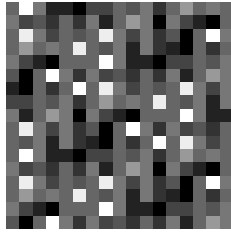


Taking Advantage of PostScript

John F. Sherman



Taking Advantage of PostScript

John F. Sherman

University of Notre Dame

Art, Art History & Design
University of Notre Dame
Notre Dame, IN 46556 5639

FRPEXX@IRISHMVS.cc.nd.edu



intro

Designers face a new requirement: to acquire and master a digital craft. The traditional crafts of color theory, design, typography and drawing have always been required for the designer. All of these are important and I do not advocate their dismissal. But the need for a digital craft is becoming more apparent as the innovations of technology arrive faster and faster.

Designers, however, cannot focus on simply being well-trained users of expensive pieces of hardware and software. By mastering the technology, they can reach a point where innovative new solutions to visual communication problems can be discovered.

The process by which an image can be made has changed dramatically. An image can be constructed by a combination of hand drawings, video capture, and computer programming. The finished image provides a solution by means of faster investigation, greater choices, and new creative possibilities.

Digital craft entails learning a new language — a visual language. The greater the depth of understanding and experience in a language, the greater the vision of what can exist in the mind of the creator. Different languages allow different realities to exist. Many of the images created for this book are visual ideas that have been made into images by writing them down. They can only be achieved by an intimate knowledge of the PostScript language. I find it exciting that there are images that can only be created by writing a PostScript program. Understanding PostScript is one major avenue to mastering the technology of producing graphics.

1.1 *why learn PostScript ?*

Why learn PostScript when there are so many good graphics programs available that are much easier to use than programming? There are two answers.

First, knowing the PostScript Page Description Language (PDL) gives the designer an insight into graphics software based on the PostScript image mode. The PostScript PDL has strengths and weaknesses. Having this knowledge base builds confidence because it permits you to work with the strengths of software and not with false expectations. When something does not work as expected, you may be able to devise a way to work around or attack the problem from another direction.

Second, the PostScript language is a richer graphics language than what is made available by menu selections and dialog boxes in all the graphics software packages available. There are visual opportunities available to you that are only available by programming. A design you write in PostScript might supplement a packaged technique in a software program or be written totally in PostScript code.

Either way, you are in control of the design process.

1.2 *PostScript's background*

PostScript is a page description language developed by Adobe Systems Incorporated. PostScript resides within a printer and acts as an interpreter for the data sent to it by either a software program or an original program written by a programmer. A page description language is the means through which a printer prepares a page containing text, line art, and digitized images. The page is constructed pixel by pixel (dot by dot). The size of dot will depend on the printer, and ranges from 300 to 2540 dots per inch (dpi). A 300 dpi printer will draw a 1 inch square 300 dots to a side, while a 2540 dpi printer will draw the square 2540 dots per side. Figures 1-1 and 1-2 are the same shape; they have the same description. The dotted line is the true shape. What will be different is the quality of its presentation through either printing or a monitor's display. The greater the resolution provided, the closer to the true description the shape will appear.

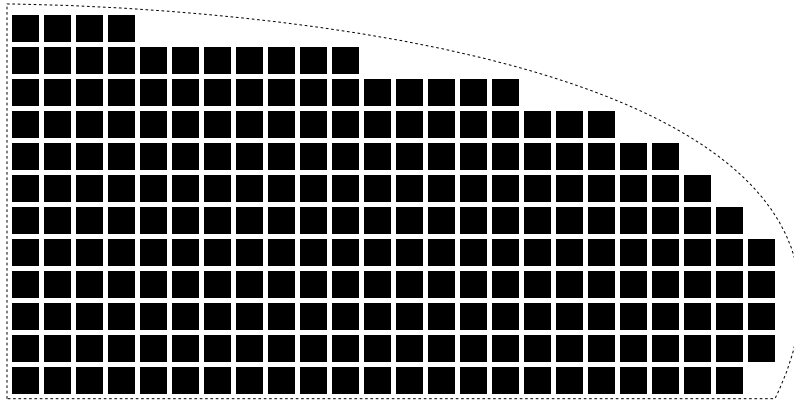


figure 1-1

Figure 1-2 is the same shape with double the resolution.

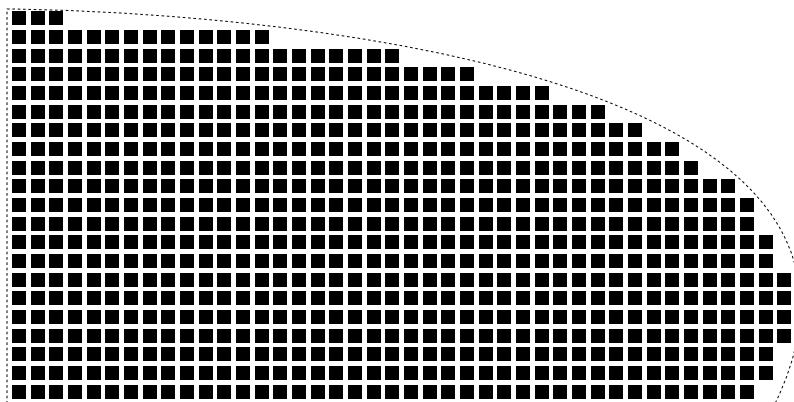


figure 1-2

The most widely known example of a 300 dpi printer is the Apple LaserWriter. The Linotron L100 has a maximum resolution of 1270 dpi and can be set to print at 600 and 300 dpi. The Linotron L300 is also available with a maximum printing resolution of 2540 dpi. The advantage of using the increased resolutions of these printers is that a greater number of visually distinct grays can be produced. The Linotron L100 can produce about one hundred visually separate gray levels because of its fine dot pattern.

The PostScript language is printer independent, enabling you to alternate from one brand of printer to another without making any changes to the original document. All the major original equipment manufacturers of high resolution printers have licensed PostScript for their printers making it the de facto industry standard. The designer can use a low cost laser printer for sketching and then later use the more costly high resolution imagesetters when the design is completed. The difference in the resolution changes the rendering quality and the number of distinct grays that can be depicted.

Until recently, a laser printer equipped with a PostScript interpreter was the only means to see the designs created by writing PostScript programs. Display PostScript is an adaptation of the PostScript PDL from Adobe Systems Inc. for use on computer monitors, and the NeXT computer is the only computer that uses it. Display PostScript is to the graphic display of computer monitors as the PostScript PDL is to different laser printers. It provides a common imaging model for the graphic display (screen description language) and the laser printer. The Macintosh uses the QuickDraw language for its screen display and a file's data is converted from QuickDraw to PostScript for printing. Because of this, the full richness of the PostScript language cannot be used and then viewed on the Macintosh screen.

Display PostScript is monitor independent and automatically takes advantage of the full capabilities (gray levels or color) of the computer's monitor without having to rewrite the part of the software responsible for the graphic display. Display PostScript has several advantages. Most importantly, what is seen on the screen very closely matches what is printed. The only difference is the resolution of the image on the monitor compared to the resolution of the image ultimately printed on the laser printer. For example, in drawing a 1 inch square, the square would be drawn with ninety-two pixels per side (resolution of the NeXT monitor) and the same information would be used again and sent to the laser printer to create the square at 300 dpi or more.

1.3 *organization of this book*

Taking Advantage of PostScript is divided into four major parts.

Chapters 2–7 of this book will introduce some basic PostScript concepts and drawing techniques. Chapter 4 will show how these simple programs can be saved as EPS files to be used within other programs. Most of the simple beginning programs are more easily drawn in a variety of graphics software programs, but they are the building blocks to the more complicated programming techniques that come in later chapters.

Chapters 8–17 will cover advanced PostScript techniques that unleash the power of the PostScript Page Description Language. Much of what is covered here cannot be accomplished in available graphics software. Chapter 17 covers some advanced programming techniques and takes several designs step by step and explains how and why they were written as they are. Chapter 18 will touch on some of the new features of PostScript Level 2.

Chapters 19 and 20 of this book are libraries of numerous examples of PostScript programs that can be the starting point for new designs. They generally concentrate on one visual idea or programming technique. Chapter 21 is a gallery of images.

The final section of the book contains several reference appendixes.

1.4 *formats used in this book*

The main narrative of this book is in the font you are now reading.

All PostScript program listings or the mention of a PostScript operator will be set in Courier Bold and look like this:

```
PostScript code    % comment
```

Notes or comments (explained in more detail in section 2.5) will be set in regular Courier to help set them off from the program. At times, when more lengthy annotation is required, I'll bracket the explanation like this for better legibility and to save space:

```
% -----  
Extended comments; not part of the program.  
% -----
```



1-1
animation

When you see this symbol in the margin, it means that the PostScript program is available on disk for experimentation within the companion *LearnPS* tutorial. There will be a version for both Macintosh and NeXT computers. The *LearnPS* symbol may also indicate that the PostScript program for an illustration is available or that an animation demonstrating a particular point is available. Use the number or title below the symbol to help you locate the file in *LearnPS*. Most of the PostScript examples in this book are written to appear on a grid representing the bottom left corner of a page.

A PostScript program will be listed below the graphic it produces. In this way, it will serve as a title for the graphic. Otherwise, a graphic title will be found in the left margin.

Each chapter is divided into sections. We are now in section 1.4, meaning chapter 1 section 4. Often in this book I'll refer to other sections using this system.

1.5 *getting started*

Writing a simple original PostScript program is a fairly easy task to accomplish. You most likely already have the tools needed if you have a Macintosh computer and PostScript equipped printer. The program file can be written with any word processing program that can save standard *ASCII* text files. An *ASCII* text file is a standard form of saving text that all computers understand. It cannot, however, contain specialized formatting that most applications provide, such as variable point sizes, font styles, or graphics.

On the Macintosh you will need:

Software to write the program. Microsoft Word, MacWrite, WriteNow and others will work fine. You may find it convenient to use a text editor designed for working only with text files.

A utility to send the file to the PostScript printer. The most common is *SendPS* from Adobe.

A PostScript equipped printer, to interpret your files.

On the NeXT:

The NeXT computer comes with all the software you'll need to send the file to

the printer or monitor.

Chapter 2 covers getting started in more detail.

1.6 *options for downloading files*

There are quite a few utilities to send or download a PostScript file to a printer. On the Macintosh I have used these, but there are many others:

LaserStatus DA from CE Software

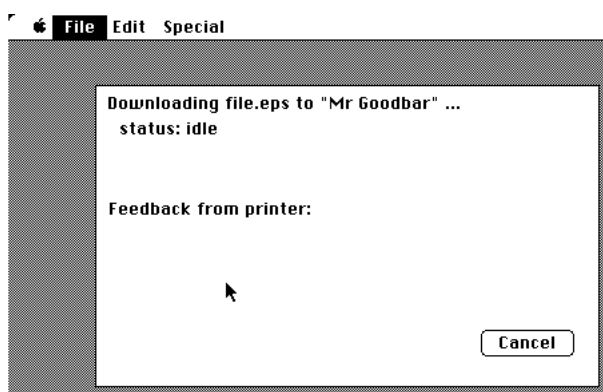
SendPS from Adobe Systems

Downloader from Adobe Systems

I have two favorite ways of working on the Macintosh. One option is to write the file using *QUED* (Q**U**ality **E**ditor) from Paracom. *QUED* is a text editor; it only opens and saves files as text. I then send the PostScript file to the printer with *LaserStatus*. Since *LaserStatus* is a Desk Accessory, I can easily switch back and forth between editor and downloader while developing a design. It looks like this below:



The second method is again to use *QUED* to write the program, but send the file to the printer using either Adobe's *SendPS* or *Downloader*. The primary difference between the two utilities is where the standard output file is directed. This file may be an error message or feedback from the printer. With *SendPS*, the error messages and feedback are sent to a file on your computer's disk. This is handy since at times the message can flash by quickly or you may need the information that is sent back. The advantage of *Downloader* is that those messages are sent to a window on the Macintosh screen. The standard output file will come up a number times later in section 16.1 and the utilities appendix.



You will be able to send PostScript files to your printer from within *LearnPS*.



overview of the basics

This section is designed to cover the key points and terms used in PostScript programming. Even if PostScript is your first attempt at writing a computer program, you should find PostScript is not nearly as intimidating to learn as other programming languages. I believe the reason for this is that the goal is to get an image on paper, which is much more interesting than calculating some interest rate or personnel data base. Plus, the “Hello World” output, the classic first goal in other programming tutorials, can now be in a variety of fonts and placed anywhere on the page.

2.1 *coordinate system*

The page is a coordinate system based on an x and y axis. The origin, the 0 0, is usually the bottom left corner of the page. We’ll see later that this can be changed at any time.

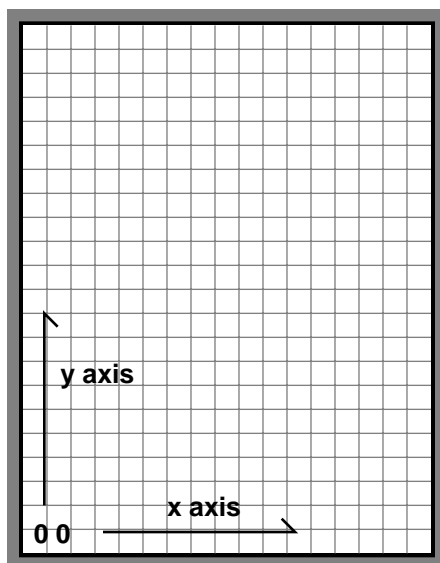


figure 2-1

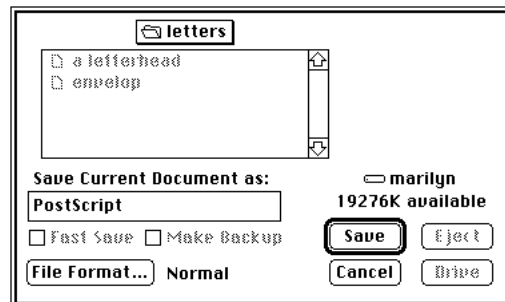
Distance is measured in points with 72 points equaling an inch. Each square above in figure 2-1 represents 36 points. This is basically the same unit of measurement used by typesetters and printers in the graphic arts industry. Traditionally 72 points equals 0.918 of an inch. In PostScript, however, 72 points equals 1 inch exactly. In some situations this difference can cause a great deal of trouble. If it is important that you match the traditional measurement scale, the coordinate

system can be changed so that 72 points does equal 0.918 of an inch. This can be done with the `scale` operator (see chapter 10).

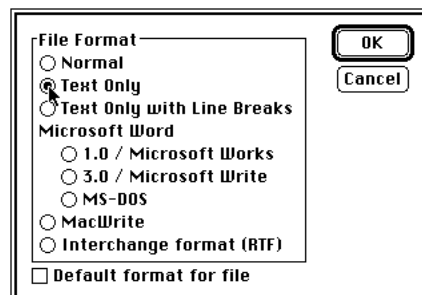
2.2 *the PostScript program must be a text file*

The PostScript program must be a text file, sometimes known as an ASCII file. A text file can be written by almost every kind of word processor. A text file is an agreed on standard format of saving words that all computers can accept and understand. Because it is a convenient means to exchange information between various computers, it cannot contain specialized formatting that is handled differently from one word processing application to another. A text file cannot contain variable font styles, point sizes, or any graphics. It can contain carriage returns, and tabs are at fixed, regular intervals.

If you are using Microsoft Word on the Macintosh, a text file would be saved by first choosing “Save” or “Save As ...” (if already saved) from the “File” menu. Notice the “File Format ...” button at the lower left corner. Choose it.



A second dialog box will appear.



Choose the “Text Only” option and then “OK.” The file will be saved as text.

MacWrite and other programs have similar options. You may try a text editor that only saves and works with text files. Their advantage is they usually come with features geared for easier programming.

2.3 *PostScript is case sensitive*

The PostScript language is case sensitive. Case sensitive means uppercase characters are seen as different than lowercase characters. That is, “word” is seen as a different word than “WORD” or “Word.” In some other programming

languages this is not the case.

2.4 *the program format*

A PostScript program can be formatted in any way convenient for legibility using tabs, returns, or extra spaces between words. These are seen as white space and do not affect the running of the program. For example,

```
gsave
    100 100 translate
grestore
```

is the same as

```
gsave 100 100 translate grestore
```

which is the same as

```
gsave
100 100
translate
grestore
```

2.5 *inserting comments*

Titles and reference comments can be included in a PostScript program by entering them after the *special character* “%” (percent sign). Everything following a “%” and before the next “return” or line end is invisible to the PostScript interpreter. The “%” is also used for header information (the first lines of a program) of a file that is written in a standard way. This will be explained further in chapter 4, “EPS Files.” The space between the code and the “%” can either be tabs or any number of word spaces.

It’s a good habit to include comments in your programs for future reference. A month from now, you won’t remember why you did what you did. A few moments of your time now will pay off later when you need the information.

Examples of inserting comments are

```
% move origin one inch
gsave
    72 72 translate
grestore
```

The comment can also be inserted in this way,

```
gsave          % move origin one inch
    72 72 translate
grestore
```

or in this way

```
gsave
    72 72 translate   % move origin one inch
grestore
```

2.6 *the graphic state*

Many of the commands in PostScript describe some kind of graphic operation or change in the graphic state. The graphic state is the set of givens or the environment a graphic is drawn in. For example, it would be safe to assume the following:

The measurement system equals 72 points to the inch.

The origin is at the lower left corner of the page.

The default color is black.

The default line weight is 1 point.

There is no default font.

These are some of the default values that define the graphic state that you begin with. There is a possibility that a previous PostScript program may have changed all this, but normally these defaults are in effect. If you draw a 72 point filled square, it will be filled with black and be 1 inch square unless defined otherwise.

There are several more default values to be aware of, but these are enough to get started with. The important concept here is that as a PostScript program is interpreted by a printer, current values for a number of things (color, line weight, etc.) can be changed, and persist until another change. These changes are changes in the graphic state.

2.7 *the current point & current path*

Graphics and type are placed on the page by establishing a current point. The current point is the current position of a drawing action. Establishing a current point can be thought of as the initial act of positioning a pencil on paper to begin a drawing.

The `moveto` operator creates the location for a character or the starting point for the drawing of a graphic. This is measured relative to the origin or 0 0 (zero zero) of the coordinate system. For example,

```
144 216 moveto
```

establishes a current point 2 inches to the right and 3 inches up from the location of the origin. There is nothing that will print yet, it is only the starting point or location. More on `moveto` later in section 2.9.

If a line is drawn from this current point located at 144 216,

```
144 216 moveto
288 288 lineto
```

a current path is created from the point 144 216 to the point 288 288. Again, nothing will print yet because it is only a path. The path will need to be painted for something to print. The current path can be painted as a stroke to give it the values of the current line weight and color, or the path may be filled with the current color. As a path is created, the current point is the most recent point on the path. Once

the path is painted, the path is cleared or initialized. These are important points to remember. A typical sequence of events could be:

- 1 No current point or current path
- 2 Create current point
- 3 Create current path
- 4 **stroke** or **fill** the current path
- 5 Current path is cleared

2.8 *the operand stack*

PostScript is a *stack* based language. An *operand* stack is like a plate dispenser at a cafeteria. The first plate in is the last plate out and the last plate in is first plate out. In PostScript, the stack is an area of memory that values, *operators*, and other items are *pushed* onto. The first item in a line of PostScript code will be the first item on the stack and the last to be used. For example, `moveto` will need two numbers already on the stack. In the case of `144 216 moveto`, `144` is on the bottom and `216` is next on top. `moveto`, the operator, then comes along on top of the two numbers and removes them to go off and make a current point. The lines of code

```
144 216 moveto
288 288 lineto
```

could be seen entering the stack as in figure 2-2 below.



stack demo

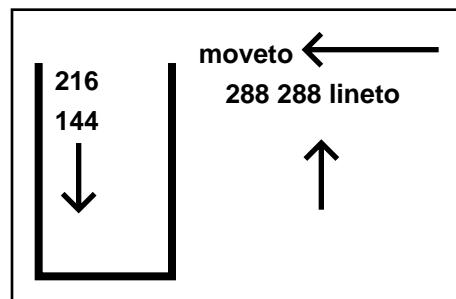


figure 2-2

Another way of looking at

```
144 216 moveto
288 288 lineto
```

would be to chart the stack as it changes. In figure 2-3, each white rectangle represents the stack as values are pushed onto it and removed by operators.

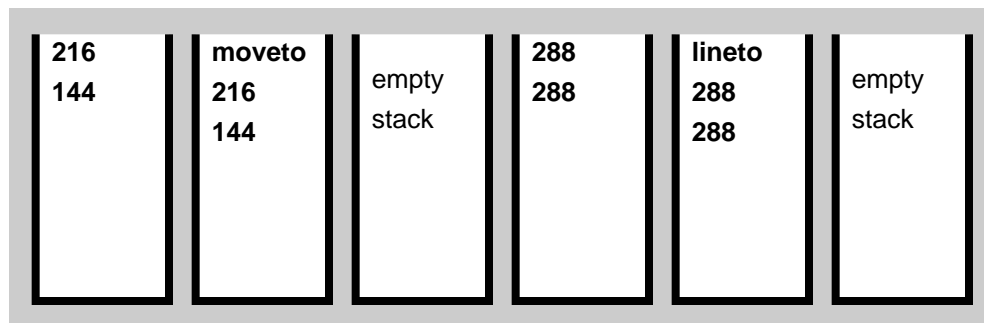


figure 2-3

The program could also be written like this:

```
288 288 144 216 moveto lineto
```

It would produce the same results and the chart would look like figure 2-4.

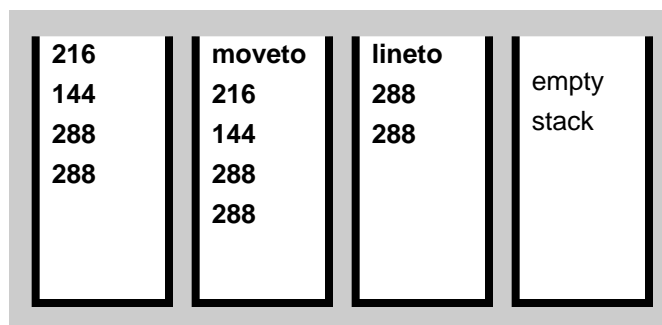


figure 2-4

This is not as convenient a way to write a program, but it works. It can be easy to lose track of which values go with what operator.

There are a number of PostScript operators that help manipulate the order of the stack. `exch`, for example, switches the top two items on the stack. More on this in later chapters.

2.9 *creating a current point*

As stated in section 2.7, `moveto` is used to create a current point and expects two numbers on the stack for its `x` and `y` location on the page. The syntax for `moveto` is

```
x y moveto
```

The `x` and `y` can be any two numbers, positive or negative. For example, the following can be used:

```
72 72 moveto
207.45 34.17 moveto
-34 17 moveto
```

Later we'll learn to use *variables* so we can write

```
over up moveto
x1 y2 moveto
```



```
1 inch 2 inch moveto
```

or even

```
x 12 add y 12 add moveto
```

The `moveto` operator expects to find two numbers waiting for it on the stack. How they arrive there is of no concern to it.

