6

Chapter Six
FUNDAMENTAL USE OF OBJECTS

What is an Object?

An Object is a dynamic element that can be manipulated on the screen. An Object might represent a vehicle, person, or machine.

What Can You Do with Objects?

You can do many things with an Object, including:

- Create it
- Destroy it
- Place it at a coordinate point in a layout
- Change its color
- Change its shape
- Move it along a predefined Path
- Move it from one point to another
- Change its speed
- Rotate it

In this chapter, we will cover four basic animation trace commands for Objects: create, place...at, destroy and set color.

What is an Object Class?

When you create an Object, you must specify its *Object Class* (also called a *Class*). An Object Class determines a variety of characteristics of Objects created from it, and thus can be regarded as a "cookie cutter" used to create Objects. The most obvious characteristics that an Object gets from its Object Class are the shape and initial colors of the Object. Other characteristics an Object initially derives from its Class include various settings related to guided motion on Proof Paths. Guided motion is introduced in Chapter 7.

Proof Animation is not "Object-oriented" software! Please do not be misled by Proof's "Object" and "Class" terminology into thinking otherwise.

What are the advantages of having Object Classes? Consider the animation of a freeway interchange shown below. Several thousand cars might pass by in an hour. Each car is an Object. You don't want to define the shape of a car several thousand times. The shape and other properties of identical cars need to be defined only once, in an Object Class definition. Then, when you need to create a new car in the animation trace file, you refer to the class definition by name. Object Class definitions are saved in the layout file.

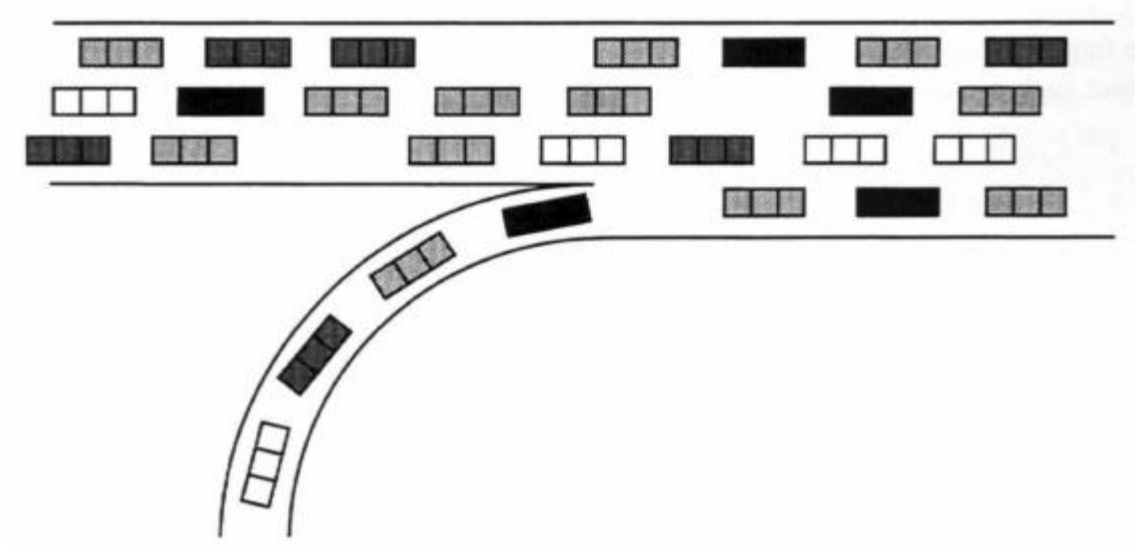

Figure 6-1. Many identical Objects (cars) are defined based on one Object Class.

We are not going to concern ourselves at this time with how an Object Class is defined. (Class definition is covered in Chapter 11.) Instead, we will provide examples that refer to a predefined layout file called shapes.lay. This file contains definitions for a filled circle-shape Object Class named CIRCLE, a filled square-shape Object Class named SQUARE, and a filled triangle-shape Object Class named TRIANGLE.

Rules for Object Names, Class Names, and Other Names

Many things in Proof are identified by names that you define. Object Classes always have names. Paths, Messages, Plots, and Bars also have names, all of which are covered in later chapters.

