schedule.

POSE Registration Form

Name: _____

Company Name: _____ (if applicable)

Street Address: _____

City: _____ State: _____ Zip: _____

Country: _____ (if outside the United States)

Phone: _____ FAX: _____ (if applicable)

E-Mail Address: _____ (if applicable)

A registration number is required for all registrations. You may select your own registration number, or one will be provided for you. A validation key will be generated from the registration number assigned to your copy of POSE, and provided to you.

If you choose to select your own registration number, you should enter only numeric characters, between 0 and 9. All nine digits are required to generate the validation key; any that are left empty will be filled in for you when your registration is processed.

Examples of good numbers to use might be your Social Security number, or perhaps the first nine digits of your telephone number (including area code).

NOTE: Validation keys are unique to each platform. For example, you cannot use a validation key generated for OS/2 on the HP-UX version of POSE.

Nine-digit registration code: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]

	Version	Platform	Price		QTY		Total
POSE	1.0	32-bit MS-DOS	$30.00	x	_____	=	$_____
POSE	1.0	OS/2 2.x/Warp	$30.00	x	_____	=	$_____
POSE	1.0	HP-UX 9.x	$30.00	x	_____	=	$_____
CO State Sales Tax @ 3.8% per copy				x	_____	=	$_____
					Total		$_____

! ! ! I M P O R T A N T ! ! !

By providing your signature below, you acknowledge and agree to the terms and conditions set forth both in this document and in the file LICENSE.TXT provided with the POSE distribution. Registration for the full version of POSE can only be processed with your signature as an indication of acceptance.

Signature: _____ Date: _____

Please, take a moment to complete the following questions:

Where did you acquire your Shareware copy of POSE?

[ ] On-line service (Compu$erve, AOL, etc.) _____

[ ] From a friend

[ ] BBS _____ Phone # ( ) _____

[ ] Internet site _____

Would you like to see versions of POSE for other ray tracing systems (i.e., Vivid, Rayshade, Polyray)?

[ ] Yes [ ] No [ ] Don't Care

Would you like to see a GUI-base version of POSE (i.e., Windows, PM, Motif)?

[ ] Yes [ ] No [ ] Don't Care

Software can always be made better. Please write your comments or suggestions concerning POSE in the space below. Thanks for registering POSE!