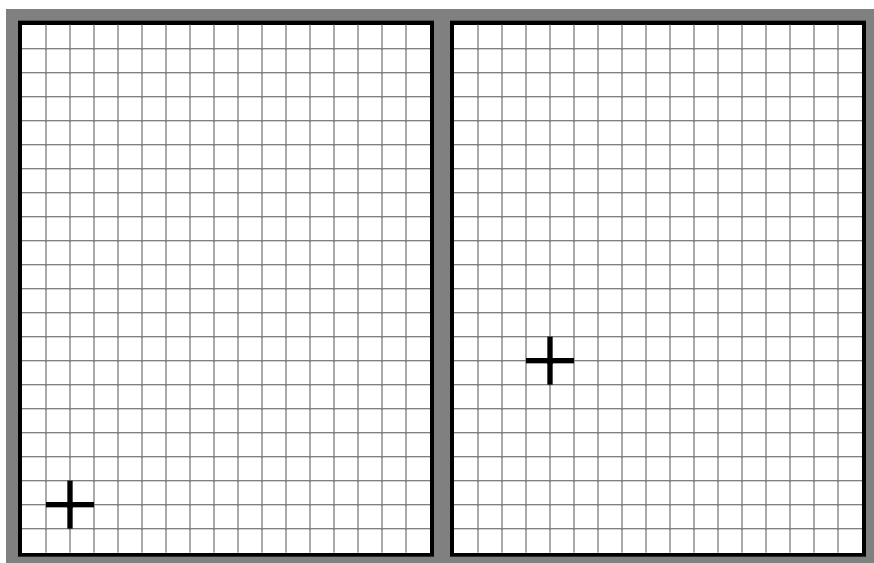


figure 2-5

The +’s in the graphic in figure 2-5 mark the location of the current points at 72 72 and 144 288 with 72 72 `moveto` and 216 288 `moveto`.

A variation of `moveto` is `rmoveto` or *relative moveto*. `rmoveto` is used to make a break in the current path as it is being constructed. An example of its use will be found at the end section 2.11.

2.10 *creating a current path*

To create a current path, `moveto` is used to make a starting or current point. Then one of the PostScript path construction operators such as `lineto` is used to identify a new point extending from that point.

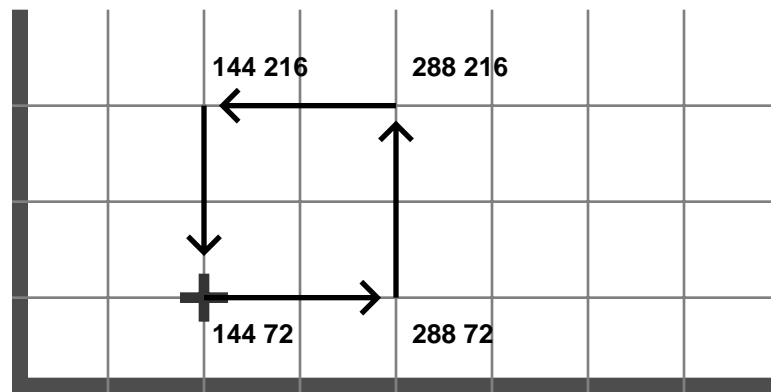


figure 2-6

To draw a 1 inch square on the page, we would first need to establish a current point and then a current path. See figure 2-6. The path could start at 0.5 inch up and over 1 inch from the bottom left corner of the page and move right 1 inch, up 1 inch, left 1 inch, and back down to the starting point. The path could also be drawn in the clockwise direction as well. The initial current could have been any one of the corners.

This description of the square is the current path; nothing has been painted yet. It is the road map for painting that comes later.

The `lineto` operator creates a current path to some new point from an existing current point. The new point becomes the new current point. The syntax for the `lineto` operator is:

```
x y lineto
```

where `x` and `y` are coordinates on the page. Also, `lineto` requires a current point from either a previous `moveto` or `lineto`. The PostScript program of our square up to this point would look like this:

```
144 72 moveto
288 72 lineto 288 216 lineto 144 216 lineto 144 72 lineto
```

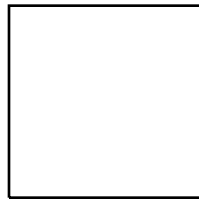
This PostScript program fragment creates a current path. If we were to print at this moment, we would have a blank page. The next step of the program is to use the path either to stroke a line or as a container for a fill. It is also possible to use the path as a boundary or cookie cutter for images to appear within. This is done with `clip`, and will be explained in section 9.4.

2.11 *painting the current path*

Once a current path has been constructed, you have the choice of either stroking it or filling it with the current color. The two operators are `stroke` and `fill`. These two operators paint the current path with the current line weight and current color. In addition, the `stroke` and `fill` operators consume the current path, meaning once the path is painted, it will no longer exist. Examples of a stroked square and a filled square with their PostScript programs follow.



2-1



```
72 72 moveto
144 72 lineto 144 144 lineto 72 144 lineto 72 72 lineto
stroke
```



2-2

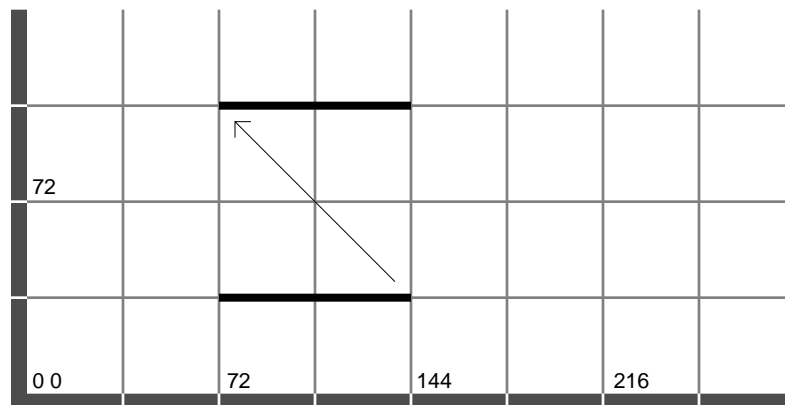


```
72 72 moveto
144 72 lineto 144 144 lineto 72 144 lineto 72 72 lineto
fill
```

As mentioned in section 1.4, if you want to experiment and print these first two programs, find examples 2-1 and 2-2 in the ExamplePS section of *LearnPS*.

Remember, the reason these examples look like this is because the default value for the current color is black and the current line weight is one point. Those values will remain until you change them. In the next chapter, we'll run this square through all sorts of variations.

The current path does not need to be one continuous connected path. In this example, `rmoveto` is used twice to continue the path after a break. The arrow in the graphic shows the movement of the current point with `rmoveto`. The point of the arrow is also drawn using `rmoveto`.



2-3

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:twoLines.eps
%%BoundingBox:72 34 144 110
```

```
3 setlinewidth           % parallel lines
72 36 moveto 144 36 lineto
-72 72 rmoveto 144 108 lineto % second line
stroke

.25 setlinewidth        % arrow
138 42 moveto -60 60 rlineto
0 -6 rmoveto 0 6 rlineto 6 0 rlineto stroke
```




drawing a square

This chapter covers a number of different methods that can be used to draw a square. In doing so, various strategies are covered in a small scale that can be applied to a variety of programming situations. The PostScript language provides more than one way to create the same result. The reason to use one over another will depend on the situation and your writing style.

3.1 *finishing the final corner of the square*

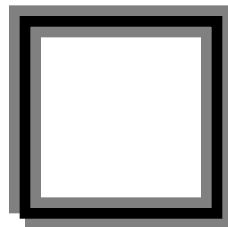
In figure 3-1 below, the black line represents the path for the wide gray line. Notice the line does not extend beyond the path. Because of this, a square stroked with a thick outline would have a noticeable notch at the final corner.

figure 3-1



The preferred way to write the PostScript program for the square drawn in the last chapter would be to include the `closepath` operator. `closepath` joins the current point on the path to the first point made by `moveto` and finishes that corner. Other examples of its use will be seen in chapter 6.

This example paints one square on top of another. The black square on top uses the `closepath` operator to finish the corner at the location of the path's beginning and ending. The PostScript program is listed below.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:closepath_1.eps
%%BoundingBox:0 0 84 84

.5 setgray 12 setlinewidth

6 6 moveto          % gray thick line

```



3-1

```
72 0 rlineto 0 72 rlineto -72 0 rlineto 0 -72 rlineto
stroke
```

```
% black square on top
0 setgray 4 setlinewidth
```

```
6 6 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
closepath stroke
```

This PostScript program introduces several new operators.

First, an explanation of the first three lines called the program header. These three lines are not necessary to create the actual graphic, but they are very important if the graphic will be placed in another document. This will be explained in more detail in the next chapter on EPS files.

The new PostScript operators introduced are:

<code>setgray</code>	changes the current gray value.
<code>setlinewidth</code>	changes the current line weight.
<code>rlineto</code>	draws a line relative to the previous current point.

`setgray` expects a number between 0 and 1 where 0 is black, 1 is white, and various grays are in between. Therefore, 0.2 is 80% black, 0.5 is 50%, and 0.9 is 10% black. `setgray` changes the current color used by `stroke` and `fill` for painting.

`setlinewidth` changes the current line weight used by `stroke` and expects a positive number. It can be 0 and as high as 700 or higher. However, it is not advisable to use zero. `0 setlinewidth` paints the line as fine as the printer is capable of producing. On a 300 dpi that would be 0.24 inches, which can be seen easily. But on a 2540 dpi Linotron, however, it would be 0.000393 inches — not visible.

`rlineto` is an alternative to `lineto`. `lineto` draws lines to points on the coordinate system relative to the origin. `rlineto` draws its lines relative to the previous current point.

We have two ways to draw the path for a square. There is

```
72 72 moveto
144 72 lineto 144 144 lineto 72 144 lineto
closepath stroke
```

and

```
72 72 moveto
72 0 rlineto 0 72 rlineto -72 0 rlineto
closepath stroke
```

Both of these program examples produce the same results.

Later, in chapter 18, two new operators for drawing rectangles in PostScript Level 2 will be explained.

3.2 *defining procedures*

A third way to draw the square is to define it as a *procedure*. Once defined, it can be used as often as needed. The definition for a square could look like this,

```
/square      { 72 0 rlineto 0 72 rlineto -72 0 rlineto
              closepath } def
```

and it could be used like this

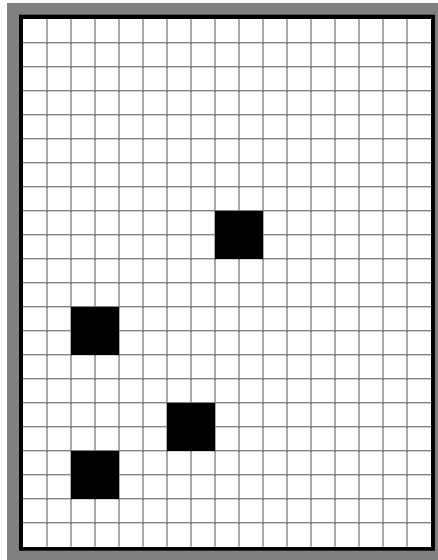
```
72 72 moveto square stroke
```

or

```
144 144 moveto square fill
```

The “/” (slash) special character identifies the word “**square**” as a name. The special characters “{” and “}” contain the definition for the name. The **def** operator associates the definition with the name. The procedure’s name or *key* is stored in the PostScript interpreter’s user dictionary while the program is run.

When a PostScript interpreter encounters a name it does not recognize, the interpreter sees if there is a definition for it in the user dictionary. If there is none present, an error message will be sent. The PostScript interpreter will assume that something was misspelled. There will be more on user dictionaries in sections 17.1–3. A complete program using the square procedure can look like this:



learn

3-2

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:square_def1.eps
%%BoundingBox:72 72 360 504
```

```
/square      { 72 0 rlineto 0 72 rlineto -72 0 rlineto
              closepath } def
```

```
72 72 moveto square fill
```

```
72 288 moveto square fill
```

```
216 144 moveto square fill
```

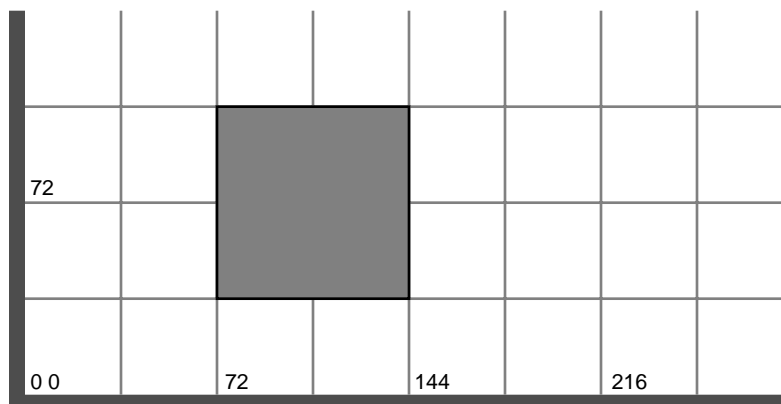
```
288 432 moveto square fill
```

3.3 *stroking & filling the same path*

As stated in section 2.11, the **stroke** and **fill** operators consume the current path. However, there is a way to use the current path for both a **stroke** and a **fill**. The current path can be copied to be used twice by saving or by taking a “snapshot” of the graphic state. The PostScript operators **gsave** and **grestore** save and restore the graphic state. The sequence for a **graphicsave** and **graphicsrestore** would be:

- 1 No current point or current path
- 2 Create current path
- 3 Save the current path by saving the graphic state and making a copy
- 4 **fill** the current path
- 5 Restore the previously saved current path
- 6 **stroke** the restored current path
- 7 Current path is cleared

The graphic state contains the current values for the current gray, line width, current point, current path, location of the origin, and a number of other values not applicable here. Here’s an example:



learn

3-3

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:gsave_1.eps
%%BoundingBox:36 36 144 108
```

```
72 36 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
closepath
```

```
gsave                                % save path, value black
    0.5 setgray fill                 % 50% black & path filled
grestore                              % saved path restored
```

```
stroke                % restored path stroked
```

The `gsave` saves the current path and the default value of black. Next, `0.5 setgray fill` changes the current value to 50% black and the current path is filled with it. `grestore` restores the current gray back to 100% black and the current path. Finally, `stroke` consumes the saved copy of the current path. The line weight of the stroke is the default value of one point.

3.4 *moving the origin*

The `translate` operator moves the origin around on the page. For example,

```
72 72 translate
```

moves the location of the origin from the lower left corner of the page to 1 inch up and over. The advantage of moving the origin is that it may make it easier to place a number of graphic objects on the page. In an earlier example, `/square` was defined to be:

```
72 0 rlineto 0 72 rlineto -72 0 rlineto closepath
```

It was moved around the page by establishing a current point at the desired location for the square, then using the procedure. A second method of placing a number of squares on the page is to have the current point always at the origin within the definition and move the origin around. For example:

```
/square
{ 0 0 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
  closepath fill } def
```

```
72 144 translate square
```

Since changing the position of the origin is changing the graphic state, it will be helpful to bracket the changes with a `gsave` and `grestore`. There can be some unexpected results if the origin is changed within a program. It is important to emphasize that using the `translate` operator does not create a current point, but transforms the coordinate system that current points are made upon.

The earlier example from section 3.2, `square_def1.eps`, can be rewritten two ways to demonstrate the use of `translate`. The first rewrite does not use `gsave` and `grestore`. The locations of the various squares are not immediately apparent and it gets rather difficult to visualize where they are because of the way the origin is changing its location.

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:square_def2.eps
%%BoundingBox:72 72 360 504
```

```
/square      { 0 0 moveto 72 0 rlineto 0 72 rlineto
              -72 0 rlineto closepath fill } def
```

```
72 72 translate square          % 72 72 from 0 0 = 72 72
```

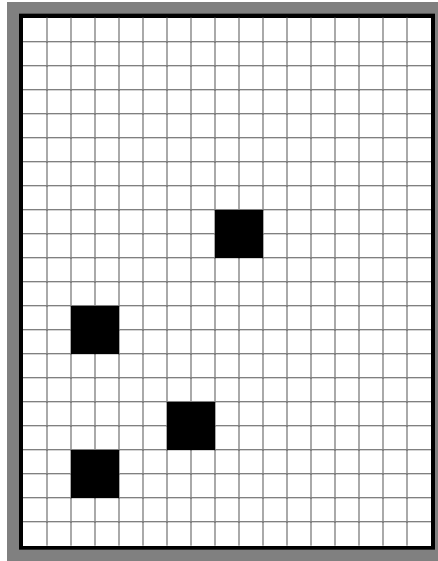
```
0 216 translate square        % 0 216 from 72 72 = 72 288
```



3-4

```
144 -144 translate square      % 144 -144 from 72 288 = -72 144
72 288 translate square       % 72 288 from -72 144 = 0 144
```

The second version places the squares at the same location on the page but the program is written differently this time by using `gsave` and `grestore`. Because of this, it is much easier to keep track of the location of the squares in this example. The location of the fourth square at 288 432 is 288 432 from the lower left corner of the page and not from an unknown point somewhere else on the page because of a change in the previous line of code.



learn

3-5

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:square_def3.eps
%%BoundingBox:72 72 360 504

/square { 0 0 moveto 72 0 rlineto 0 72 rlineto
          -72 0 rlineto closepath fill } def

gsave          % save location of origin
  72 72 translate square % change location
grestore       % restore location of origin

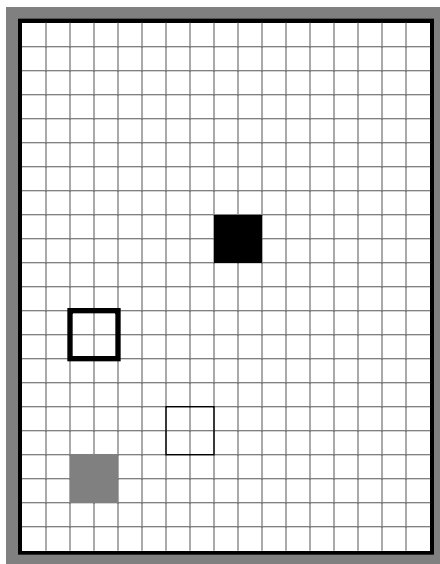
gsave
  72 288 translate square
grestore

gsave
  216 144 translate square
grestore

gsave
  288 432 translate square
grestore
```

Using `gsave` and `grestore` adds flexibility to the program. Changes to individual squares will not ripple throughout the rest of the program. Also, individual parts of a program bracketed by `gsave` and `grestore` can easily be copied and positioned into other programs.

In the next example, changes are made to the first and second squares. The third square is stroked with the default 1 point line weight and the fourth square is filled with the default color black.



3-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:square_def4.eps
%%BoundingBox:72 72 360 504

/square      { 0 0 moveto 72 0 rlineto 0 72 rlineto
              -72 0 rlineto closepath } def

gsave       % save graphic state / current color & origin
            .5 setgray
            72 72 translate square fill
grestore   % restore graphic state

gsave       % save graphic state
            4 setlinewidth
            72 288 translate square stroke
grestore   % restore graphic state

gsave       % saves origin, uses default color
            216 144 translate square stroke
grestore

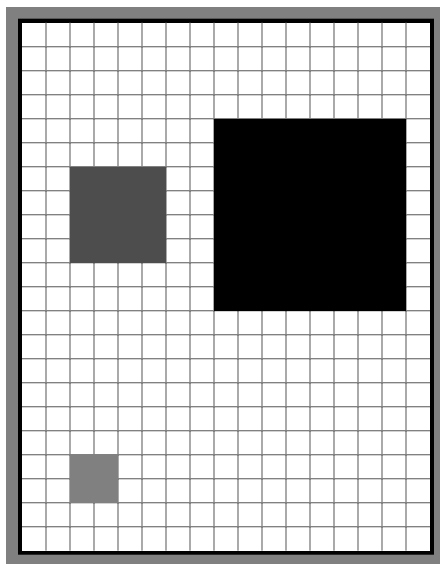
gsave
            288 432 translate square fill
grestore

```

3.5 *changing the size*

The `scale` operator makes the coordinate system larger or smaller. For example, `2 2 scale` doubles the size of the graphic state. There will also be a more detailed discussion of the `scale` operator in chapter 10.

If the procedure for `square` is defined to be a 1x1 unit, `scale` could be used to enlarge that square to whatever size is required. Note in the following example that the square is 1x1 and `scale` is used to create three sizes of the square. The individual squares are bracketed by `gsave` and `grestore` so the scaling will not multiply from square to square.



learn

3-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:square+scale_1.eps
%%BoundingBox:72 72 576 648

/square      { 0 0 moveto 1 0 rlineto 0 1 rlineto
              -1 0 rlineto closepath } def

gsave
    .5 setgray
    72 72 translate
    72 72 scale
    square fill
grestore

gsave
    288 360 translate
    288 288 scale
    square fill
grestore

gsave
    .3 setgray
    72 432 translate

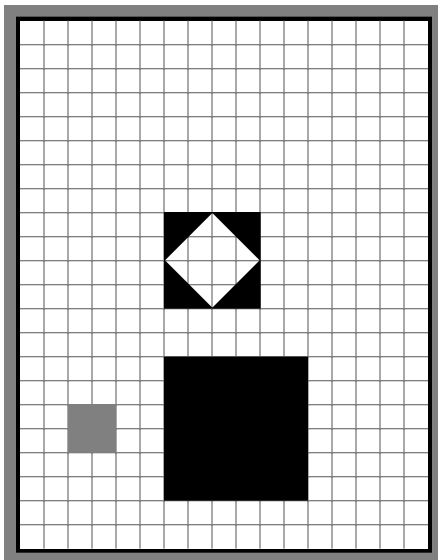
```

```

144 144 scale
square fill
grestore

```

The sequence of the PostScript operators is very important. If you are not careful, you can get yourself some unexpected results. It can make a difference if `scale` is used before or after the creation of a current path. It will also make a difference whether the path is stroked or filled. Note where the `72 72 scale` is used for each square in the following example.