For Object names, Object Class names, and most other identifier names, there are some simple rules. Upper and lower case letters are treated as unique characters in identifier names. Each name must start with an alphabetic character (A-Z or a-z). Names can be one to sixteen characters long. After the first letter, the remaining characters in a name can be letters, digits or the underscore character. Instead of a name, Objects (and only Objects) can be identified by a unique integer number.

Because Proof treats letters of different case as distinct in identifier names, you need to be careful how you reference names. "Square," "square," and "SQUARE" would be treated as three different names.

It's important to distinguish identifier names, which are case sensitive, from commands and command keywords, which are not case sensitive. Commands and command keywords can be upper or lower case, or a mixture. "TIME," "Time," and "time" are all the same command.

Creating an Object

Before you can display an Object on the screen, you must first create it. In the animation trace stream, the create command does this.

The syntax of the create command is:

create *classname ObjectID*

Here are some examples of the create command:

create WIDGET 6

create Comm_Node TTY2

create EmptyAGV 6578

create Digit1 101

create TYPE3SUB7 422

create CAR Herbie

For the animation to work, the *classname* must be the name of an Object Class that is stored in the layout file. An *ObjectID* is either an Object number or an Object name. Individual Objects must be referenced by a unique number or name. The number or name must be unique across all Object Classes. For example, there cannot be a CAR 1 and a BUS 1 existing simultaneously in the animation.

As mentioned previously, Object names as well as Class names *are* case sensitive. For example, “Herbie” and “HERBIE” would refer to two distinct Objects.

Often, a screen Object will be associated with a particular moving unit of traffic in a simulation model. In this case, each *ObjectID* should be a number, or possibly a name that contains a number. This number can be stored as an “attribute” or “parameter” of the unit of traffic in the model, which makes it easy for the model to write out animation commands that affect a specific Object.

Even after an Object has been created, it will not automatically appear on the screen. Where would it be placed? Would it be moving? Another command is needed before a created Object can appear.

Placing an Object on the Screen

Proof has a few different animation trace commands that can cause an Object to appear on the screen. The most fundamental of these is place...at.

The syntax of place...at is

place *ObjectID* **at** *x y*

place...at causes the Object to appear at a particular point in the Proof coordinate system, but this command does not cause the Object to move.

Here are some examples of the place...at command:

place 6578 at 40 40

place MACHINE at 12.5 25

place 4 at 10 -10

place Car at -5.02 35

You can combine examples of create and place...at into a single, hand-edited animation trace file that you can use to see the results.

Exercise 6-1: Creating and Placing an Object

From the Windows **Start** menu, start any software that can save a document as a text file (such as NotePad or WordPad, which come with Windows and can be found under **Programs**, **Accessories** in the **Start** menu; or another text editor of your choosing; or a word processing program such as Microsoft Word). You should then see a blank new document window, but if not, begin a new file (**File**, **New**).

Note: Notepad defaults to saving the document as a "text document." However, as mentioned in Chapter 5, if you are using WordPad or any word processing program, you must be sure to save the shapes.atf file (when the time comes to save it) as "text document" or "text only" so that formatting codes don't corrupt the file.

You specify the type using the "Files of Type" option that normally appears below the file name line in the Save dialog.

Type in some trace commands that create some filled circles, squares, and triangles and place them at various locations on the screen. Then type in **time 100** and **end** as the last two lines.

Save the file (**File**, **Save** or **File**, **Save As…**) as shapes.atf. Be sure to save it as a text file, and save it in the same folder where shapes.lay is located, which is in the exercise folder.

Remember that in Proof, names are case sensitive. In shapes.lay, the Object Class names are capitalized (SQUARE, CIRCLE, and TRIANGLE). So, make sure that you've used all caps for these Object Class names in your animation trace file.

Your shapes.atf file should look something like the listing below.

```
create SQUARE 1
create TRIANGLE 2
place 1 at 20 20
place 2 at 30 30
time 100
end
```

After saving shapes.atf, start Proof (or if Proof is already started, switch to Proof using the tab on the Windows taskbar at the bottom of your screen). Choose **File, Open Layout & Trace** and, select shapes.lay. (If you don't see this file name it means you put your shapes.atf file somewhere else. You can open the layout and trace files one at a time, or you can move shapes.atf to the correct place.) Then go to **File, Open Trace (.ATF)**, and in the dialog box under "Files of type:" set the type at "All Files (*.*)." Click on the "running man" icon on the toolbar, and your objects should appear. Observe that after Objects appear the animation clock runs until time 100 is reached, as indicated in shapes.atf.