3-8

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:square+scale_2.eps
%%BoundingBox:72 72 432 504

/square { 0 0 moveto 1 0 rlineto 0 1 rlineto
-1 0 rlineto closepath } def

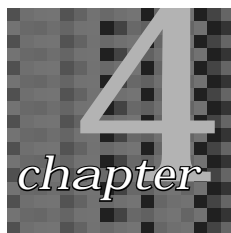
gsave                                % small gray square
    .5 setgray
    72 144 translate
    72 72 scale                       % scale graphic world 1 to 72
    square
    fill
grestore

gsave                                % lower square
    2 setlinewidth
    288 144 translate
    72 72 scale
    square
    stroke
grestore

gsave                                % upper square

```

```
288 432 translate  
square  
72 72 scale  
stroke  
grestore
```

EPS files

There are two different types of PostScript files: PostScript files that are sent to the printer and PostScript files that will be first incorporated into other documents before being sent to the printer. A good example of this is page composition software that imports PostScript illustrations. PostScript files intended to be included into another document are known as Encapsulated PostScript Files. The files are known as EPS files or EPSF files. These files may have an **.eps** file extension, not required in the Macintosh environment but used on the NeXT.

4.1 *.ps files*

The expectation with a PostScript file or text file with a **.ps** extension is that it will print one or more pages of a document. It may contain EPS files. It will contain the **showpage** operator.

showpage is the PostScript print operator. It is necessary when sending a file to a printer. In order to save space, it is not used at the end of the PostScript examples in this book. If you use *SendPS* to download your programs, you can use the “Add showpage” option under the options menu to automatically add **showpage** to your programs.

4.2 *.eps files*

The expectation with an Encapsulated PostScript File or text file with an **.eps** extension is that it is one page and may contain the **showpage** operator and a print preview.

By definition, an EPS file is a single image that may be eventually placed in another document. It also includes information that tells the importing software what the dimensions are and other information that may be needed. This information is contained in the file’s header or first lines of code. More on this in the next section.

The **showpage** operator is optional and will cause problems if the importing document is not set up to neutralize it. If it’s not, the page will print prematurely. This is handled by temporarily redefining **showpage** to do nothing until the page is ready to print. Renaming PostScript operators is covered in more detail in section 17.2.

Often a print preview will be included with the Encapsulated PostScript File so it can be seen on the computer’s monitor. The basic reason for this is the PostScript information used to create the page isn’t compatible with how images are formed

on a computer's monitor. The exception is the NeXT computer, on which the same information is sent to both screen and printer. There is no need for a preview version of the graphic.

The Macintosh needs a print preview of the PostScript file. On the Macintosh, what is seen on the screen is drawn by QuickDraw, Apple Computer's graphics language. Files created with QuickDraw are known as PICT files (short for PICTure). Even though a PICT version is seen on the screen, Adobe's PostScript version is what is sent to the printer. Depending on the graphic, the overlap in the two graphic models will produce a rough or very accurate QuickDraw preview. In all cases, what is seen on the Macintosh screen is never what is sent to the printer. The EPS options on the Macintosh are:

PostScript information / no print preview

PostScript information / 1 bit PICT preview for basic positioning

PostScript information / color or gray scale PICT preview

On IBMs and compatibles, the same situation exists as with the Macintosh. What is seen on the screen is different than what is sent to the printer.

All the EPS files we will write as part of this book and in the HyperCard version of the *LearnPS* tutorial will not have a preview for the Macintosh screen.

4.3 *the program header*

There are two lines that must be present in a PostScript file to make it a Encapsulated PostScript File. It is these lines that make an EPS file an EPS file. They are:

```
%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox: 72 72 216 216
```

There are a number of conventions in writing the beginning of a PostScript program. The following program, `2inSquare.eps`, is an example using the most common comments. They are outlined in more detail in Adobe's technical document #PN LPS5002, Encapsulated PostScript Structuring Conventions. The two above must be there, but it is a good habit to provide as much information as possible. These comment lines will be defined in section 4.6.

Notice that all of the first six lines begin with the “%” special character. These lines are not read or used by the PostScript interpreter. They contain information used by software that incorporate the file into a larger document. The software is designed to read this information as the file is imported.

The following PostScript program will look like this displayed on the NeXT or printed on a PostScript laser printer.



4-1



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:2inSquare.eps
%%Creator:John F Sherman
%%CreationDate:June 1990
%%DocumentFonts:Times-Roman
%%BoundingBox:72 72 216 216

72 72 moveto
216 72 lineto 216 216 lineto 72 216 lineto closepath
fill

/Times-Roman findfont 36 scalefont setfont
1 setgray
90 100 moveto
(EPS file) show

```

`2inSquare.eps` can be placed in a number of Macintosh applications as is. Because it is only PostScript code without a QuickDraw preview, it will appear on the screen as an outline box. The size of this outline box is taken from the `%%BoundingBox` comment. `2inSquare.eps` will print correctly, however, using the PostScript information.

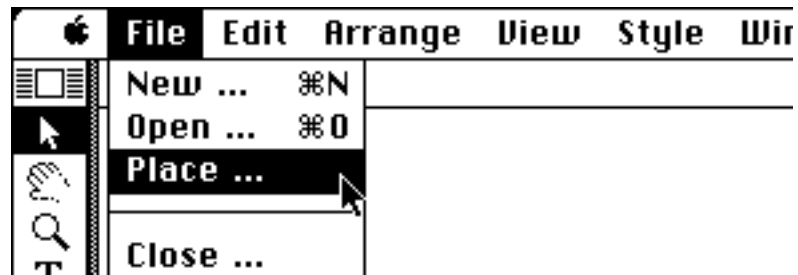
4.4

placing EPS files into Macintosh documents

The file, `2inSquare.eps` can be placed as is into an Illustrator or PageMaker document. In fact, every PostScript example in this book can be placed into documents in the same way.

To place an EPS file into an Illustrator document, first create the text file with any word processor. Its name doesn't have to have the `.eps` file extension and its name doesn't even have to match the `%%Title:` comment. When "Place" under the "File" menu in Adobe Illustrator is chosen (see figure 4-1), a dialog box will appear to locate an EPS file. The dialog box will display TEXT and EPSF file types (more on file types in the next section).

figure 4-1



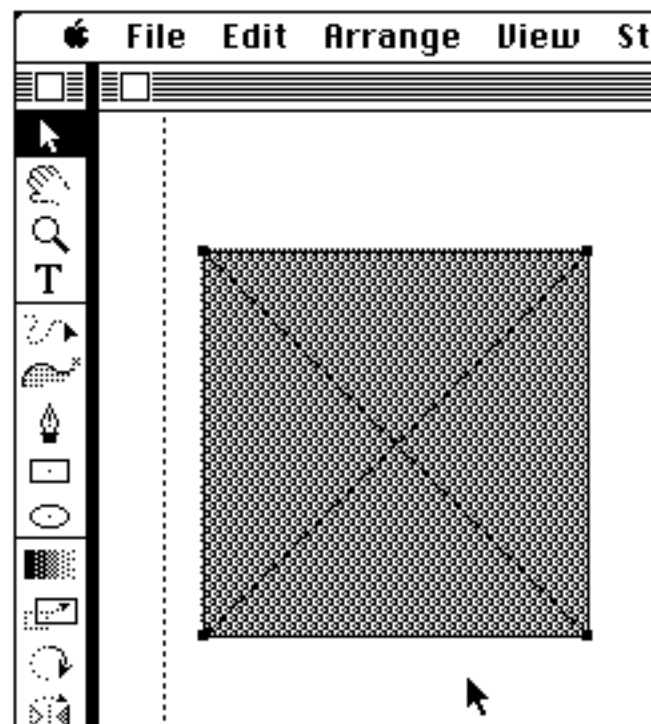
When a file is selected and opened, Illustrator will check the beginning of the file to see if the correct header information is present. Specifically, as mentioned in the previous section, these two lines:

```

%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox:72 72 216 216
    
```

Once placed, the contents of the file cannot be edited, but it can be moved to any location on the page. In addition, the scale, rotate, reflect, and skew transformation tools of Illustrator can be used on the file.

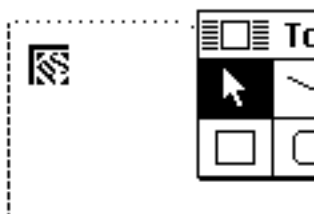
figure 4-2



The placed `2inSquare.eps` will appear like figure 4-2 in Adobe Illustrator. When this file is printed, the PostScript information will be used.

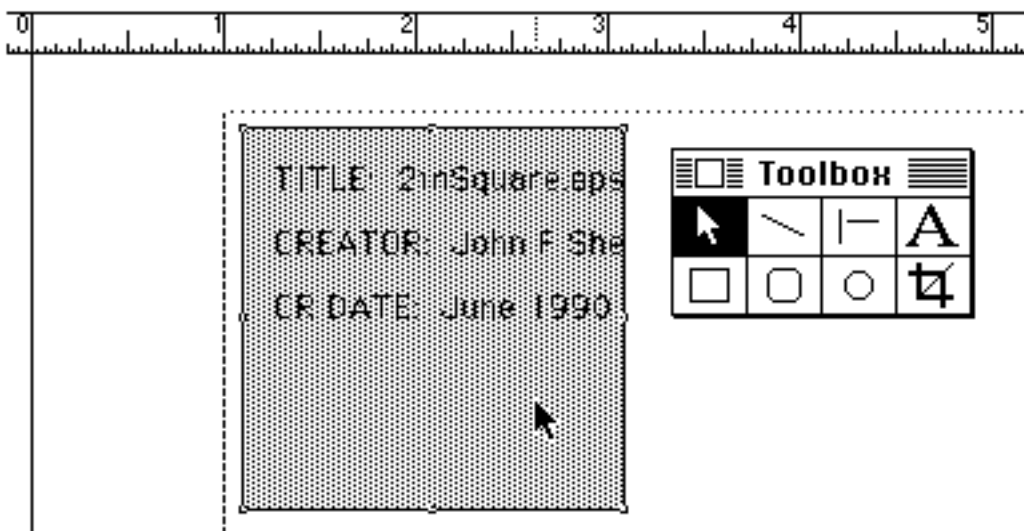
In Aldus PageMaker, essentially the same steps are taken with the “Place” menu found under the File menu. Select the EPS file from within the dialog box in the usual way. The arrow cursor will become a PS cursor. Click the PS cursor on the page where you want the PostScript file to be located (see figure 4-3).

figure 4-3



Again, only a gray box will appear on the page but notice that the title, creator, and creation date appear within the box (see figure 4-4). This will be displayed if those lines are present in the header of the program.

figure 4-4



The gray box for the EPS file can be moved wherever it's needed once placed.

4.5 *changing a TEXT to an EPSF file type*

Some Macintosh applications can only import EPS files if their file type has been changed from TEXT to EPSF. Their file contents are not any different, just a tag that tells the importing software what kind of file it is.

figure 4-5



Every Macintosh file has a file type and creator associated with it. This information is used by the Finder to determine what icon to display (file type) and what application to launch (file creator). In figure 4-5, Files A and B are made by Adobe Illustrator and Files C and D are made by Microsoft Word (the creators). These two applications can make the same file type. Files A and D are TEXT. Microsoft Word (as an option) and Adobe Illustrator (by default) can save their files as TEXT. File B is EPSF and File C is WDBN. This means you could open an Illustrator file with MS Word and look at the PostScript code it has written. It doesn't work the other

way, however; Illustrator can't open Microsoft Word's files. Illustrator files are PostScript files. Illustrator gives you the option to include a preview to make the file an Encapsulated PostScript File. When this is done, Illustrator changes the file type from TEXT to EPSF. Because of this, the file will have a different icon. File A is a TEXT file, File B is an EPSF. File C is Microsoft Word's default file type WDBN and File D is a TEXT file. Also, file types and creators are used in dialog boxes to filter away file names of other programs.

In some other Macintosh applications, such as Letraset's Ready, Set, Go, the file type of `2inSquare.eps` will need to be changed from "TEXT" to "EPSF." This can be done with a number of utilities, such as CE's DeskTop or MacTools. In Illustrator and PageMaker, the Place dialog box displays all TEXT files, EPS formatted or not. In other software, the dialog box displays EPSF or other graphic file formats, but not TEXT files.

The steps involved in changing a file that's a TEXT file type to an EPSF file type can be done with several utilities. I'll demonstrate how its done using DiskTop, a desk accessory from CE Software. The process will be similar with other utilities, such as MacTools from Central Point Software.

Name	Type	Creator
2inSquare.eps	TEXT	????
applications	28 files/folders	
bmFonts	13 files/folders	

figure 4-6

After opening DiskTop, select the file you wish to change. Notice that in this case the file is of type TEXT and is created by ????. It is ????. because the file was not originally written on a Macintosh and the Macintosh therefore doesn't know who made it. Your file may show MSWD as the creator if the file was made by Microsoft Word or MACA if it was written by Claris MacWrite. For our purposes, it does not matter what the creator ID is.

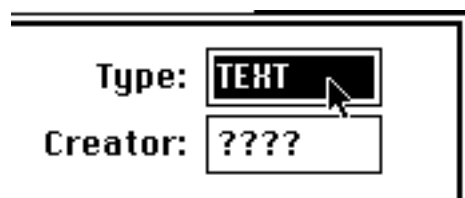


figure 4-7

Select "Get Info ..." from the DiskTop menu. In the upper right hand corner you will see two boxes to enter new Type and Creator IDs (see figure 4-7).

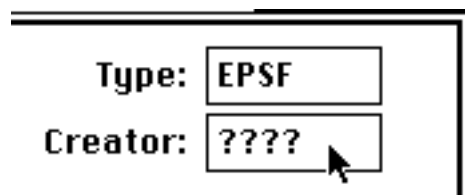


figure 4-8

Change TEXT to EPSF and save the changes (see figure 4-8). The file is now ready

to be used by programs such as Ready, Set, Go or DesignStudio that look for the EPSF file type. Unfortunately, you will need to reverse the steps taken and change the file back to TEXT should you need to edit the file.

It's important to note that just changing the file type to EPSF from TEXT doesn't automatically make a file an Encapsulated PostScript File. The correct info must be present in the header of the file. Also, not all Macintosh software will be able to take advantage of EPS files. Check the manual of the software in question.

4.6 *the program header, line by line*

Here is a brief explanation of each of the header lines used in section 4.3. All of these lines are seen as comments by the PostScript interpreter. They aren't used to make a mark on the page, but to give information to either importing software or print servers. They do not have to be in a certain order except the following should always be the first line:

```
%!PS-Adobe-2.0 EPSF-1.2
```

This line identifies that the PostScript program conforms to the standard Adobe structuring conventions. More often than not, it must be present for the file to be properly used by an application. Some programs will not be able to use the file if it isn't present.

```
%%Title:2inSquare.eps
```

This line is useful, as seen earlier in the PageMaker example (figure 4-4). There, the file name is shown in the gray box. In other applications, it may be listed when information is requested.

```
%%Creator:John F Sherman
```

This line might read

```
%%Creator:Adobe Illustrator 88(TM) 1.6
```

if it was created by Adobe Illustrator.

```
%%CreationDate:June 1990
```

This is a record of when the file was created.

```
%%DocumentFonts:Times-Roman
```

This line can be very important in some print environments. This line may be used to let a printer know it has to get a font from its hard disk or server. If more than one font is being used in a file, it would be written as follows:

```
%%DocumentFonts:Times-Roman
```

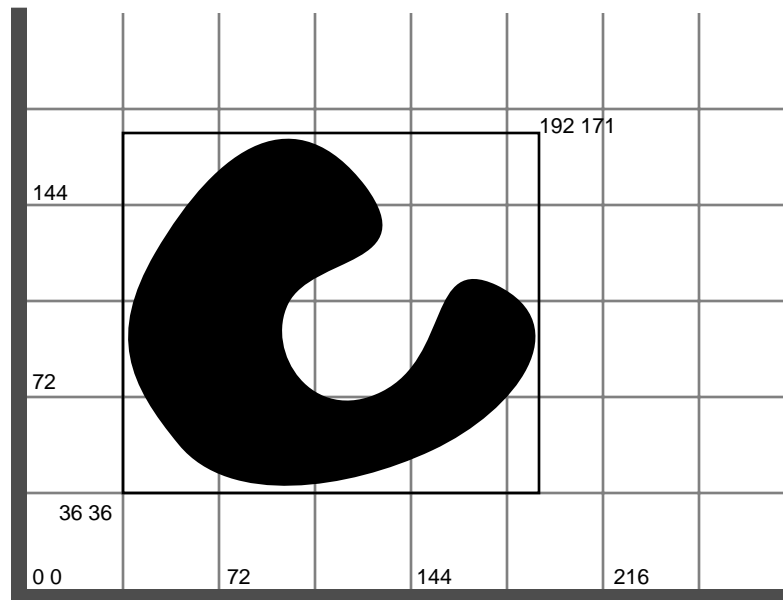
```
%%+ AvantGarde-Demi
```

```
%%+ Helvetica
```

```
%%BoundingBox:72 72 216 216
```

This is the most important line in the file header because it describes the smallest rectangle that the graphic would fit in. This information is used for positioning the EPS file into other files. In the example below, the shape fits in a rectangle whose lower left is at x and y 36 36 and upper right is at 192 171. Its bounding box would therefore be:

```
%%BoundingBox:36 36 192 171
```



4-2

```
!PS-Adobe-2.0 EPSF-1.2
%%Title:BBBox_1.eps
%%CreationDate:June 1990
%%BoundingBox:36 36 192 171
```

```
36 36 translate
12 90 moveto
36 131 65 149 91 115 curveto
113 85 71 89 62 71 curveto
53 53 71 25 97 38 curveto
123 51 114 90 140 78 curveto
171 63 148 31 116 16 curveto
84 1 40 -5 21 18 curveto
2 41 -6 59 12 90 curveto
fill
```

If `BBBox_1.eps` were to be brought into TopDraw on the NeXT computer, it would appear as in figure 4-9.

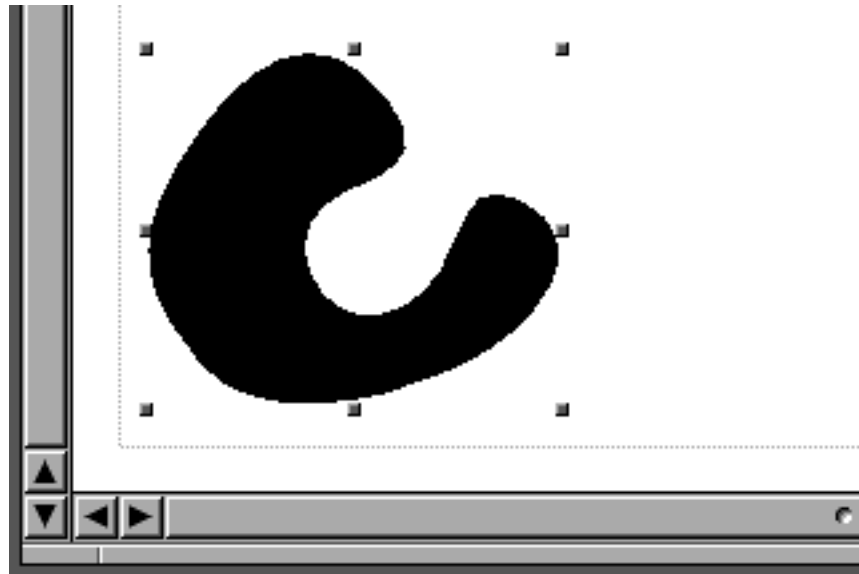


figure 4-9

The eight little squares that surround the shape appear when the object is selected for an edit in TopDraw. These are handles that can be used for stretching and scaling the picture and lay on the border that corresponds to the `BoundingBox`.

There is more information on the `BoundingBox` in section D.5.





understanding error messages

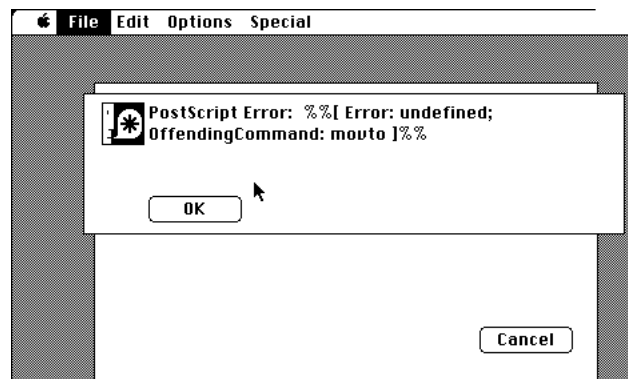
As with any programming language, mistakes of all kinds can happen. There can be misspellings, missing arguments, or operators out of sequence. The PostScript interpreter provides feedback to help the programmer locate the mistake.

Depending on the means you are using to download the PostScript file, the error message can be received in a number of different ways. The error message can be sent to a text file on your disk drive or sent to a window on the computer's monitor.

An error message will look like this:

```
%%[Error: undefined; OffendingCommand: movto]%%
```

It will appear on the Macintosh monitor like this when using *SendPS*:



This message may appear briefly on the monitor and then be sent to a text file on disk under the name of the printer. It might be named "LaserWriter Log" and be found in the same folder as *SendPS*. Following are some of the most common error messages and where to look when they occur.

5.1

undefined

undefined is one of the most common error messages to receive.

For example, with this PostScript program,

```
%!PS
%%Title:mistake1.ps
```

```
72 72 movto      % moveto is misspelled
```



5-1



5-2

```
144 72 lineto 144 144 lineto 72 144 lineto 72 72 lineto fill
```

this error message is sent back:

```
%%[Error: undefined; OffendingCommand: movto]%%
```

This message could be read as “I cannot run this PostScript program because I don’t understand movto.” In this case, it will be immediately recognized that we misspelled `moveto`. Another example is the following:

```
%!PS
%%Title:mistake2.ps

/Box {72 72 moveto
      144 72 lineto 144 144 lineto 72 144 lineto 72 72 lineto
      fill} def
box      % Box, not box, is defined above
```

The program fragment produces this error message:

```
%%[Error: undefined; OffendingCommand: box]%%
```

This is a situation similar to the previous error. In both examples, when the PostScript interpreter encounters a word it doesn’t understand, it searches first the dictionary of user defined procedures, and then the system dictionary to see if the questionable word is defined there. If not, the interpreter creates the `undefined` error. In the first example `moveto` was misspelled; `movto` will not be found in any dictionary. The second example has the same basic problem: `box` is not the same as `Box`, and `Box` is what is defined and located in the user dictionary.

5.2

typecheck

Another common mistake is the case when an operator isn’t misspelled but is missing from a certain PostScript operator sequence. For example:



5-3

```
%!PS
%%Title:mistake3.ps

/Helvetica-Bold 100 scalefont setfont % findfont is missing

36 36 moveto
(Type) show
```

The program fragment produces this error message:

```
%%[Error: typecheck; OffendingCommand: scalefont]%%
```

Here, `findfont` was forgotten after the `/Helvetica-Bold`. `scalefont` was in error because there wasn’t anything to scale. Another example is this program that creates a pattern.



5-4

```
%!PS
%%Title:mistake4.ps

/word (LearnPS) def
20 20 1 [.1 0 0 .1 0 0] {word} imagemask % boolean not 1
```

The program fragment produces this error message:

```
%%[Error: typecheck; OffendingCommand: imagemask]%%
```

The error here is not what is missing, but that the `1` located after the `20 20` should be either a `true` or a `false`. With the related `image` operator, the `1` would be correct in that position.

5.3

stackunderflow

The `stackunderflow` error occurs when an operator does not have the expected number of *arguments* on the stack. In the following line of PostScript, `arc` needs five numbers on the operand stack to do its work and there are only four present waiting to be used.



5-5

```
%!PS
%%Title:mistake5.ps

72 36 0 270 arc stroke % missing the value for radius
```

The program fragment produces this error message:

```
%%[Error: stackunderflow; OffendingCommand: arc]%%
```

In this next example, the height argument is missing for the `image` operator.



5-6

```
%!PS
%%Title:mistake6.ps

/picStr 2 string def
/Mickey {16 1 [.2 0 0 .2 0 0] % missing height
        {currentfile picStr readhexstring pop} image} def
Mickey
FE 0F FD B3 FD 7D F5 F8 F7 F8 F6 D3 80 D3 00
F7 00 CF 06 1F 04 0F 8C 0F FC 0F FC 0F FE 1F
```

The program fragment produces this error message:

```
%%[Error: stackunderflow; OffendingCommand: image]%%
```

This message states that one of the arguments for the `image` operator is missing. Had the matrix *array* or procedure been missing, the error would have been a `typecheck` error.

5.4

rangecheck

The `rangecheck` error is caused when an expected value is outside of the required range. In the example below, the matrix array for the `image` operator is missing the last zero in the array, which should read `[.2 0 0 .2 0 0]`.



5-7

```
%!PS
%%Title:mistake7.ps

/picStr 2 string def
```

```

/Mickey      { 16 15 1 [.2 0 0 .2 0]  % array missing last 0
              {currentfile picStr readhexstring pop} image} def
Mickey
FE 0F FD B3 FD 7D F5 F8 F7 F8 F6 D3 80 D3 00
F7 00 CF 06 1F 04 0F 8C 0F FC 0F FC 0F FE 1F

```

The program fragment produces this error message:

```
%%[Error: rangecheck; OffendingCommand: image]%%
```

5.5

limitcheck

The most common reason for a `limitcheck` error is that a current path has become too complex. For example, the maximum number of points that can be created in a current path before an error on the Apple LaserWriter Plus is fifteen hundred. This number will be different depending on the printer. The resolution that the printer is set to can also have a bearing on whether this error will occur.



5-8

```

%!PS
%%Title:mistake8.ps

/Helvetica-Bold findfont 100 scalefont setfont
/fountstring 256 string def
0 1 255 { fountstring exch dup put } for

36 36 translate
0 0 moveto
(abcdefghijklm) true charpath clip

620 150 scale
255 1 8 [255 0 0 1 0 0] {fountstring} image

```

The program fragment produces this error message:

```
%%[Error: limitcheck; OffendingCommand: clip]%%
```

What can be difficult about this error is anticipating when it will be made. A PostScript program might run error free on an Apple LaserWriter but fail on a high resolution Linotron. The program `mistake8.ps` above, for example, fails on an Apple LaserWriter Plus but runs without problem on a NeXT laser printer.

There are two solutions when this error occurs. In the example above, the `clip` could be done in several parts, clipping first “abcde,” then the “fghi,” and then the “jklm.” This can be accomplished by using the `stringwidth` operator.

The syntax for `stringwidth` is:

```
string stringwidth returning xw yw
```

where `string` is the *string* to be measured, and `xw` is the returned width of `string`. The returned `yw` value is usually discarded because the vertical measurement in roman fonts is always zero.



5-9

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fix8.eps
%%Date:17 Dec 1990
%%DocumentFonts:Helvetica-Bold
%%BoundingBox:0 0 355 70

0 0 355 70 rectstroke

/Helvetica-Bold findfont 50 scalefont setfont
/fountstring 256 string def
0 1 255 { fountstring exch dup put } for
/x1 (abcde) stringwidth pop 12 add def
/x2 (abcdefghi) stringwidth pop 12 add def

gsave
  12 24 moveto
  (abcdefghijklm) true charpath clip

  350 70 scale
  255 1 8 [255 0 0 1 0 0] {fountstring} image
grestore

gsave
  x1 24 moveto
  (fghi) true charpath clip

  350 70 scale
  255 1 8 [255 0 0 1 0 0] {fountstring} image
grestore

gsave
  x2 24 moveto
  (jklm) true charpath clip

  350 70 scale
  255 1 8 [255 0 0 1 0 0] {fountstring} image
grestore

```

The second way is to use the `setflat` operator. To get the program above to work on the Apple LaserWriter Plus, `9 setflat` is added to the beginning of the program. `setflat` controls the accuracy of curved paths. The larger the number, the coarser the curve. In `mistake8.ps` above, the better solution might be to divide the clipping operation because the `9 setflat` makes the letterform's curves very rough. Following is an example of what a section of the print would look like with `setflat` used.

Another common reason for getting a `limitcheck` error is using the autotrace feature of Adobe Illustrator on the wrong kind of template.

5.6 *nocurrentpoint*

The `nocurrentpoint` error is sent when a needed current point is missing. It is usually a missing `moveto`. In this example, `curveto` needs an existing current point. Unlike `arc` or `arcn`, this line of PostScript must be preceded by either a `moveto` or the current point on a path being made.



5-10

```
%!PS
%%Title:mistake9.ps

100 28 116 110 144 72 curveto stroke
```

The program fragment produces this error message:

```
%%[Error: nocurrentpoint; OffendingCommand: curveto]%%
```

Another example is type positioned by `moveto`.



5-11

```
%!PS
%%Title:mistake10.ps

/Helvetica-Bold findfont 72 scalefont setfont
(Type) show
```

The program fragment produces this error message:

```
%%[Error: nocurrentpoint; OffendingCommand: show]%%
```

In this example, the `lineto` needs to begin with a `moveto`.



5-12

```
%!PS
%%Title:mistake11.ps

144 72 lineto 144 144 lineto 72 144 lineto 72 72 lineto
fill
```

The program fragment produces this error message:

```
%%[Error: nocurrentpoint; OffendingCommand: lineto]%%
```

In each case, a clue is provided on where to find the error.

5.7

syntaxerror

The typical reason for this error is that the special characters (, [, {, and < are not paired with the corresponding),], }, and > in a program. In this example, the closing parenthesis is missing from the string.



5-13

```
%!PS
%%Title:mistake12.ps

/Helvetica-Bold findfont 72 scalefont setfont
36 36 moveto
(Type show
```

The program fragment produces this error message:

```
%%[Error: syntaxerror; OffendingCommand: Type show
```

Had the first parenthesis been the one missing instead of the second, the message would be this:

```
%%[Error: syntaxerror; OffendingCommand: Type]%%
```

In this example, the closing parenthesis is missing from the definition.



5-14

```
%!PS
%%Title:mistake13.ps

/word (LearnPS def
20 20 2 [.1 0 0 .1 0 0] {word} image
```

The program fragment produces this error message:

```
%%[Error: syntaxerror; OffendingCommand: LearnPS def
```





drawing basics

Up to this point, we have covered the basics of creating a path for lines and rectangles using only the `lineto` and `rlineto` operators. Obviously, there are a number of other path construction operators for making other kinds of paths. There are five PostScript operators for making circles, arcs and curves. All of these operators can be used in combination with each other to make any shape possible. Once made, the path can be stroked, filled, or both.

6.1 *creating arcs*

The `arc` and `arcn` operators are used to draw arcs and circles. Their syntax is:

```
xc yc radius ⌘begin° finish° arc
xc yc radius ⌘begin° finish° arcn
```

Where:

<code>xc</code>	is the center of the arc/circle on the x axis.
<code>yc</code>	is the center of the arc/circle on the y axis.
<code>radius</code>	is the radius of the arc/circle.
<code>begin°</code>	is the beginning point of the path.
<code>finish°</code>	is the finishing point of the path.

∞The difference between `arc` and `arcn` is the direction of the drawing from `⌘begin°` to `finish°`. See figure 6-1 below. The circle is divided into degree points in the counterclockwise direction from the three o'clock position. The same points are used by both `arc` and `arcn`. With `arc`, the direction of `begin°` to `finish°` is in the counterclockwise direction, with `arcn` it is clockwise. The 0° and 360° points share the three o'clock position. Therefore, both `0 360 arc` and `0 360 arcn` could be used to draw a circle.

Neither `arc` nor `arcn` needs an existing current made by a `moveto` the way `lineto` does. However, they can take advantage of a `moveto` as will be shown later in `arc_2.eps`. If there is an existing current point, it will attach itself with a straight line to the `begin°`. To insure against an unwanted line, clear the current path with the operator `newpath`.

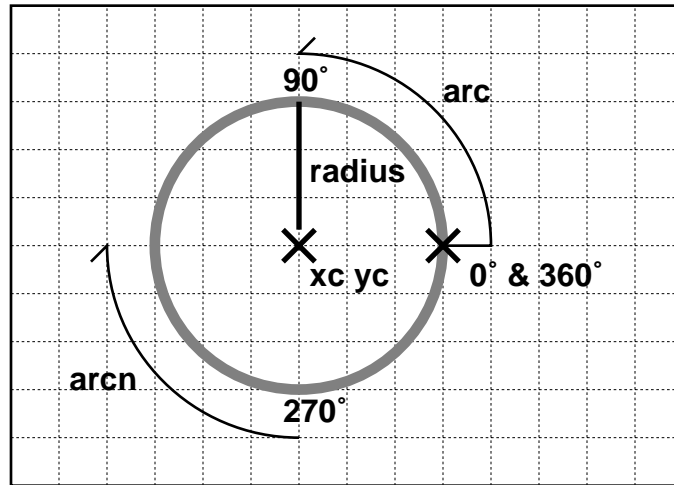
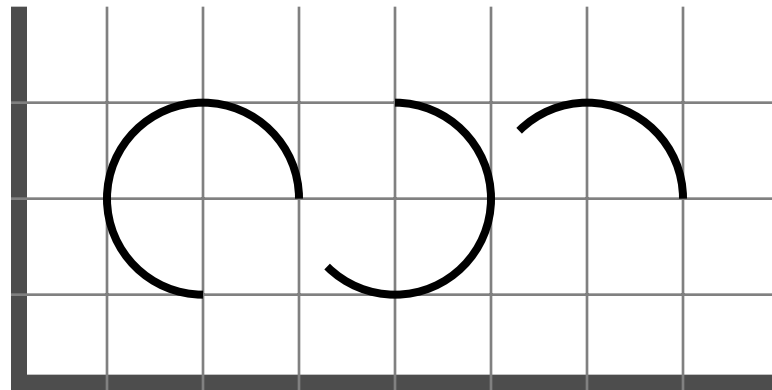


figure 6-1

Here are three arcs drawn with `arc`.



6-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arc_1.eps
%%BoundingBox:34 34 256 110
    
```

```

3 setlinewidth

72 72 36 0 270 arc stroke

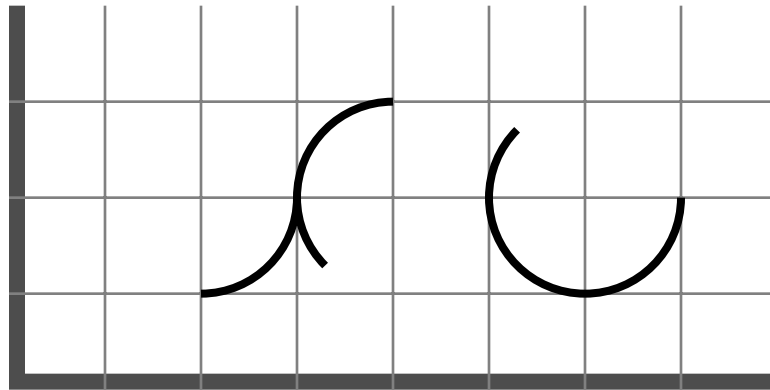
144 72 36 225 90 arc stroke

216 72 36 0 135 arc stroke
    
```

Now, the same program again with the same arguments used in `arc_1.eps`, but substituting `arcn`. The opposite portion of the arc is drawn.



6-2



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arcn_1.eps
%%BoundingBox:34 34 256 110

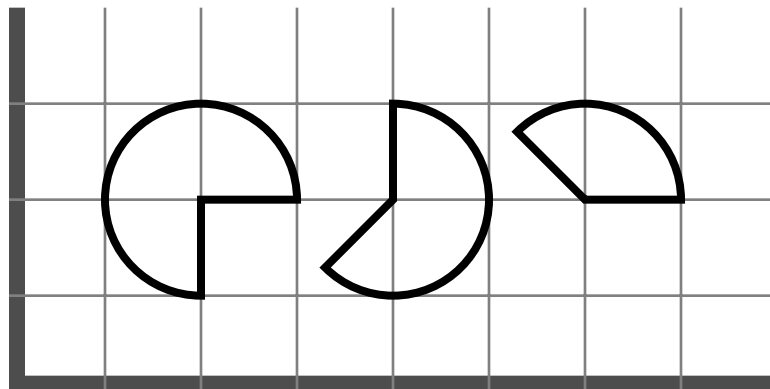
```

```
72 72 36 0 270 arcn stroke
```

```
144 72 36 225 90 arcn stroke
```

```
216 72 36 0 135 arcn stroke
```

The two previous arc examples were drawn without the customary establishing of a current point. If there is no existing current, one is created at the location of `begin`, the starting point of the arc. If there is an existing current point, a line will be drawn to the point of `begin`. Notice the difference in the following two PostScript programs when a current point is made at the center of the arc and `closepath` is used.



6-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arc_2.eps
%%BoundingBox:34 34 256 110

```

```
72 72 moveto
```

```
72 72 36 0 270 arc closepath stroke
```

```
144 72 moveto
```

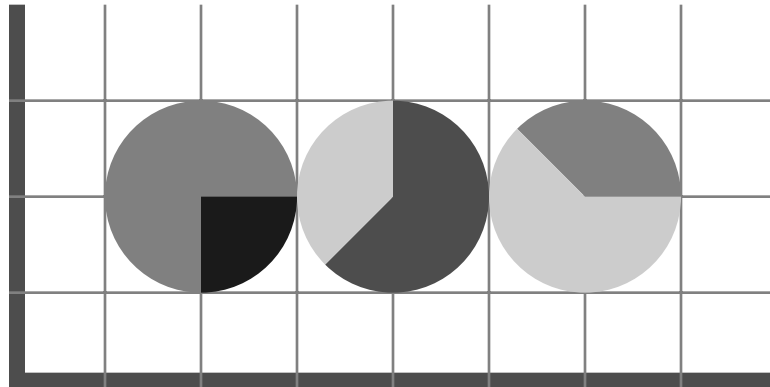
```
144 72 36 225 90 arc closepath stroke
```

```
216 72 moveto
```

```
216 72 36 0 135 arc closepath stroke
```

In all the arcs in the previous and next example, a current point is first made at the center of the arc. `begin` connects to that current point and therefore draws a line. After `arc`, the current point will be at `finish`. Then, `closepath` draws a line to the original current point made with the earlier `moveto`.

In this example, both `arc` and `arcn` are used with `fill`.



learn

6-4

```
!PS-Adobe-2.0 EPSF-1.2
```

```
%%Title:arc_3.eps
```

```
%%BoundingBox:34 34 256 110
```

```
.5 setgray
```

```
72 72 moveto
```

```
72 72 36 0 270 arc closepath fill
```

```
.1 setgray
```

```
72 72 moveto
```

```
72 72 36 0 270 arcn closepath fill
```

```
.3 setgray
```

```
144 72 moveto
```

```
144 72 36 225 90 arc closepath fill
```

```
.8 setgray
```

```
144 72 moveto
```

```
144 72 36 225 90 arcn closepath fill
```

```
.5 setgray
```

```
216 72 moveto
```

```
216 72 36 0 135 arc closepath fill
```

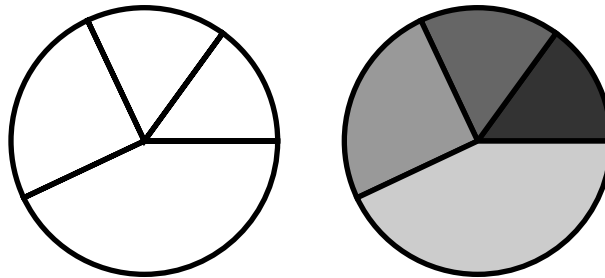
```
.8 setgray
```

```
216 72 moveto
```

```
216 72 36 0 135 arcn closepath fill
```


6.2 *making a pie chart using arc*

Using `arc` to develop a pie chart is a good exercise in learning how to make use of the operand stack and a few of the PostScript math operators. The pie chart below is for the percentages of 43%, 25%, 17%, and 15%. This program example is written to handle any four percentages that add up to 100. The program for the filled version of the pie chart can be found in chapter 19.



learn

6-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:pieChart_1.eps
%%Creator:John F Sherman
%%CreationDate:June 1990
%%BoundingBox:10 10 154 154

2 setlinewidth
10 10 translate

43 25 17 15      % the 4 percentages with space between
/p1 exch 3.6 mul def
/p2 exch 3.6 mul p1 add def
/p3 exch 3.6 mul p2 add def
/p4 exch 3.6 mul p3 add def

/x 72 def
/y 72 def
/r 72 def
/wedge{arc closepath stroke} def

x y moveto
  x y r 0 p1 wedge

x y moveto
  x y r p1 p2 wedge

x y moveto
  x y r p2 p3 wedge

x y moveto
  x y r p3 p4 wedge

```

Looking at the first part of the program, `43 25 17 15` are the four percentages with a word space separating them. They can be any four percentages as long as

they add up to 100. The four numbers are entered onto the stack.

`/p1 exch 3.6 mul def` defines our first point on the arc moving counterclockwise from the 0° point. If we were to chart the stack of these two lines of code, it would look like figure 6-2.

exch	15	mul	54	def	17
/p1	/p1	3.6	/p1	54	25
15	17	15	17	/p1	43
17	25	/p1	25	17	
25	43	17	43	25	
43		25		43	
		43			

figure 6-2

In column one, the four numbers, `/p1`, and `exch` are entered onto the stack. `exch` exchanges the top two items on the stack. Therefore, as we see in column two, `/p1` and `15` have switched places on the stack. `p1` will eventually be the name for our first point on the arc. Next in column three, `3.6` and `mul` are added to the stack. `mul` multiplies the top two numbers on the stack and pushes (puts) the product onto the stack. We multiply by 3.6 because our four numbers are fractions of 100 and our pie wedges will be fractions of 360. In column four, `54` is on top from the multiplying of 3.6 by 15. Next comes `def`, which associates `/p1` with the value 54. The steps are then repeated in the same way for points `/p2`, `/p3`, and `/p4`. However, with those points the new point will also need to have the value of the previous point added to it. We do this because we need to make our way around the circumference of the pie. Note that in defining the variable `p1` the `{ }` are not used. They are only used when defining procedures.

The next line of the program begins in the same way except for the addition of adding the value of `p1` to definition of `p2`. Starting at where 17 times 3.6 equals 61.2 is pushed onto the stack, the stack would continue as shown in figure 6-3.

61.2	add	115.2	def	25
/p2	p1 (54)	/p2	115.2	43
25	61.2	25	/p2	25
43	/p2	43	25	43
	25			
	43			

figure 6-3

`p1` and `add` are added to the stack. As you might guess, `add` adds the top two numbers on the stack. We defined `p1` in the previous line to be 54. So we then have in column three `p1` plus 61.2 equaling 115.2 pushed onto the stack. It is then given the name `p2`. Our stack now has two numbers left to process as `p3` and `p4`. They are done in the same way as `p2`. We now have these points around the pie (see figure 6-4).

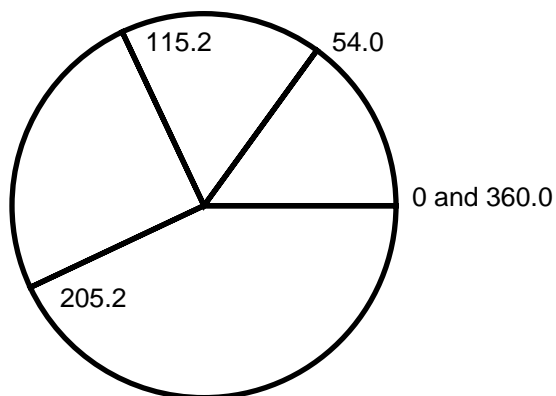


figure 6-4

The next part of the program is:

```
/x 72 def
/y 72 def
/r 72 def
/wedge {arc closepath stroke} def
```

`x` and `y` determine the center of the pie chart. They will be used by both the `moveto` and `arc` operators later in the program. `r` is the radius of the pie chart. Defining the `x`, `y`, and `r` variables like this gives us flexibility. We need to change only one number, not many, to make a change to the chart. `wedge` is the name for the procedure `{arc closepath stroke}`. This simplifies the program and helps it to be more readable.

Next in the program comes the actual drawing of the pie chart using the values and procedures defined earlier.

```
x y moveto
  x y r 0 p1 wedge
```

These program lines draw the first wedge, and so forth.

To rewrite the program to work with five percentage numbers, add these lines:

```
/p5 exch 3.6 mul p4 add def
and
x y moveto
  x y r p4 p5 wedge
```

Follow this same pattern for each wedge that will be needed for the chart. Also, since we know that the last point must be 360, we could leave out the process of defining the last point and put in 360.

6.3 *drawing curves*

Complex curves are drawn in PostScript with the `curveto` operator, which uses Beziér cubic control points. If you have ever used Adobe Illustrator, you may already have become acquainted with them when drawing curves. Most complex curves are difficult to draw without the aid of a software package such as Adobe Illustrator, but some basic examples can be demonstrated.

The syntax of `curveto` is:

`b1x b1y b2x b2y x2 y2 curveto`

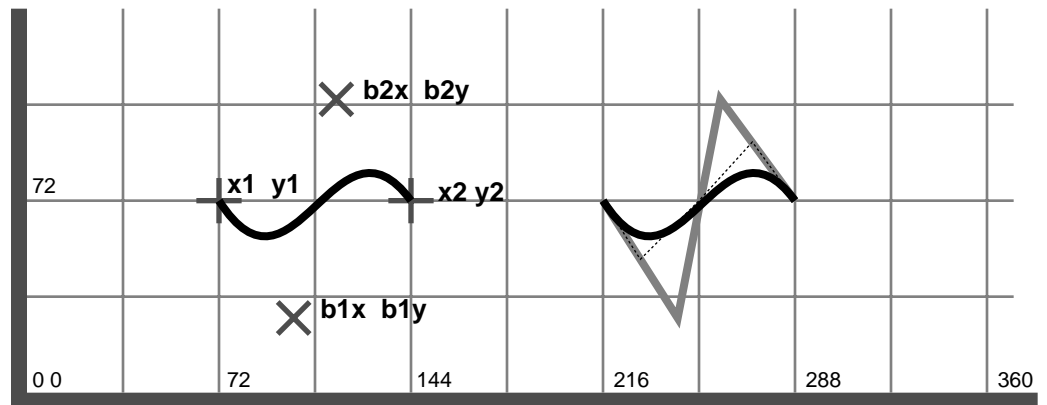
Where

`b1x b1y` is the location of the first Beziér control point.

`b2x b2y` is the location of the second Beziér control point.

`x2 y2` is the end of the curve and new current point.

`curveto` also requires an existing current point, `x1 y1`, as a starting point, unlike `arc` and `arcn` which can get by without one. It is important to note that the two points `b1x b1y` and `b2x b2y` are not points on the actual path of the curve. Following is an example and diagram. `x2 y2` becomes the new current point.



6-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:curveto_1.eps
%%BoundingBox:72 55 144 88
    
```

```

3 setlinewidth
72 72 moveto
100 28 116 110 144 72 curveto stroke
    
```



curveto demo

Beziér cubic control points create a curve by using the points `x1 y1`, `b1x b1y`, `b2x b2y`, and `x2 y2` to mark off a frame as seen below in the first frame of figure 6-5. Then each line segment of the frame is repeatedly halved until the curve is formed. Figures 6-5 and 6-6 illustrate this.