Removing Objects No Longer Needed

The destroy command is used to remove from the screen any Object that is no longer needed. The Object is erased from the screen (if necessary) and removed from Proof's internal data storage. The destroy command has no effect on the underlying Object Class.

The syntax for destroy is

destroy ObjectID

For example:

```
destroy Car
```

```
destroy 28
```

Remember the discussion in Chapter 5 about simultaneous trace commands. If you insert

destroy commands immediately after the place...at commands in shapes.atf (see Exercise 6-1), you will never see the Objects because no time elapses between the time you place the Objects and destroy them.

You should always destroy an Object at the point in the animation trace stream after which you will have no further use for that Object.

Exercise 6-2: Destroying Objects

Modify the shapes animation by inserting some destroy commands into the shapes.atf file you previously created in Exercise 6-1. (If you did not create shapes.atf, you can copy a completed version from the sample folder into the exercise folder.) Remember to let some time elapse between creating and destroying a given Object.

What happens if you try to place an Object that has been destroyed? After saving this change into your shapes.atf file, re-opening the layout and trace files in Proof, and beginning to run the animation, you should receive an error message telling you have have tried to place an Object that does not exist.

Exercise 6-3: Moving an Object the Hard Way

Now that you have the time, create, place at, and destroy commands at your disposal, you can move Objects across the screen.

Edit the shapes.atf file you created in Exercise 6-1 with your text editor or word processor. Modify the trace stream so that it creates one Object and places that Object at evenly spaced places at evenly spaced times. (For example: time 10, place 1 at 20 20, time 20, place 1 at 30 20, time 30, place 1 at 40 20, etc.) The idea is to cause a single Object to move across the screen by "visiting" several coordinate points at discrete time intervals, as illustrated in Figure 6-2.

From Proof, choose **File, Open Layout+Trace**, select shapes.lay, and re-run the animation. Try running your animation at different viewing speeds. How does this affect the motion?

This is not how you will move Objects in your real animations! Proof provides other, more powerful ways to move Objects than the method used above. These other methods, which are much easier to use and which automatically result in smooth motion, are introduced beginning in Chapter 7.

However, it pays to keep in mind that Proof's low-level commands can be adapted to special purposes. Moving Objects "the hard way" using place...at illustrates such an adaptation.

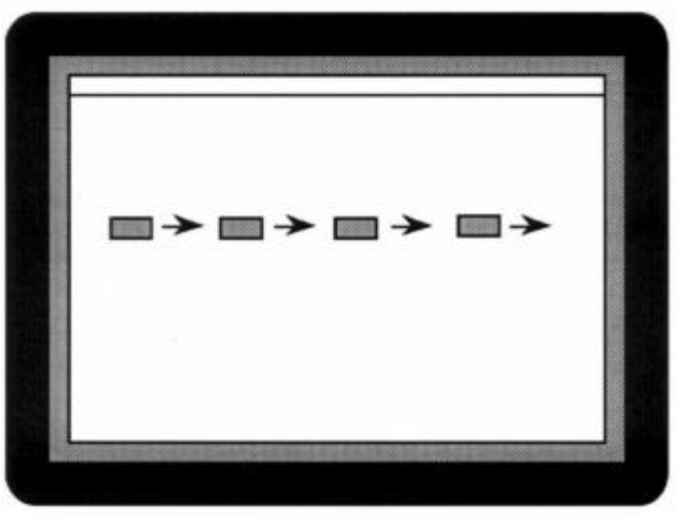

Figure 6-2. A single Object moves across the screen "the hard way"

Setting and Changing an Object's Color

You use the set color command to change the color of an Object that has already been created. If the Object is already visible on the screen, its color will change as soon as Proof processes the set color command in the animation trace file.

The syntax for set color is

set *ObjectID* **color** { *color* | **class }**

For example,

```
Set 3 color class
Set 1 color F1
Set truck234 color yellow
```

The available colors in Proof are listed below:

Background Colors		Foreground Colors	
BACKDROP	**Bac**	BLUE	**F1**
LAYOUT (Gray)	**L1**	RED	**F2**
		WHITE	**F3**
Additional	**L2**	YELLOW	**F4**
layout	.	PINK	**F5**
colors	.	TAN	**F6**
	.	GREEN	**F7**
	L32		
		Additional	**F8**
		foreground	.
		colors	.
			.
			F32

To specify a color, use the corresponding F-number, L-number, or name. Since only the first seven foreground colors have alternate names, you will probably find it easier to specify colors using the F-number. Backdrop should be specified by Bac or Backdrop.

If you set a multi-colored Object to say, F5, the entire Object will become the single color, F5.

The class keyword will cause the Object to change the default color or colors of the Object Class of which it is a member. This is the only way to restore multiple colors to an Object.

You'll get the chance to use set color in an exercise in Chapter 8.

Background vs. Foreground Colors

When working with Objects, you should consider your animation as containing precisely three layers. The two back layers (or, the background) reside behind the front layer, or (the foreground). Of the background colors, Backdrop (which defaults to black) appears behind the Layout colors, and Layout colors appear behind Foreground colors. See Figure 6-3.The foreground colors are opaque over the background colors. Therefore, Objects of any of the foreground colors can move without visible interference "across" other Objects or drawn

elements that use the background colors (Layout and Backdrop). Layout-colored Objects can move across the Backdrop color, or *under* any foreground color, without interference.

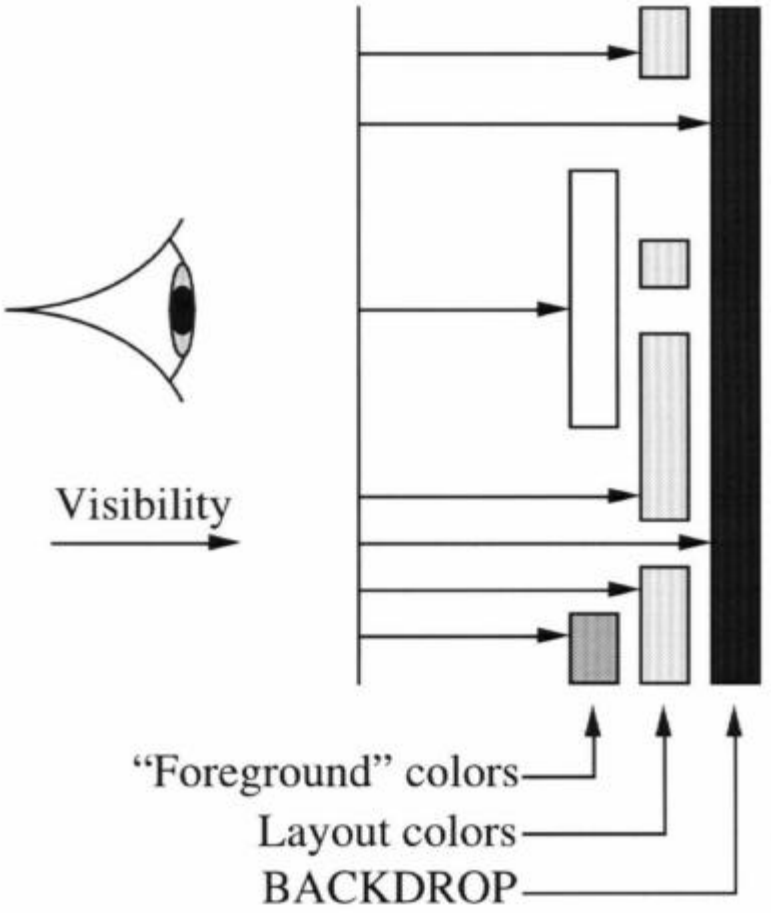

Figure 6-3. How colors appear

Preferences vary as to what color the Backdrop color should be. Version 4 of Proof uses a beige color as the initial Backdrop color for a new layout. Some people might prefer black or white or another shade. Older versions of Proof used black as the default Backdrop color and many of the exercises are saved with a black backdrop. It is easy to modify the Backdrop color or any other color for a given layout. This is described in Chapter 15.