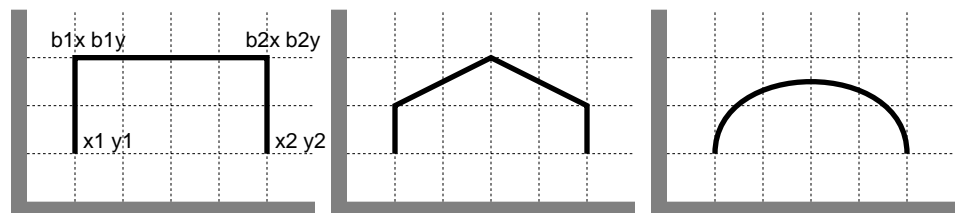


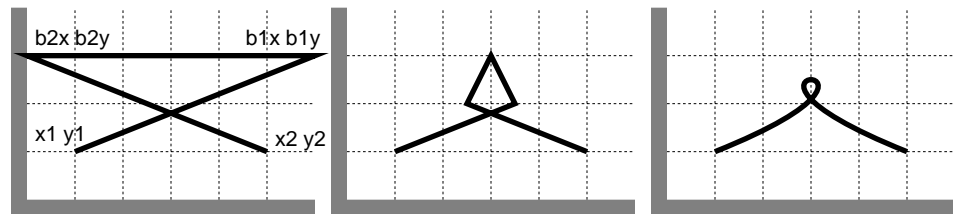
figure 6-5

The curve of the last frame is produced by

```

18 18 moveto
18 54 90 54 90 18 curveto stroke
    
```

figure 6-6



The curve of the last frame is produced by

```
18 18 moveto
108 54 0 54 90 18 curveto stroke
```

There is also the related `rcurveto`, which is much like `rlineto`.

6.4 *the arcto operator*

An alternative way of constructing arcs or curves is the `arcto` operator. `arcto` rounds off a corner made by two lines with an arc of a specified radius. The syntax of `arcto` is:

```
x1 y1 x2 y2 radius arcto
```

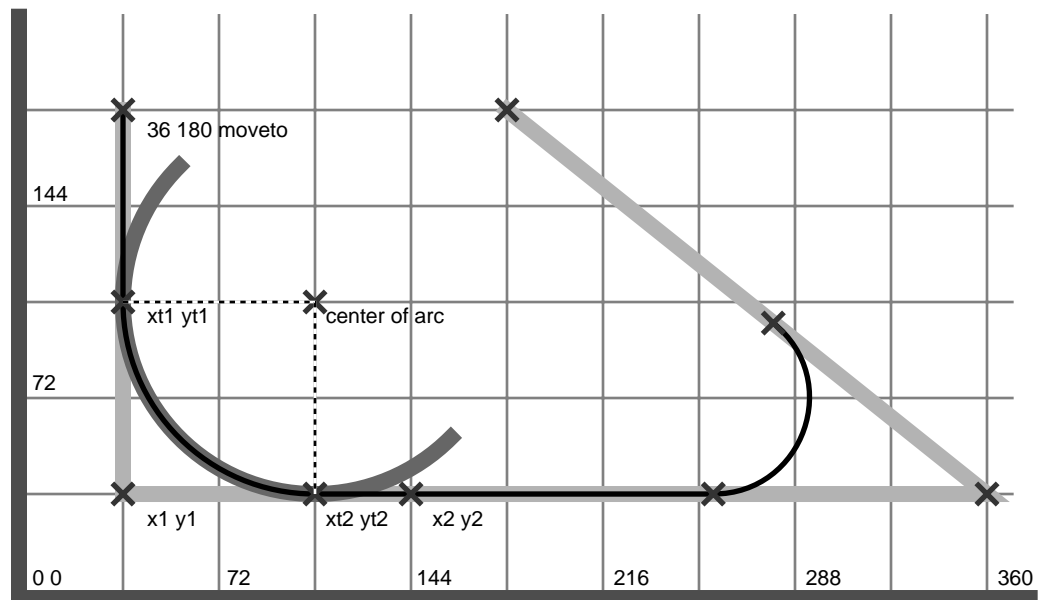
Where:

`x1 y1` is the end point of the first line and marks the corner.

`x2 y2` is the end point of the second line.

`radius` is the radius of the arc rounding off the corner of the two lines.

In addition, `arcto` requires a current point and pushes the new tangent points `xt1 yt1` and `xt2 yt2` onto the operand stack. In the majority of cases, these four points are discarded. `xt2 yt2` becomes the new current point. See example below.





6-7



arcto demo

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arcto_1.eps
%%CreationDate:June 1990
%%BoundingBox:34 34 294 165

```

```
3 setlinewidth
```

```

36 180 moveto
36 36 144 36 72 arcto
360 36 180 180 36 arcto

```

```
stroke
```

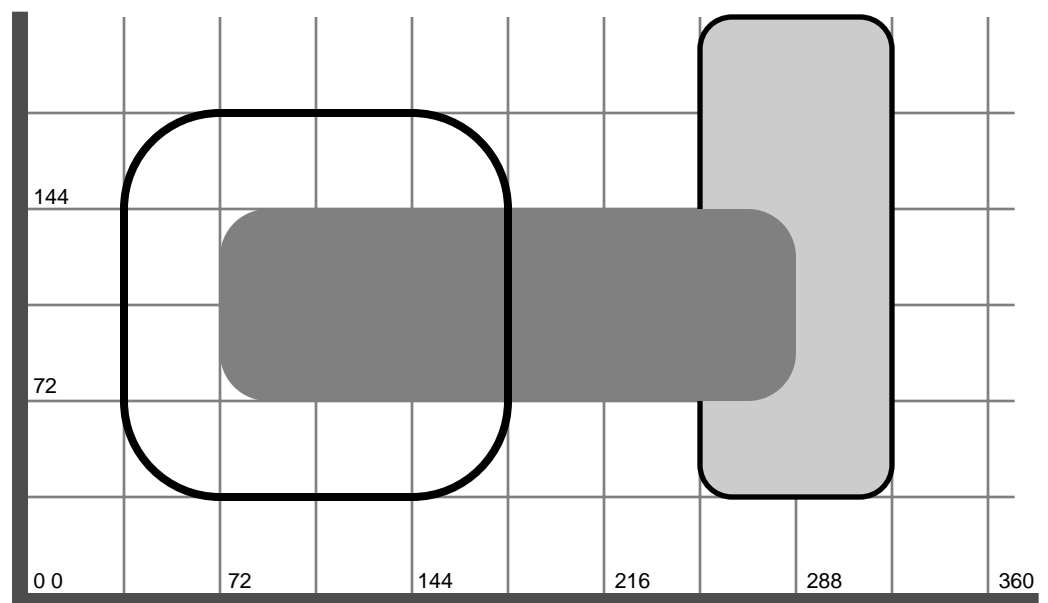
```
8 {pop} repeat
```

In the example above, a current point is made at `36 180` and a path is drawn to `x1 y1` and then continues to `x2 y2`. The corner of those two lines is then rounded by an arc with a radius of 72. The two points where the arc is tangent to the two lines create the new points `xt1 yt1` and `xt2 yt2`. The current point is now `xt2 yt2` and the line will continue from there. `x2 y2` will no longer exist.

The reason for the `8 {pop} repeat` is to remove the tangent points left on the stack. The two sets of tangent points of `xt1 yt1` and `xt2 yt2` are a by-product of the two `arcto`s and remain on the stack. This is a total of eight numbers that are not needed for anything at this time. In another situation, they may be taken advantage of. The operator `pop` removes the top most item on the operand stack, and since we have eight numbers to remove, `8 {pop} repeat` does this. It's the same as writing:

```
pop pop pop pop pop pop pop pop
```

The `arcto` operator can be used to make rectangles with rounded corners. An example follows.





6-8

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rounded_rect.eps
%%BoundingBox:34 34 325 218

/roundedRect      {
    /radius exch def /height exch def /width exch def
    width 2 div 0 moveto      % start in the middle
    width 0 width height radius arcto % 1st corner
    width height 0 height radius arcto % 2nd corner
    0 height 0 0 radius arcto      % 3rd corner
    0 0 width 0 radius arcto closepath % 4th corner
    16 {pop} repeat } def      % discard tangent points

gsave
    252 36 translate 2 setlinewidth
    72 180 12 roundedRect      % 72 wide, 180 high
    gsave .8 setgray fill grestore stroke
grestore

gsave
    72 72 translate .5 setgray
    216 72 18 roundedRect fill
grestore

gsave
    36 36 translate 3 setlinewidth
    144 144 36 roundedRect stroke
grestore

```



rounded_rect

The procedure `roundedRect` is designed to use three arguments to draw a rectangle with rounded corners. The first two are the `width` and `height` of the rectangle and the third is the `radius`. It would be used like this:

```
72 144 24 roundedRect stroke
```

The first line of `roundedRect` is

```
/radius exch def /height exch def /width exch def
```

This line is how the three arguments are “passed on” to the rest of `roundedRect` procedure. `72`, `144`, and `24` are on the operand stack with `24` on top when `roundedRect` arrives. All in one line it would be

```
72 144 24 /radius exch def /height exch def /width exch def
```

The first `exch` would switch `24 /radius`.

```
72 144 /radius 24 def /height exch def /width exch def
```

`/radius` would then be defined as `24`.

```
72 144 /height exch def /width exch def
```

The second `exch` would switch `144 /height` and so on as before.

The rest of the procedure uses `width` and `height` in various combinations with `0` and `radius` as arguments for the `arcto` operator.



type basics

Fonts are stored as outlines either in a printer's ROM, RAM, or an attached hard disk. The advantage of outlines is that the font can be scaled to any size without worry of ragged edges. The PostScript interpreter will fill in the appropriate number of dots. A bit-map would not provide the same flexibility. Each font is actually a dictionary of definitions for each letterform in the font. Earlier, in chapter 3, we defined the procedure `square` to be a series of drawing operations that drew a square. The procedure `square` was kept in a user dictionary in the PostScript interpreter while the program using it was run on the printer. In much the same way, each character of the alphabet has a series of drawing operations associated with its name in a font dictionary. For example, within every PostScript laser printer there is a font dictionary named Times-Bold. In that dictionary, `g` would be defined by a series of drawing operations that would create its outline.

7.1 *diagram of a letterform*

In figure 7-1 below, point **a** marks the location of the origin for a Times-Bold `g`. Point **b** represents both the width of the letterform and the new current point. Point **b** is the location of the origin of the next letterform to follow. It's very similar to the idea of making a current point, creating a path, and having the new current point at the end of the path. The black box identifies the smallest rectangle the `g` can fit in. It is known as the letterform's bounding box. The amount of space a letterform uses within a line of type, the distance between points **a** and **b**, is slightly more than its visible width.

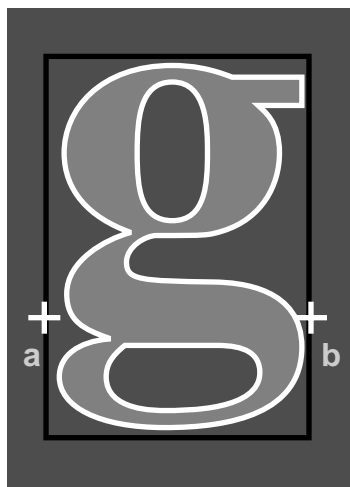
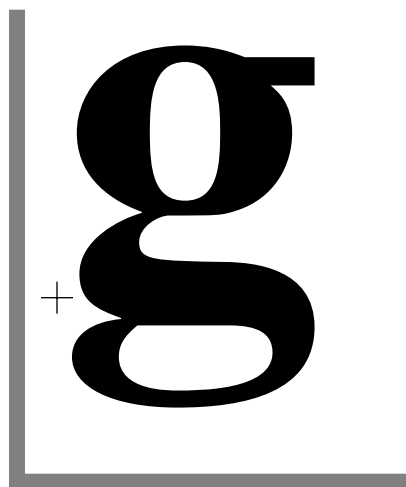


figure 7-1

7.2 *placing type on the page*

Positioning type on the page is accomplished in much the same way as a drawing action is started. A current point is first made using the `moveto` operator. The difference, however, is that the current point when made for type placement does not represent an actual point on a letterform, but represents its origin on its baseline. A letterform's origin is slightly left and on the baseline of the character. See the *g* in figure 7-1. The following example prints a *g*. The + marks the location of `18 72 moveto`, which is the character's origin.



7-1

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:type placement1.eps
%%BoundingBox:0 0 150 180
```

```
/Times-Bold findfont 200 scalefont setfont
```

```
18 72 moveto
(g) show
```

Here's an explanation of the program:

`/Times-Bold findfont` locates the font dictionary for Times-Bold.

`200 scalefont` scales the font to 200 point.

`setfont` makes Times-Bold the current font.

`18 72 moveto` establishes a current point. This point will be the character's origin and will also be on the baseline.

The characters to be set are identified by being contained within parentheses. The "(" and ")" are special characters used to identify what is known as a "string" or group of characters to be used for some purpose. In this case, they will be painted on a page.

`show` paints the characters held within the "(" and ")" in the current font with

the current color. In this case, the current font is Times-Bold and the current color is the default black.

In this example, the *g* will be painted with a 70% gray.



learn

7-2

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:type placement2.eps
%%BoundingBox:0 0 150 180
```

```
/Times-Bold findfont 200 scalefont setfont
```

```
18 72 moveto
.3 setgray
(g) show
```

After `show` in both preceding PostScript program examples, the new current point is located at the width of the character and on the baseline. See point **b** of figure 7-1. To illustrate this, notice how the line is drawn in the next example.

The beginning of the line is the current point located at the width of the character *g* and on the baseline. `100 0 rlineto` draws a path 100 points to the right of the current point. `stroke` then paints the path and initializes the current path.



7-3



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type placement3.eps
%%BoundingBox:0 0 225 180
    
```

```

/Times-Bold findfont 200 scalefont setfont
18 72 moveto
.3 setgray
(g) show
    
```

```

0 setgray
2 setlinewidth
100 0 rlineto stroke
    
```

When setting a series of characters to form a word, the character spacing is based on the font's metric file. The font metric file contains character widths. Each character in line to be set references the previous character.



figure 7-2

In figure 7-2 above, point **a** represents the first current point. Point **b** is the new current after the *g* is set. Point **b** therefore becomes the location for the origin of the character *a*. After *a* is set, point **c** becomes the new current point.

7.3 *various font strategies*

Since it is probable that a number of different fonts and sizes would be used on a page, following are a number of ideas for switching from one current font to another.

The first method would be to save the graphic state using `gsave` and `grestore`. In this way, there may be one font that you will continually return to in a PostScript program. An example would be:



learn

7-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type placement4.eps
%%BoundingBox:0 -4 346 126

/Helvetica findfont 170 scalefont setfont

0 0 moveto
(A) show

gsave
  /Times-Bold findfont 48 scalefont setfont
  .3 setgray
  10 10 moveto (aaaaaaaaaaaaaaaa) show
grestore

(B) show % no moveto, current point from A still active

gsave
  /StoneSerif-Bold findfont 48 scalefont setfont
  .5 setgray
  10 40 moveto (bbbbbbbbbb) show
grestore

(C) show % no moveto, current point from B still active

gsave
  /Helvetica-Bold findfont 48 scalefont setfont
  .8 setgray
  10 80 moveto (cccccccccccc) show
grestore

```

The first `gsave` saves three key values. The current font made by `/Helvetica findfont 170 scalefont setfont`, the current default value of black, and the new current point made after the `(A) show`.

The current font is then changed to `/Helvetica-Bold findfont 48 scalefont setfont`, the current value is changed to `0.3 setgray (70%)`, and a new current point is made with `10 10 moveto` for the row of `a`'s.

`grestore` restores the current font back to `/Helvetica findfont 170 scalefont setfont`, the current value back to black, and the current point left after the `(A) show`.

The whole process then gets repeated for the rows of `b`'s and `c`'s.

A second strategy is to give different font settings a name. For example,

```
/H18 {/Helvetica findfont 18 scalefont setfont} def
/H36 {/Helvetica findfont 36 scalefont setfont} def
/H72 {/Helvetica findfont 72 scalefont setfont} def
```

A third option is to define

```
/f {findfont exch scalefont setfont} def
```

and use it by writing this:

```
18 /Helvetica f
```

The third method is used in this example. Note how the placement of individual characters is accomplished.

TYPE



7-5

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:TYPE.eps
%%BoundingBox:0 0 190 50

/f {findfont exch scalefont setfont} def

68 /Helvetica-Bold f
   0 0 moveto (T) show

72 /StoneSerif-SemiboldItalic f
   (Y) show

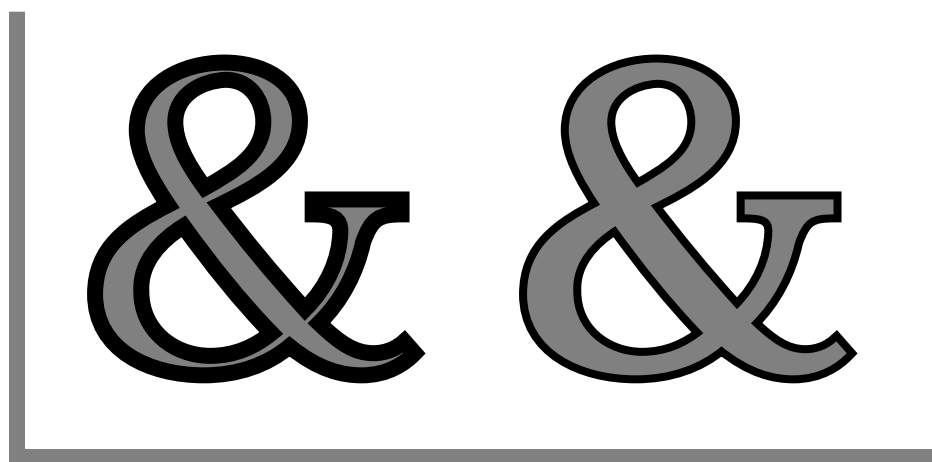
74 /Times-Bold f
   6 0 rmoveto (P) show   % move current 6 points right

68 /Helvetica-Oblique f
   -8 0 rmoveto (E) show  % move current -8 points left
```

Notice how `rmoveto` is used to kern the type right and left to correct the letterspacing. Kerning operators are explained in more detail in chapter 11, “advanced type.”

7.4 *stroking & filling type*

Instead of using `show` to paint the type as in all the previous examples, the letterform’s outline can be obtained and made into the current path. Then `gsave` and `grestore` can also be used to save the current path to both `stroke` and `fill` as was previously done to the square in section 3.3.



learn

7-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:&&.eps
%%BoundingBox:0 0 350 170

/StonSerif-Bold findfont 170 scalefont setfont

6 setlinewidth

18 36 moveto (&) true charpath
  gsave
    .5 setgray fill
  grestore
  stroke

180 36 moveto (&) false charpath
  gsave
    stroke
  grestore
  .5 setgray fill

```

Note the difference it makes if the type is stroked second or first.

`show` is replaced with `true charpath`. `true charpath` makes the character’s outline the current path. `charpath` is preceded with either `true` or `false`. According to the PostScript Language Reference (also known as the Red Book),

`true charpath` is suitable for filling or clipping, but not for stroking. The other choice, `false charpath`, is suitable only for stroking. In the PostScript program example above both are used, both seem to work correctly. Clipping is explained in section 9.4.

7.5 *font names*

If you are working on an Apple LaserWriter Plus or the later LaserWriter NT or NTII, there are thirty-five different font outlines that can be accessed in the printer's ROM. Following is a list of their names written as they would need to be written for a PostScript program. First, the basic group of fonts that are standard on every PostScript device:

`Courier`
`Courier-Bold`
`Courier-Oblique`
`Courier-BoldOblique`

`Times-Roman`
`Times-Bold`
`Times-Italic`
`Times-BoldItalic`

`Helvetica`
`Helvetica-Bold`
`Helvetica-Oblique`
`Helvetica-BoldOblique`

`Symbol`

These additional fonts are known as the Plus Set:

`AvantGarde-Book`
`AvantGarde-BookOblique`
`AvantGarde-Demi`
`AvantGarde-DemiOblique`

`Bookman-Light`
`Bookman-LightItalic`
`Bookman-Demi`
`Bookman-DemiItalic`

`Helvetica-Narrow`
`Helvetica-Narrow-Oblique`
`Helvetica-Narrow-Bold`
`Helvetica-Narrow-BoldOblique`

`NewCenturySchlbk-Roman`
`NewCenturySchlbk-Italic`
`NewCenturySchlbk-Bold`
`NewCenturySchlbk-BoldItalic`

`Palatino-Roman`

Palatino-Italic
 Palatino-Bold
 Palatino-BoldItalic

ZapfChancery-MediumItalic

ZapfDingbats

There are hundreds of fonts available from Adobe. A program to print most of the characters of a font (to see what they look like) follows. Substitute the font name, before the `findfont` operator with the font name to be set.

ABCDE
FGHIJ
KLMNO
PQRST
UVWXYZ
abcdefgh
ijklmno
pqrstuv
wxyz
1234567890



7-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:quickLook.eps
%%BoundingBox:30 125 180 425

/Times-Bold findfont 24 scalefont setfont

/left 36 def      % left margin
/newline {show currentpoint exch pop 30 sub
           left exch moveto} def

left 400 moveto   % location of first line

(ABCDE) newline  % newline acts like a carriage
(FGHIJ) newline  % on a typewriter
(KLMNO) newline

```

```
(PQRST) newline  
(UVWXYZ) newline  
(abcdefgh) newline  
(ijklmno) newline  
(pqrstuv) newline  
(wxyz) newline  
(1234567890) show  
  
.5 setgray 30 125 moveto  
150 0 rlineto 0 300 rlineto -150 0 rlineto  
closepath stroke
```




the repeat & for operators

The **repeat** operator, as its name implies, repeats something a specified number of times. The **for** operator counts by a specified increment from a given base number to a given limit. Both operators can be used to create similar results but they work in entirely different ways.

8.1 *repeat*

The syntax for **repeat** is:

```
number procedure repeat
```

For example,

```
4 {line} repeat is the same as writing
```

```
line  
line  
line  
line
```

where **line** may be a procedure defined earlier in the program to draw a line. The procedure can also be a PostScript operator. A common example is:

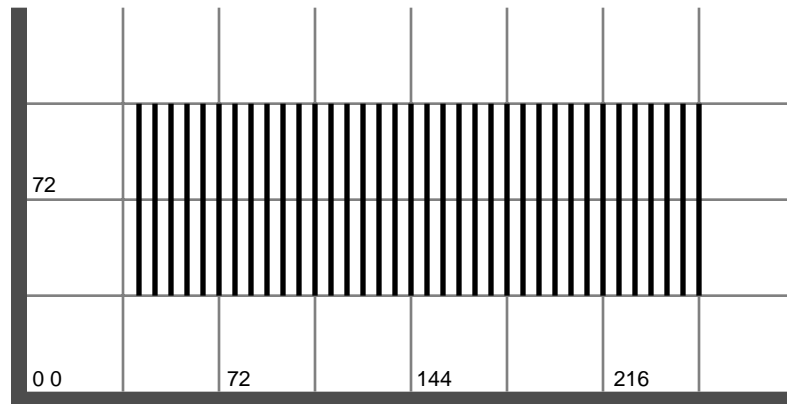
```
4 {pop} repeat
```

This would remove the top four items on the operand stack. You have seen this used before with **arcto** (section 6.4) and you'll see it used later with the **kshow** (section 11.5) operator.

In the following example, **repeat** is used to draw multiple lines.



8-1



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_1.eps
%%BoundingBox:36 36 252 108

/vline {0 0 moveto 0 72 rlineto stroke} def

36 36 translate
2 setlinewidth

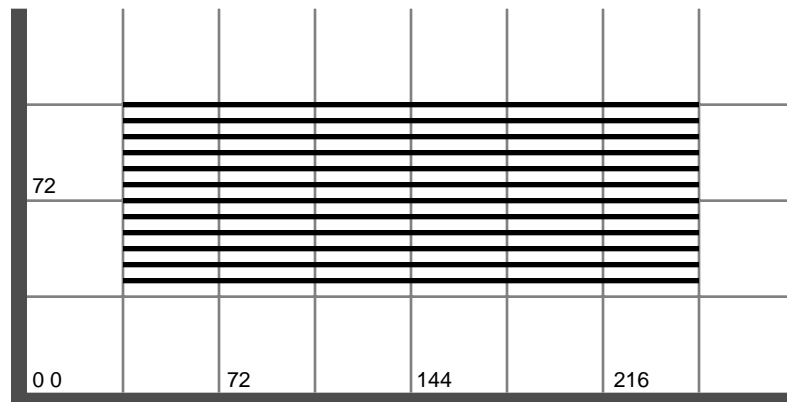
36 {6 0 translate vline} repeat

```

In the program above, `vline` defines a 72 point vertical line from the origin. The origin is then moved 6 points right thirty-six times with the `6 0 translate` before each use of the procedure `line`.

It's the same as writing `6 0 translate line` thirty-six times.

The next example is basically the same thing but moving up and drawing horizontal lines.



8-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_2.eps
%%BoundingBox:36 36 252 108

```

```

/hline { 0 0 moveto 216 0 rlineto stroke} def

36 36 translate

2 setlinewidth
12 {0 6 translate hline} repeat

```

To combine both programs to make a grid, at least the first `repeat` line should be within a `gsave` and `grestore` so that the second `repeat` starts at the proper location. Otherwise, the origin will be at 252 36 after the first `repeat`, throwing off the location of the second set of lines. Here is how that could be written:



8-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_1&2.eps
%%BoundingBox:36 36 252 108

/vline {0 0 moveto 0 72 rlineto stroke} def
/hline {0 0 moveto 216 0 rlineto stroke} def

2 setlinewidth
36 36 translate

gsave
    vline
    36 {6 0 translate vline} repeat
grestore

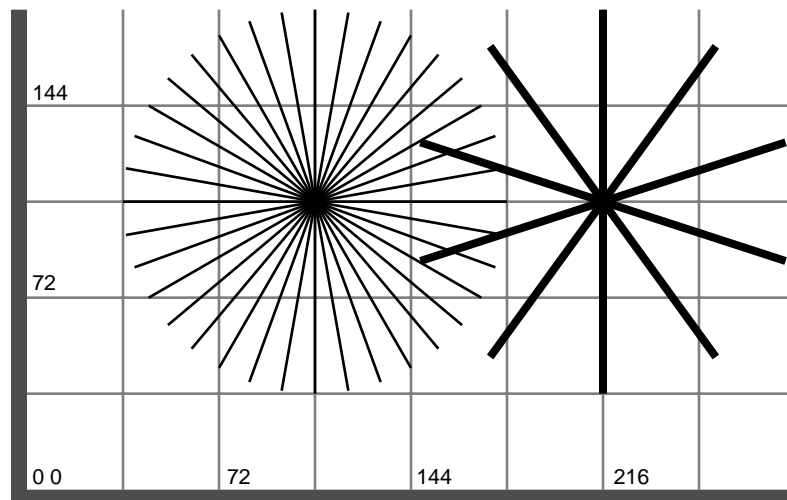
gsave
    hline
    12 {0 6 translate hline} repeat
grestore

```

The `repeat` can be used with the `rotate` operator. See the next example. The origin is first moved to 108 108 and then rotated thirty-six times at 10° intervals as each line is drawn. Then the origin is moved over 108 points and rotated ten times at 36° intervals as a thicker line is drawn. More on `rotate` in section 10.3.



8-4



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_3.eps
%%BoundingBox:36 36 288 180

/line {0 0 moveto 0 72 rlineto stroke} def

108 108 translate
36 {10 rotate line} repeat

3 setlinewidth 108 0 translate
10 {36 rotate line} repeat

```

8.2 *for*

The `for` operator counts up or down to a specified limit by a specified increment from an initial number. The syntax for `for` is:

```
initial increment limit procedure
```

Where:

- initial** is the first number of the count.
- increment** is the increment of the count (it can be positive or negative).
- limit** is the last number of the count.
- procedure** is the procedure executed after every count.

For example, `1 1 10 { } for` is the same as `1 2 3 4 5 6 7 8 9 10`.

Since there is no procedure supplied between the `{ }`, nothing is executed after each count from 1 to 10. If a procedure had been supplied, it would be executed after each count. For example,

```
17 1 1 6 {add} for
```

would be same as writing:

```
17 1 add 2 add 3 add 4 add 5 add 6 add
```


These two lines of code can be demonstrated further with the following program. Since you can't see what is happening inside the PostScript interpreter, this program will print the result of the two lines of PostScript code above. Parts of this program will be explained in more detail in section 8.3.



8-5



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:for_test.eps
%%BoundingBox:36 36 108 72

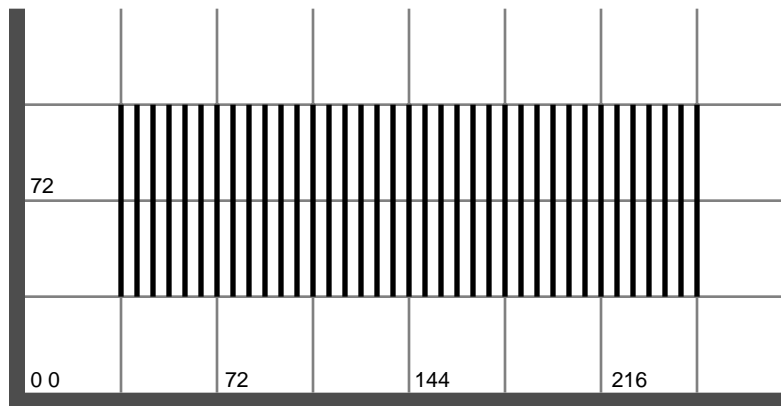
/Times-Bold findfont 24 scalefont setfont
/str 4 string def          % container for 4 characters

36 36 moveto
17 1 1 6 {add} for        % count from 1 to 6 adding each to 17
str cvs show              % put answer into str, convert into
                           % a string and show

/Times-Roman findfont 24 scalefont setfont
72 36 moveto
17 1 add 2 add 3 add 4 add 5 add 6 add
str cvs show

```

The `for` operator can be used to draw rows of parallel lines much like the `repeat` examples discussed in the previous section. However, it is accomplished by leaving numbers on the stack to be picked up later by the `moveto` operator. In the next example, the `for` operator counts by 6 up to 216. This puts or “pushes” 0, 6, 12, 18, 24, ..., 216 onto the operand stack before the execution of the procedure `line`, which was defined as `{0 moveto 0 72 rlineto stroke}`. Notice that there is only one number supplied for the `moveto` operator within the procedure definition. After each count of the `for` loop, there will be a new number on the stack to be used with `0 moveto`.



```

%!PS-Adobe-2.0 EPSF-1.2

```

```
%%Title:for_1.eps
%%BoundingBox:36 36 252 108

/line {0 moveto 0 72 rlineto stroke} def

36 36 translate
2 setlinewidth

0 6 216 {line} for
```

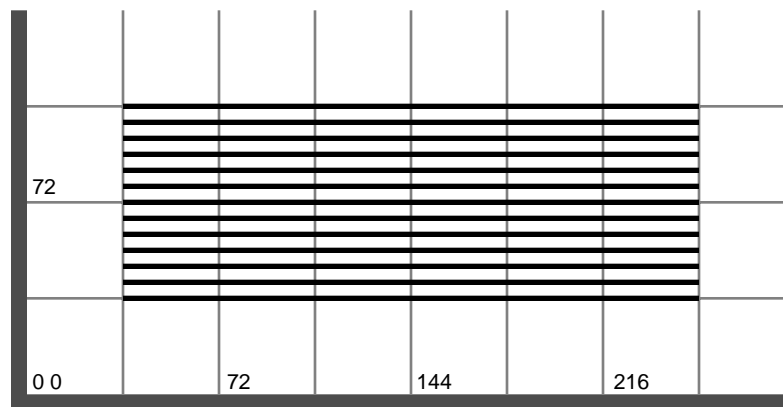
This would be same as writing:

```
/line {0 moveto 0 72 rlineto stroke} def

36 36 translate
2 setlinewidth

0 6 216 { } for % all x values are pushed onto the stack
0 line 6 line 12 line 18 line 24 line 30 line 36 line
42 line 48 line 54 line 60 line 66 line 72 line 78 line
84 line 90 line 96 line 102 line 108 line 114 line
120 line 126 line 132 line 138 line 144 line 150 line
156 line 162 line 168 line 174 line 180 line 186 line
192 line 198 line 204 line 210 line 216 line
```

A similar technique can be used to draw a series of horizontal lines. However, the `for` operator will be used to supply the `y` value needed by `moveto`. The `line` procedure therefore is defined as `{0 exch moveto 216 0 rlineto stroke}`. `exch` switches the top two items of the operand stack. Therefore, `0` trades places with the new number left on the stack. In this way, `0` can always be the `x` value and the new number is the `y`.



8-7

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:for_2.eps
%%BoundingBox:36 36 252 108

/line {0 exch moveto 216 0 rlineto stroke} def

36 36 translate
```

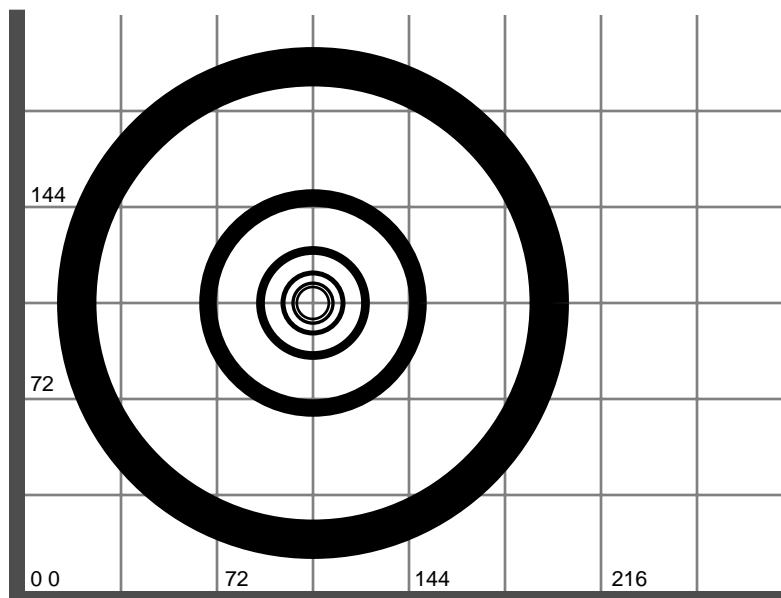
```
2 setlinewidth
0 6 72 {line} for
```

The following are two examples where `for` is used to supply one of the numbers for the `scale` operator. Since `scale` needs two values, for both the `x` and `y` scaling factor, `dup` is used to duplicate the top item on the stack. In the next example, `for` counts from 1 to 2.25 by increments of 0.25. `dup` duplicates each count and pushes the copy onto the stack. Therefore,

```
1 .25 2.25 {dup scale circle} for
```

is the same as

```
1 1 scale circle
1.25 1.25 scale circle
1.5 1.5 scale circle
1.75 1.75 scale circle
2 2 scale circle
2.25 2.25 scale circle
```



learn

8-8

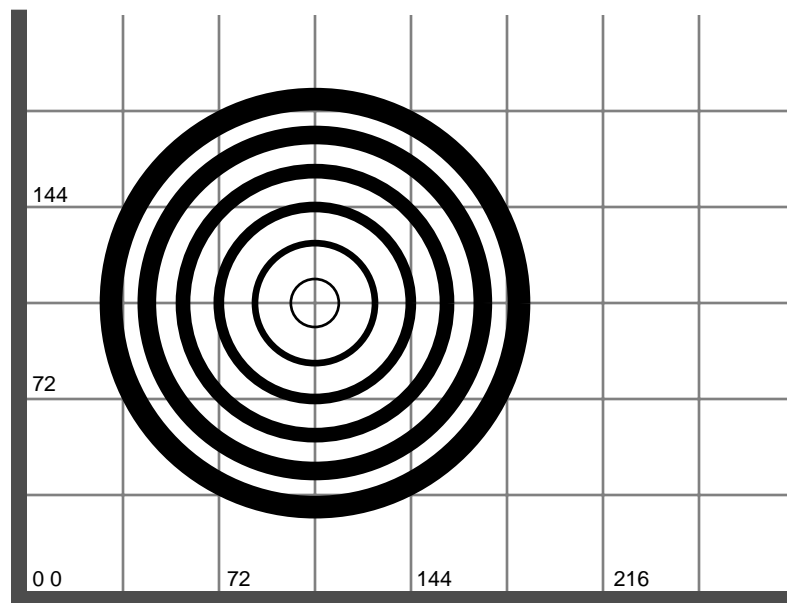
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:for_3.eps
%%BoundingBox:0 0 216 216
```

```
newpath
/circle{0 0 6 0 360 arc stroke} def
```

```
108 108 translate
```

```
1 .25 2.25 {dup scale circle} for
```

In the next version of the program, the `circle` procedure definition includes `gsave` and `grestore` and therefore changes the rate of scale the circle receives.



learn

8-9

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:for_3b.eps
%%BoundingBox:24 24 192 192

newpath
/circle      {0 0 9 0 360 arc stroke} def

108 108 translate

1 1.5 9 {gsave
         dup scale circle
        grestore} for

```

8.3 *using for & put with strings*

In the previous examples, the `for` operator was used to produce results that could have been written with the `repeat` operator. `for` has other uses besides being an alternative to the `repeat` operator. `for` can also be used with the `put` operator to manipulate strings and arrays. We will concentrate here on the manipulation of strings. The technique will be seen again later when we discuss creating fountains with the `image` operator in chapter 13. Its use on arrays is essentially the same.

In the following example, a portion of the alphabet is printed by a completely different technique than explained in chapter 7. There, the string to be printed was enclosed within () for the `show` operator to paint. Here, the characters of the string will be created by the `for` and `put` operators.

Where:

- string** is a group of characters.
- index** identifies a position within **string**; the first character is at index 0, the second character, is at index 1, and so on.
- integer** is the integer to be put into position **index** of **string**.

In the previous example, `0 1 43 {str exch dup put} for`, after the first count of `for` you would have

```
0 str exch dup put
```

`exch` switches the top two items on the stack, which gives us

```
str 0 dup put
```

`dup` duplicates 0, so we now have

```
str 0 0 put
```

`put` now puts 0 in index 0, or position 0 of `str`. The changing string `str` would look like this after the first count:

```
(0000000000000000000000000000000000000000000000000000000000000000)
```

After the second count it would be

```
(0100000000000000000000000000000000000000000000000000000000000000)
```

and so on until it is filled:

```
(012345678910111213141516171819202122232425 ... 40414243)
```

The next line, `36 36 moveto str show`, paints the string `str` beginning at the location of 36 36. Nothing appears till around 36 234 because there are no visible characters assigned to characters 0–31 in the ASCII chart and character 32 is the word space. The 33rd character is the exclamation point. See appendix A.

The last two lines of the program draw a vertical line marking where the string began at 36 36.

In the next example, `getinterval` is used to get a section of a string. In this case, it is used to obtain only the visible portion of the string created in the previous PostScript example. The syntax for `getinterval` is:

```
string index count getinterval returning substring
```

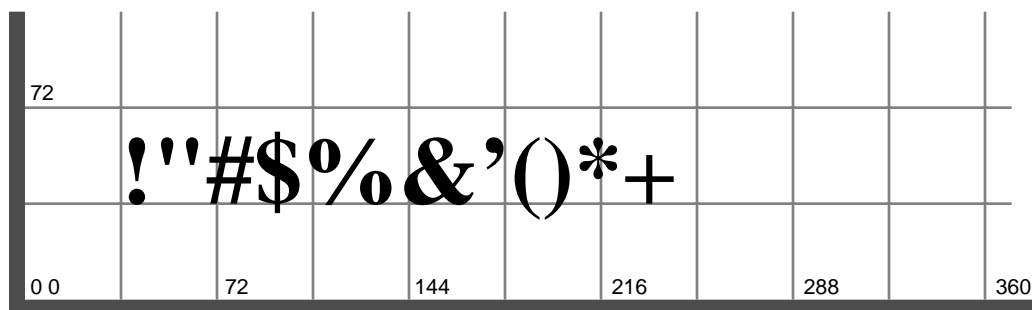
Where:

- index** is the starting point into the string.
- count** is the number of characters starting at **index**.

For example,

```
(abcdefg) 2 3 getinterval returns (cde)
```

In this example, `getinterval` is defined as the section of `str` starting at 33 plus the next 10 characters (a total of eleven characters).



8-11

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:for_5.eps
%%BoundingBox:36 33 252 70

/Times-Bold findfont 36 scalefont setfont
/str 44 string def

0 1 43 {str exch dup put} for

/piece {str 33 11 getinterval} def

36 36 moveto piece show

```




more on drawing

There is more to do with current paths than just **stroke** and **fill** them. The character of the **stroke** and **fill** can be changed in a variety of ways. The path can also be used as a kind of cookie cutter or mask to have images appear only within its boundaries.

9.1 *line endings & corners*

There are three ways to end a line. These are shown below. In all three, the black line represents the path and the gray is its width after stroking. The first one is called the butt cap. Whatever the line's width, it doesn't affect the line's length. The other two will increase the line's length by the thickness. The butt cap is the default setting.

0 **setlinecap** (the default)



1 **setlinecap** produces the round cap



2 **setlinecap** produces the square cap



There are also three possible corners.

0 **setlinejoin** the miter join (the default)

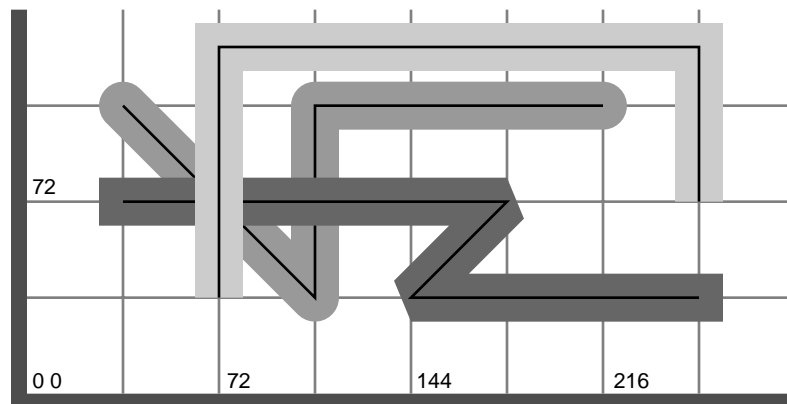
1 **setlinejoin** round join

2 **setlinejoin** bevel join

Examples of all of this follows. Again, the black line represents the path and the gray is its width after stroking.



9-1



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title: caps&joins.eps
%%BoundingBox:27 27 261 144

18 setlinewidth

1 setlinejoin 1 setlinecap    % bottom
  .6 setgray 36 108 moveto
  108 36 lineto 108 108 lineto 216 108 lineto stroke

2 setlinejoin 2 setlinecap    % middle
  .4 setgray 36 72 moveto
  180 72 lineto 144 36 lineto 252 36 lineto stroke

0 setlinejoin 0 setlinecap    % top & back to default
  .8 setgray 72 36 moveto
  72 130 lineto 252 130 lineto 252 72 lineto stroke

```

9.2 *dashed lines*

Dashed lines are created by using the `setdash` operator. The syntax is:

```
array offset setdash
```

Where:

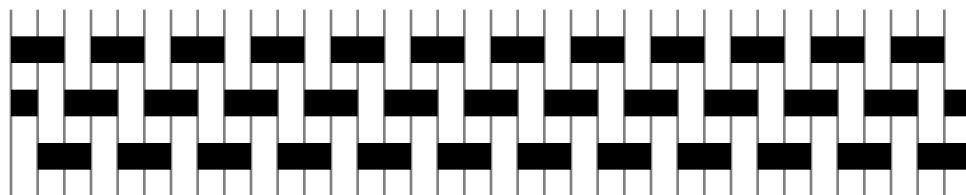
array is a collection of values that alternately specifies a stroke segment and a gap. There cannot be any negative values in the **array**.

offset shifts the line left or right.

This can be seen in the following example. Each line is 10 points wide and is the same alternating dash of a 20 point stroke and 10 point gap. What is different about each is the offset values of 0, 10, and -10 from top to bottom. There are additional examples in chapter 19, "library of examples."



9-2



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setdash_1.eps
%%BoundingBox:0 0 360 70

/v1 {0 0 moveto 0 70 lineto} def
gsave
  .5 setgray
  36 {10 0 translate v1} repeat stroke
grestore

10 setlinewidth

[20 10] 0 setdash      % top
0 55 moveto 360 55 lineto stroke

[20 10] 10 setdash    % middle
0 35 moveto 360 35 lineto stroke

[20 10] -10 setdash   % bottom
0 15 moveto 360 15 lineto stroke

```

9.3 *non-zero winding & the even/odd rules*

The non-zero winding and the even/odd rules determine what is filled or hollow within complex paths.

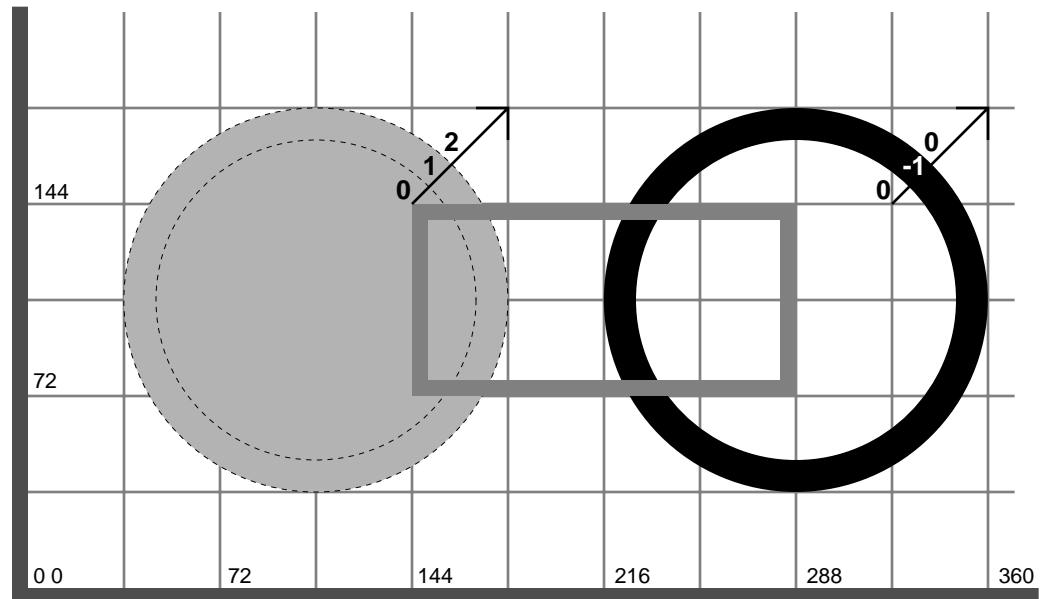
The most obvious application of the non-zero winding rule is in the drawing of letterforms that have holes (more precisely known as counters) within them. Letterforms such as *A*, *e*, and *O* are examples. If the paths of the counter are not drawn in the opposite direction of the main letterform shape, the counter shape will be filled.