Object Interference and Cancellation

If an Object moves or is placed over another Object or layout element, the color of the overlapping area will be the highest priority color. Foreground colors have higher priority than layout colors, and layout colors have higher priority than the backdrop color. Within the same color classification, higher numbered colors have precedence over lower numbered colors. For example, F5 takes precedence over F4.

To see an informative example of color priorities, you can run the animation contained in palettex.lay *and* palettex.atf *in the* features *folder. (In reality, Proof's algorithms for color rendering are more complex than as described above; however, if you understand the basic concepts described above, you normally won't have to worry about the details. A detailed description of Proof's rendering algorithms is given in Appendix D.)*

Review: How Animation Trace Commands are Processed

Let's re-examine which trace stream commands are executed at various times. Suppose your shapes.atf (in Exercise 6-1 or 6-2) looked like the following:

```
create SQUARE 1
place 1 at 20 20
time 10
create CIRCLE 2
place 2 at 30 30
create TRIANGLE 3
place 3 at 40 20
time 25
destroy 2
time 30
create TRIANGLE 2
place 2 at 50 40
time 40
destroy 1
time 45
destroy 3
destroy 2
time 55
end
```

Note that Object number 2 is recycled after the first Object 2 is destroyed.

The processing of the commands in this sample shapes.atf would occur at the times indicated in Figure 6-4.

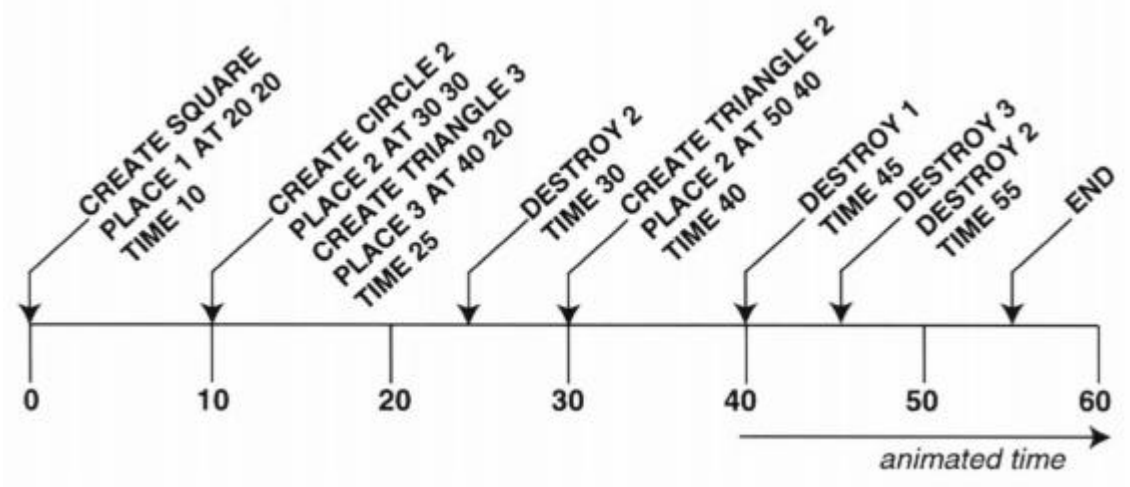

Figure 6-4. Processing of commands in the animation trace file

For each time command that is processed, the animation trace file is telling Proof that the commands that follow occur at the specified time. Thus, those commands are not processed until that time has arrived. The first create and place...at commands are processed at time 0. Because there is no preceding time command, Proof assumes that these are "time = 0" events. Proof then reads the time 10 command and begins animating, but the part of Proof that reads the trace file waits until the animated/simulated time reaches 10 before reading and processing the create CIRCLE 2 command. This cycle repeats for each time command encountered.

Another way to think of this is illustrated by slightly annotating the trace stream as follows. The gray characters are not part of the trace command syntax, but they help to explain what Proof is doing. (The indentation is not part of the syntax either but if it were present the animation would still run.)

```
at time 0:
        create SQUARE 1
        place 1 at 20 20
at time 10:
        create CIRCLE 2
        place 2 at 30 30
        create TRIANGLE 3
        place 3 at 40 20
at time 25:
        destroy 2
at time 30:
        create TRIANGLE 2
        place 2 at 50 40
at time 40:
        destroy 1
at time 45:
        destroy 3
        destroy 2
at time 55:
        end
```