In the following example, both circles were drawn with two paths. The intention is to make one the outside edge and the other the inside. In the first circle, both the inside and the outside paths are drawn in the same direction and filled. The paths were drawn again in black to show where they were drawn. Since they were drawn in the same direction, both circles filled. In the second circle, the inside and outside paths are drawn in opposite directions. Since this is the case, the smaller inside path becomes the inside edge of the circle.

The rectangle is another example of the inside path being drawn in the opposite direction of the outside path. The paths are used a second time and stroked to identify the paths.

A test for whether a given point will be inside or outside of a filled area can be done by drawing a line from that point to the outside. Starting with a value of zero, when

the line crosses a clockwise path add one, and when the line crosses a counter-clockwise path subtract one. If the result is zero, the point will be outside the shape, thus the name non-zero winding rule. Notice the lines in the two circles in the example below. On the left, the dotted line represents the path of both circles.



9-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:windingRule.eps
%%BoundingBox:16 34 364 182

% left circle; both circles drawn in the same direction
.7 setgray
newpath
108 108 72 360 0 arcn % clockwise
108 108 60 360 0 arcn % clockwise
fill

% right circle; inside circle drawn in opposite direction
newpath
288 108 72 360 0 closepath arcn % clockwise
288 108 60 0 360 closepath arc % counterclockwise
fill

% rectangle counterclockwise drawing
.5 setgray 144 72 moveto
288 72 lineto 288 144 lineto 144 144 lineto closepath
% clockwise drawing
150 78 moveto
150 138 lineto 282 138 lineto 282 78 lineto closepath
fill
    
```

The even-odd rule is similar to the non-zero rule in that it too helps to determine what parts are and are not filled within a complex path. In figure 9-1, the letterform *g* is an example. The counters of the first *g* are drawn in the opposite direction of the outside shape. In the second and third *g*, the counters are drawn in the same

learn

9-4

direction. We get the expected results in the second *g*, but the third appears correct. The reason is the third *g* is filled by using `eofill`, an alternate for `fill`. If `eofill` had been used on the first *g*, the results would have been the same and been correct.

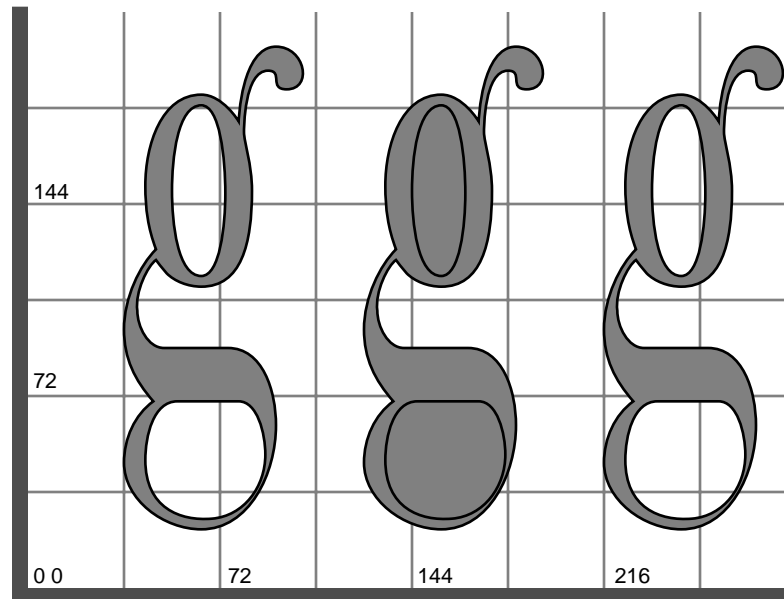


figure 9-1

The even-odd rule uses the same test line, crossing the paths as it leaves the shape. However, one is added every time a path is crossed. In the first and third *g* above, two paths are crossed, an even number. It doesn't matter what direction the paths are drawn in. In more complex paths, the results are not as predictable as in figure 9-2.

In figure 9-2, the first shape is filled by `fill`, the second by `eofill`.

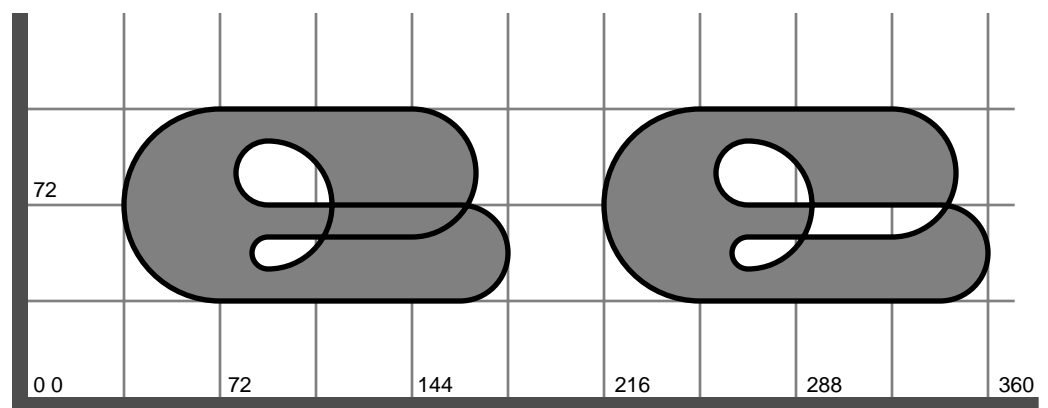


figure 9-2

Other examples of the non-zero winding and the even/odd rules can be found in chapter 19, "library of examples."

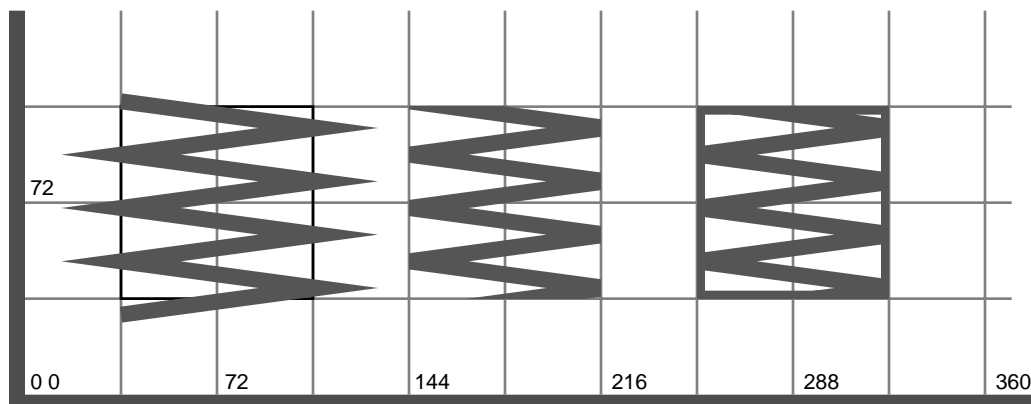
9.4 clipping

Clipping forces all subsequent painting to appear only within the boundaries of a

path. An example of a clipping area are the margins that form the printable area on a laser printer's page. If something is drawn beyond that boundary, it's clipped or cropped. All the path construction operators such as `lineto`, `arc` and `curveto` can be used to create a clipping path.

The clipping action should be placed between a `gsave` and `grestore` to prevent it from affecting all that follows in the program. The `gsave` and `grestore` can be considered in a sense an on/off switch for the `clip`. All graphics after the `clip` and the next `grestore` will be cropped by the clipping path.

`clip` should also be followed by the `newpath` operator. Otherwise, the clipping path may get stroked or filled. In the example below, the first box and zigzag are for reference. No clipping is taking place. In the second position, the `box` clipping path followed by `newpath` clips the zigzag. The third part shows what happens when `newpath` is left out. The clipping path is also stroked and half its width is clipped away. `newpath` initializes the clipping path from painting but does not affect the clip.



learn

9-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:clip_3.eps
%%BoundingBox:12 30 326 115

/box {0 0 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
      closepath} def

/zag {6 setlinewidth .333 setgray
      0 74 moveto 74 -10 rlineto -74 -10 rlineto
      74 -10 rlineto -74 -10 rlineto
      74 -10 rlineto -74 -10 rlineto
      74 -10 rlineto -74 -10 rlineto} def

36 36 translate
box stroke
zag stroke
1 setlinewidth 0 setgray

gsave
  108 0 translate
  box clip newpath
  zag stroke
grestore

```

```

gsave
  216 0 translate
  box clip
  zag stroke
grestore

```

Other examples of clipping and an example of the related `eoclip` can be found in the library of examples chapter.

9.5 *setscreen*

Gray values are simulated on laser printers by arranging the dots made by the printer into patterns that the eye sees as different grays. The dots of the printer are grouped together to form cells. Within a given area, if half the dots are used in each cell, a 50% gray is produced.

The `setscreen` operator controls the dot shape of the cell and the frequency (the number of cells per inch) for the current color. The default for the NeXT laser printer, for example, is a round dot at 60 cells per inch at 0°. The syntax for `setscreen` is:

```
frequency angle procedure setscreen
```

Where:

frequency is the number of cells per inch.

angle is the angle of the screen in degrees.

procedure controls the dot shape of the cell. A number of different shapes are possible including square, oval and line dots. The size of the dot will depend on the value of the current color.

If the round dot default were changed from 60 to 10 lines per inch and the angle from 0° to 45°, it would be written as the following line of PostScript:

```
10 45 {dup mul exch dup mul add 1.0 exch sub} setscreen
```

It would look like figure 9-3.



figure 9-3

The `setscreen` operator assigns a priority value to each pixel of each cell of the halftone. The priority determines the arrangement of pixels for each cell. The

amount of pixels used will depend on the gray value needed. The darker the gray, the more pixels used. If a 400 dpi laser printer were set to have 40 cells per inch, each cell would be 10x10 or contain 100 pixels.

If you could see one of those cells and chart it, it would look like figure 9-4. The 100 pixels are divided into an x y axis with its center as the origin or 0 0. Each pixel has an x y value. The upper right is always 1 1, the lower left is always -1 -1, and so on. If the cell were 20x20, the intermediate x y locations would adjust accordingly. The `setscreen` procedure uses these x y locations to determine and assign each pixel's priority.

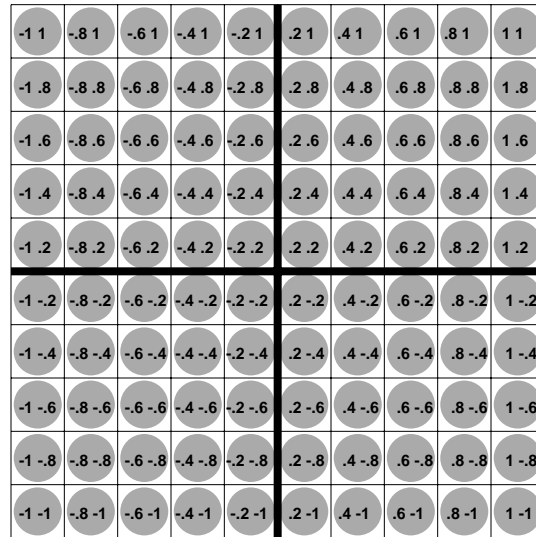


figure 9-4

The easiest `setscreen` procedure to use as an example is `{pop}`. `pop` discards the top number on the stack. If the pair of numbers is `.2 .4`, the `.4` would be the one on top of the stack because it would be the last number received. `pop` would discard it leaving `.2`. Looking at the chart of the cell, `pop` would discard all the y values, leaving each pixel's x as the pixel's priority number. The left row therefore will all be `-1`, the next row are all `-.8`, and so on down each row (see figure 9-5).

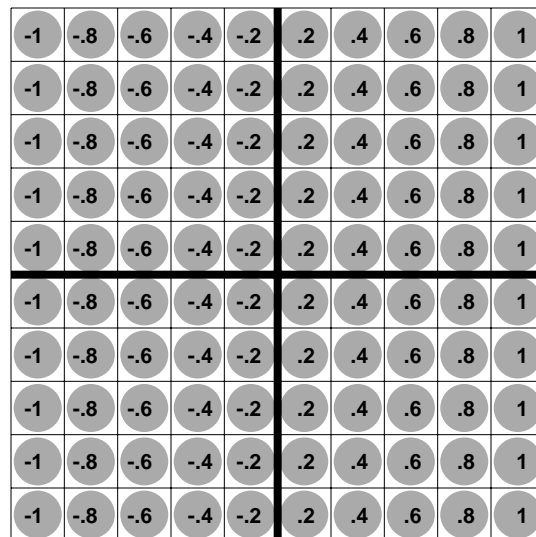
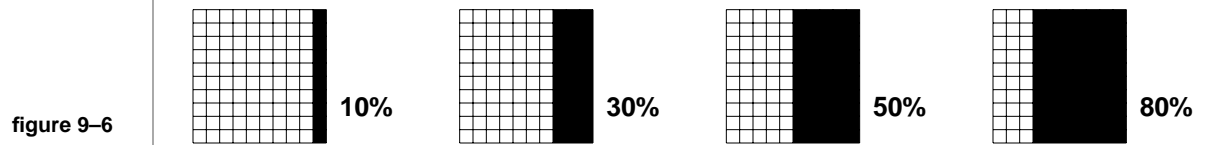
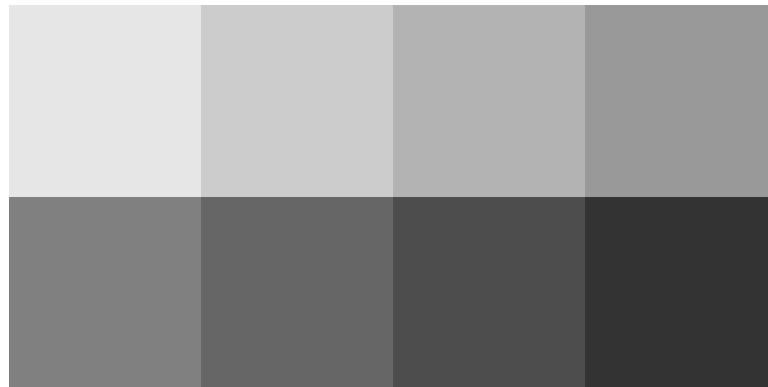


figure 9-5

Using this priority system, different gray values will fill as in figure 9-6.



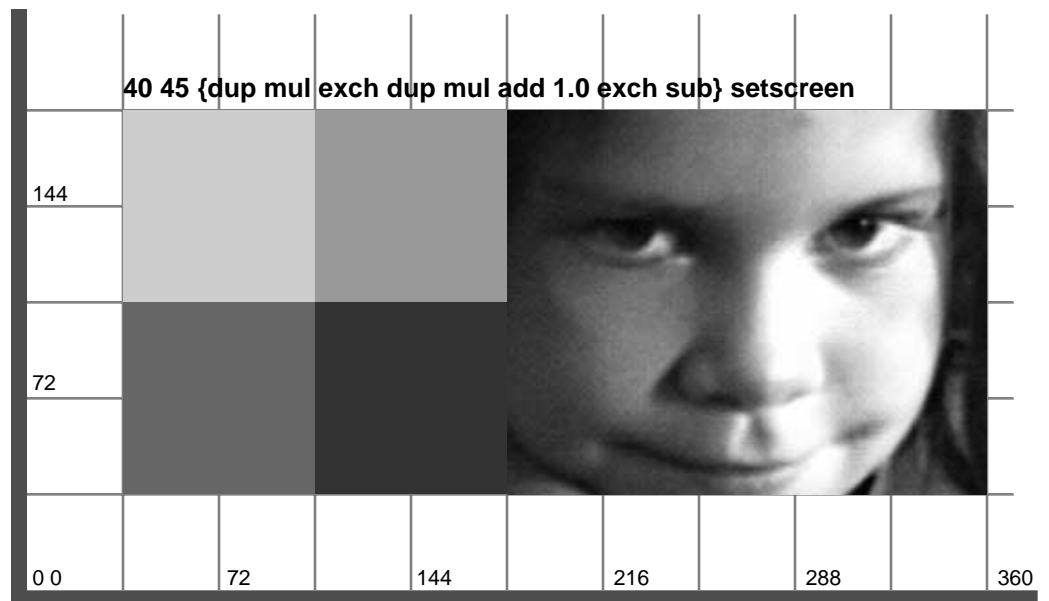
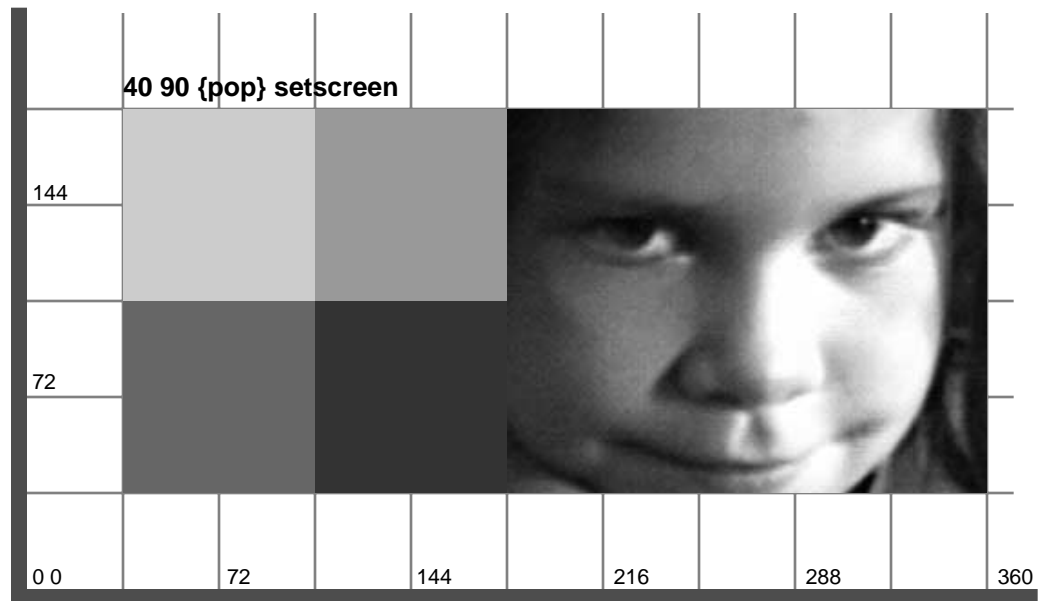
Actual size, it will look like this:



Other procedures will cluster the pixels in the center as a round, oval, or square dot. There are a number of possible patterns available. In addition, the `setscreen` will affect halftone pictures.

9.6 *setscreen with halftones*

The `setscreen` operator will affect the imaging of all gray values, including scanned gray scale pictures. `setscreen` does not affect 1-bit pictures. The dot pattern in this case is already determined. In the two examples below, squares painted with a 20%, 40%, 60%, and 80% black are next to a scanned gray scale picture. Other `setscreen` examples can be found in chapter 19, “library of examples.”



9.7 *pathforall*

pathforall retravels a current path and executes one of four procedures at each instance of a *moveto*, *lineto*, *curveto*, and *closepath* respectively.

The syntax for *pathforall* is:

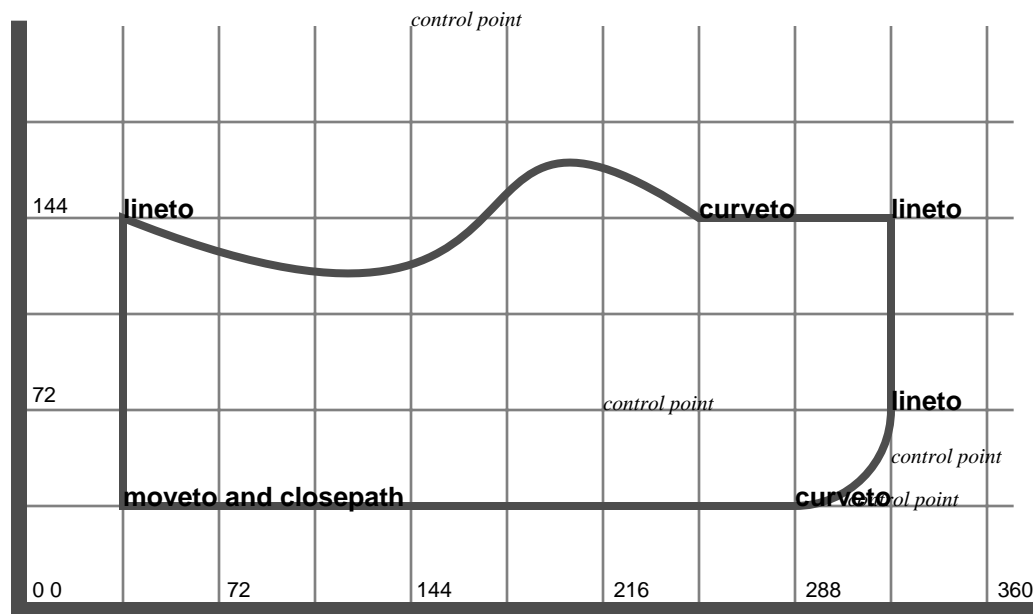
```
movetoProc linetoProc curvetoProc closepathProc pathforall
```

Where:

movetoProc	returns the x y location of the moveto then executes the movetoProc procedure at every instance of a moveto encountered.
linetoProc	returns the x y location of the lineto then executes the linetoProc procedure at every instance of a lineto or rlineto encountered.
curvetoProc	returns the x y x¹ y¹ x² y² of the curveto and executes the curvetoProc procedure at every instance of a curveto or rcurveto encountered. If the curve is made by the arc , arcn , or arcto operators, it will be converted to the curveto equivalent.
closepathProc	executes the closepathProc procedure at every instance of a closepath .

The **x y** coordinates returned are in user space. The four procedures can perform any operation desired and can use the coordinates pushed on the stack. If the values are not used, they should be cleared from the stack with the **pop** operator.

In the following example, I use **pathforall** to print the PostScript operators used up to that point. The location of each label is from the coordinates pushed onto the stack by **pathforall**. Note in the **start** procedure the **currentpoint** operator. This pushes the **x y** coordinates of the current point after the (**moveto**) **show** onto the stack to be used later by the **fin** procedure.



9-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:pathforall_1.eps
%%BoundingBox:34 34 360 220

/start{moveto (moveto) show currentpoint} def
/line {moveto (lineto) show} def
/curve{moveto (curveto) show
        /Times-Italic findfont 8 scalefont setfont

```

```

moveto (control point) show
moveto (control point) show
/Helvetica-Bold findfont 10 scalefont setfont} def
/fin {moveto( and closepath) show} def

/path {36 36 moveto
36 144 lineto
216 72 144 216 252 144 curveto
72 0 rlineto
288 72 36 0 270 arcn
closepath} def

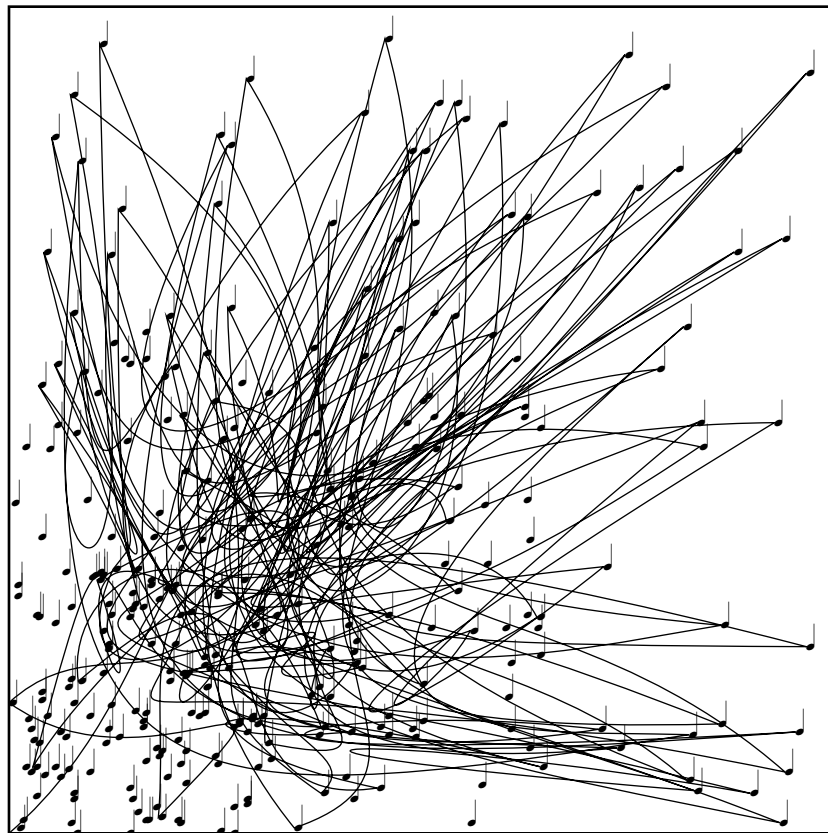
/Helvetica-Bold findfont 10 scalefont setfont

3 setlinewidth 0.3 setgray
path stroke

0 setgray
path
{start} {line} {curve} {fin} pathforall
newpath

```

The following example uses `pathforall` in the same way as the design *Blast*, found in the beginning of chapter 2. All the music notes are located at the 3 pairs of x y for the `curveto` operator generated by `pathforall`.





9-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:pathBlast.eps
%%DocumentFonts:Sonata
%%BoundingBox:0 0 310 310

0 0 moveto 310 0 lineto 310 310 lineto 0 310 lineto
closepath stroke

/a {100 rand exch mod} def
/b {100 rand exch mod} def
/c {200 rand exch mod} def
/d {200 rand exch mod} def
/e {300 rand exch mod} def
/f {300 rand exch mod} def

/note {moveto (q) show} def

.5 setlinewidth
173417 srand
/Sonata findfont 10 scalefont setfont

0 0 moveto
100 {a b c d e f curveto} repeat
{pop pop}{pop pop}{note note note}{} pathforall stroke

```




the CTM

CTM is the acronym for the Current Transformation Matrix. The CTM can be thought of as the coordinate system. A 1 inch square can be defined and expected to print as a 1 inch square. It will print as a 1 inch square as long as the CTM has not changed. If the coordinate system or matrix is doubled in size, the square will appear to be twice its intended size when used. The `scale` operator is one of several PostScript operators used to transform the matrix in some way. The most common CTM operators are:

<code>translate</code>	moves the location of the origin or 0 0.
<code>scale</code>	changes the size of the coordinate system or matrix.
<code>rotate</code>	rotates the coordinate system or matrix.
<code>concat</code>	can perform the <code>translate</code> , <code>scale</code> , and <code>rotate</code> transformations all in one command.

In particular, understanding the `concat` operator will be necessary before getting into chapters 11, 12, 13, and 14. The concepts covered here will easily apply to those four chapters.

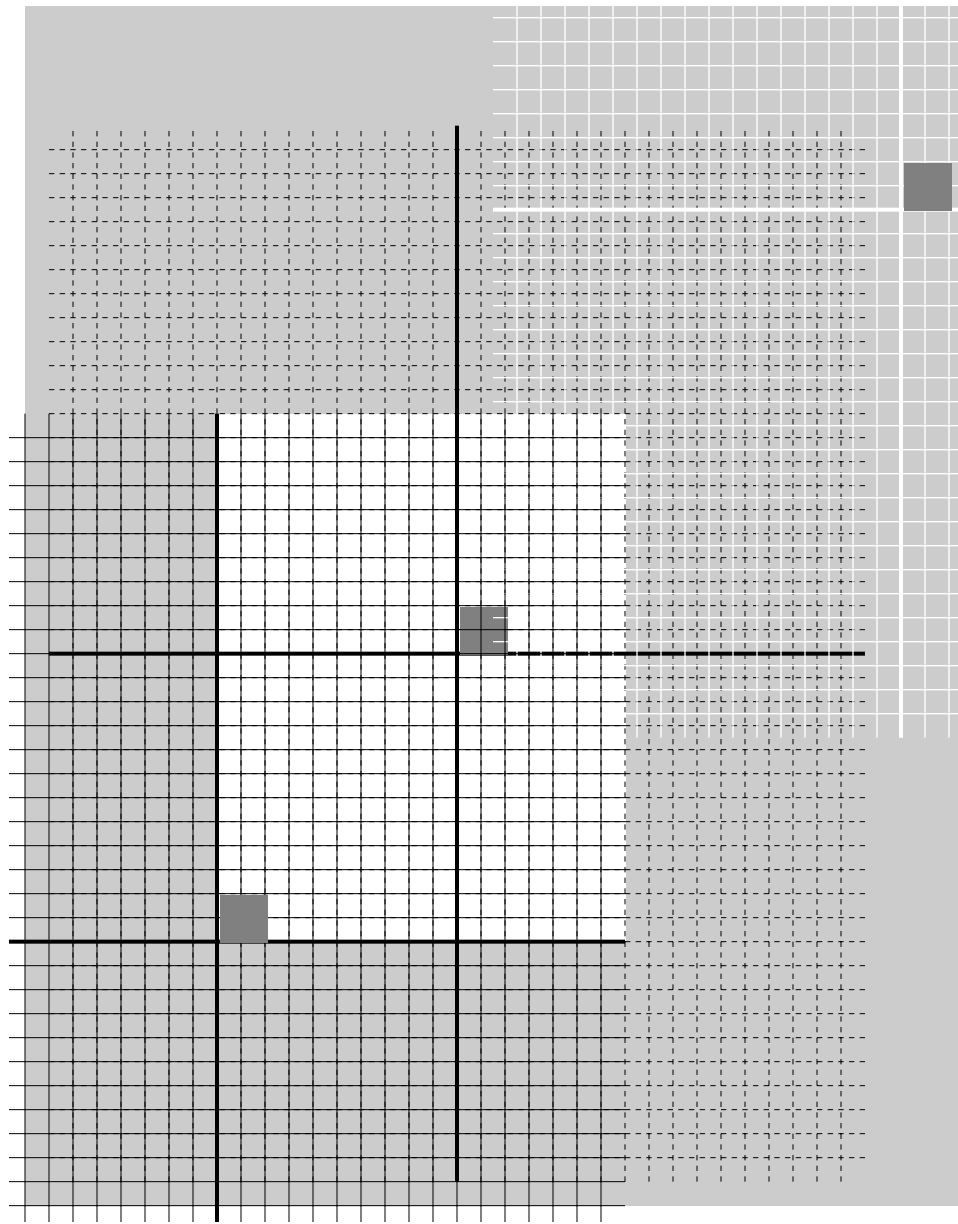
Learning these various transformations can provide a number of interesting visual opportunities.

10.1 *translation of the origin*

`translate` was introduced briefly in chapter 3, section 3.4. There it was used to position a square at different locations on the page. Often, using `translate` to move around the page is the best way to organize a design. Having a consistent reference point to work from can be very handy. It facilitates procedures being used in one design to be used in another.

It is important to remember that `translate` transforms the CTM, which is part of the graphic state. Consider this example. Three squares are placed, the second and third using `translate` to move the origin to determine their placement. Note that `gsave` and `grestore` are not used to save the graphic state between placement of squares.

The white rectangle represents the page, the different grid patterns represent locations of the CTM by using `translate`. As you can see, you could have an image placed off the page. When this program is printed, as far as you know, the third square doesn't exist.



10-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:translate_1.eps
%%BoundingBox:0 0 1098 1170

/square      { 0 0 moveto 72 0 rlineto 0 72 rlineto
              -72 0 rlineto closepath fill} def

.5 setgray
square
360 432 translate square
666 666 translate square
    
```

By using `gsave` and `grestore` to save and restore the original CTM between

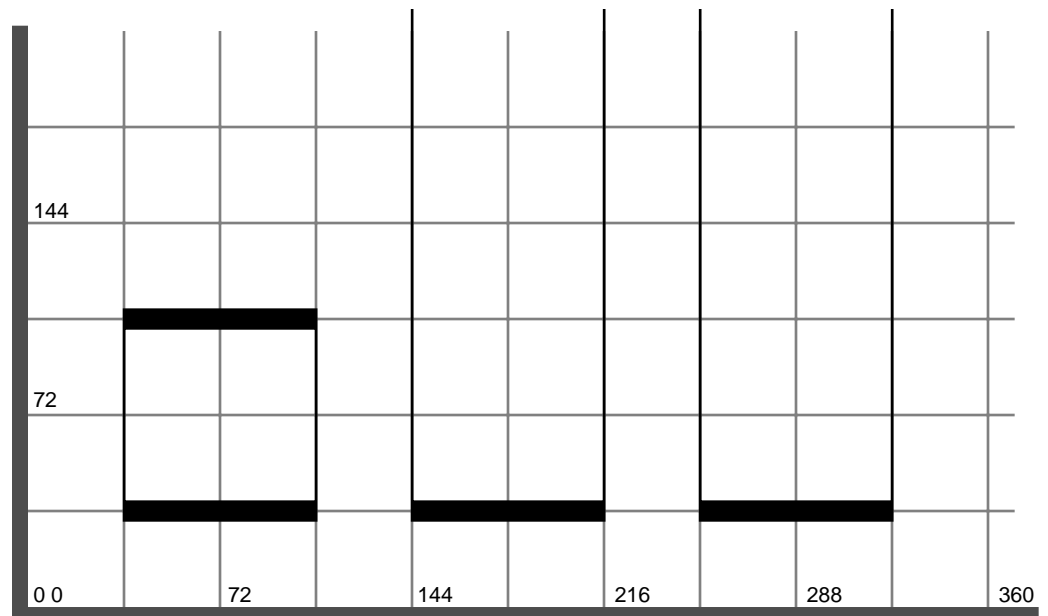
transformations made by the `translate` operator, graphics can more predictably be placed on the page.

10.2

scale

`scale` expects two arguments, an `x` and `y` value where 1 = the existing size, 0.5 = half the size, and 2 is twice the size. `0.5 2 scale` would therefore compress the x axis by half and double the y axis.

`scale` was introduced briefly in section 3.5. There the `scale` operator was used to make squares of various sizes. As mentioned there, you can get yourself into trouble by not carefully noting where `scale` is sequenced with other operators and obtain unexpected results. Study the following example and note on which line `1 8 scale` is entered.



10-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:scale_seq.eps
%%BoundingBox:30 30 330 288

/square {0 0 moveto 72 0 lineto 72 72 lineto 0 72 lineto
closepath } def

gsave          % first
  36 36 translate
  square
  1 8 scale    % after square path & translate
  stroke
grestore

gsave          % second
  144 36 translate
  1 8 scale    % after translate, before square path
  square

```

```

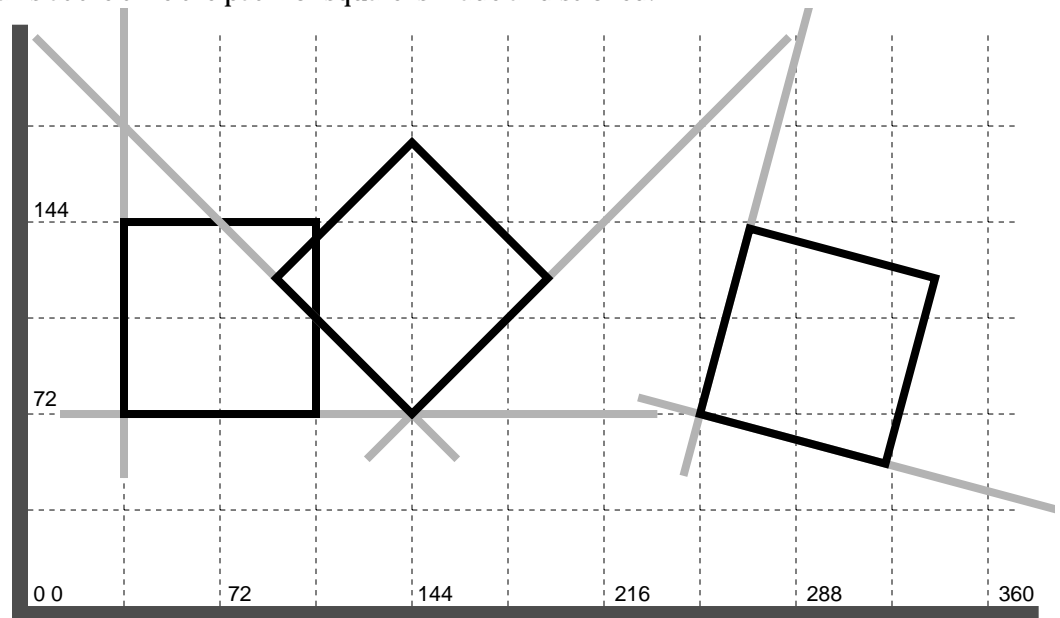
        stroke
    grestore

    gsave                % third
        1 8 scale      % before translate, square path, & stroke
        252 4.5 translate    % 8 x 4.5 = 36
        square
        stroke
    grestore

```

10.3 *rotate*

As you might expect, `rotate` rotates the CTM. It expects one number representing a degree of rotation. A positive number is a counterclockwise rotation and a negative number is a clockwise rotation. 0° is at the three o'clock position. In the example below, the first square is normal, the second is `45 rotate` and the third is `-15 rotate`. The crossing gray lines represent the change of location of the x axis at the time the path for a square is made and stroked.



learn

10-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rotate_1.eps
%%Creator:John F Sherman
%%CreationDate:June 1990
%%BoundingBox:34 70 360 180

/square {0 0 moveto 72 0 lineto 72 72 lineto 0 72 lineto
closepath stroke} def

3 setlinewidth
gsave
    36 72 translate
    square
grestore

```

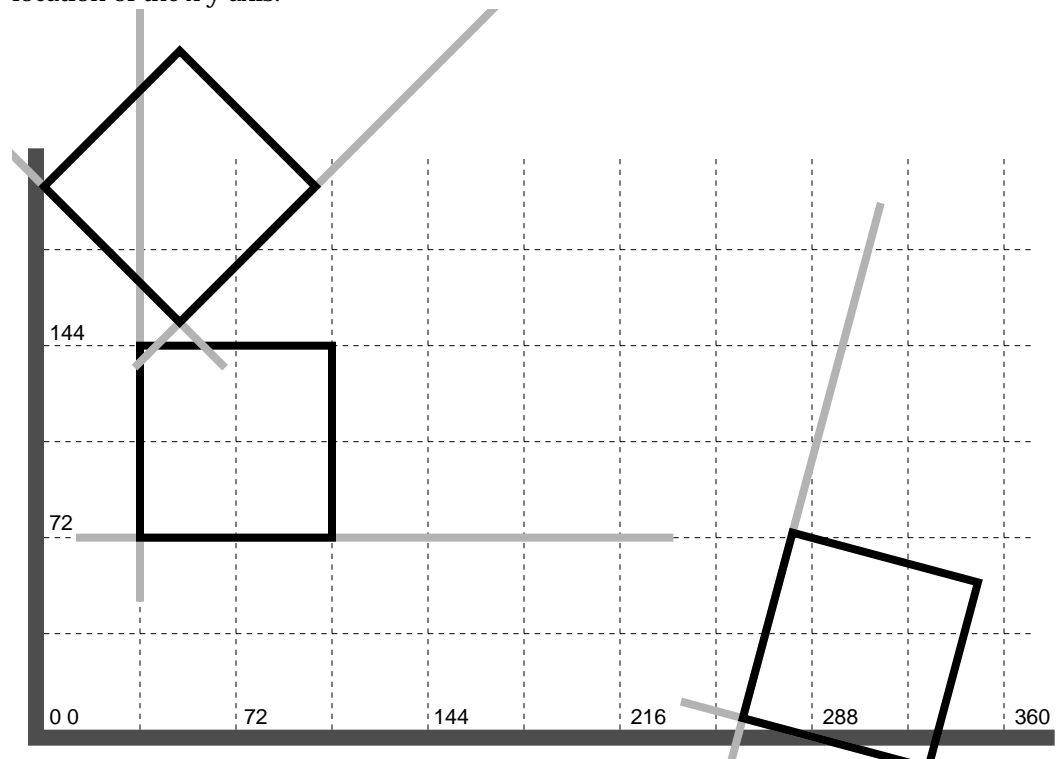
```

gsave
  144 72 translate
  45 rotate
  square
grestore

gsave
  252 72 translate
  -15 rotate
  square
grestore

```

As was the case with `scale`, the sequence in which `rotate` is used is very important. In the next example, `rotate` is used before `translate` instead of after. The same degrees of rotation and translation are used as in the previous example. Note the difference. Earlier we moved the origin then rotated, now we rotate then move the origin. Again the crossing gray lines represent the change of location of the x y axis.



learn

10-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rotate_2.eps
%%BoundingBox:0 -12 360 270

```

```

/square {0 0 moveto 72 0 lineto 72 72 lineto 0 72 lineto
closepath stroke} def

```

```

3 setlinewidth
gsave
  36 72 translate

```

```

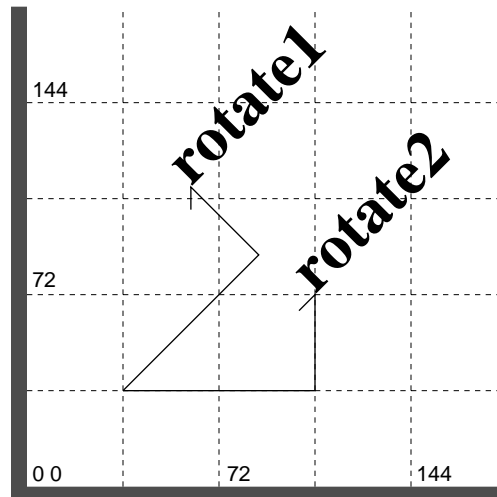
square
grestore

gsave
  45 rotate
  144 72 translate
square
grestore

gsave
  -15 rotate
  252 72 translate
square
grestore

```

It also makes a difference whether the current point for a graphic is made before or after the rotation. Note when `rotate` is used in the example below. A current does not move with the rotation of the CTM.



10-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rotate_3.eps
%%BoundingBox:54 72 162 180

/Times-Bold findfont 24 scalefont setfont
36 36 translate
.5 setlinewidth

gsave      % moveto after rotation of CTM
  45 rotate
  72 36 moveto (rotate1) show
grestore

gsave % moveto before rotation, but (rotate2) show is after
  72 36 moveto
  45 rotate
  (rotate2) show
grestore

```

10.4

concat

The manipulation of the current transformation matrix using `concat` will also be useful in understanding the `makefont`, `image`, and `imagemask` operators explained in later chapters.

The matrix can be thought of as an elastic grid that scales all graphics to the page. If the CTM were doubled, everything would print twice the intended size. The `translate`, `scale`, and `rotate` operators change the CTM, and their three combined operations can be done at once with the `concat` operator. `concat` is more powerful but it's less intuitive than the other three.

In figure 10-1 below, a circle is drawn on a grid representing the CTM.

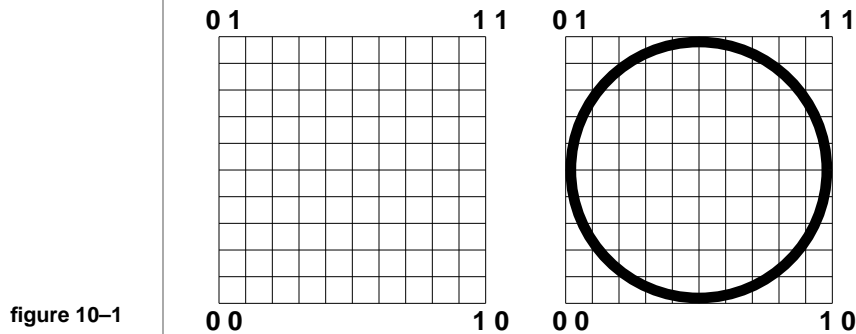


figure 10-1

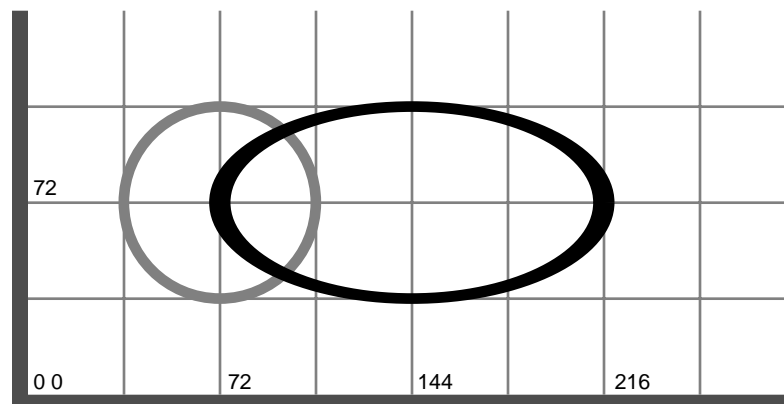
If that grid was scaled to twice its normal size on the x axis, the width of the circle would also be twice its normal size. This could have been done with

```
2 1 scale
```

but it can also be done with

```
[2 0 0 1 0 0] concat
```

The first number of the array scales the x axis and the fourth number scales the y axis. If we wanted the height of the circle to be doubled, the matrix array would be `[1 0 0 2 0 0]`. Referring to the first and fourth numbers, 1 is 100% or actual size, .5 is 50% and 3 is 300%. If there has already been a scale applied to the CTM, the new transformation will be in addition to the existing CTM. An example of the scaled circle with `concat` follows:





10-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:circle by2.eps
%%BoundingBox:34 34 220 110

.5 setgray
4 setlinewidth
newpath 72 72 36 0 360 arc stroke

0 setgray
[2 0 0 1 0 0] concat
newpath 72 72 36 0 360 arc stroke

```

There are other transformations possible with `concat`. The second and third numbers skew the CTM up or down and left or right. Figure 10-2 is an example of skewing the x axis. The second number controls the angle of the x axis and the third controls the angle of the y axis. Figure 10-2 is an example of `[1 1 0 1 0 0]` `concat`. The 1 in the second position of the matrix array skews everything up 45°.

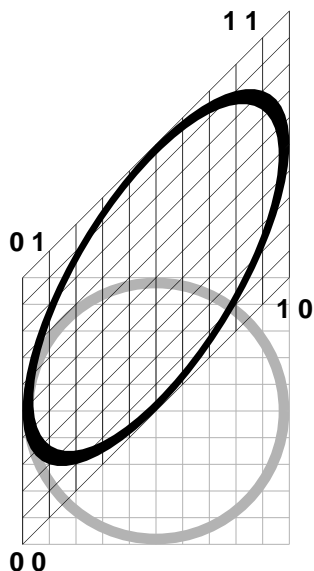


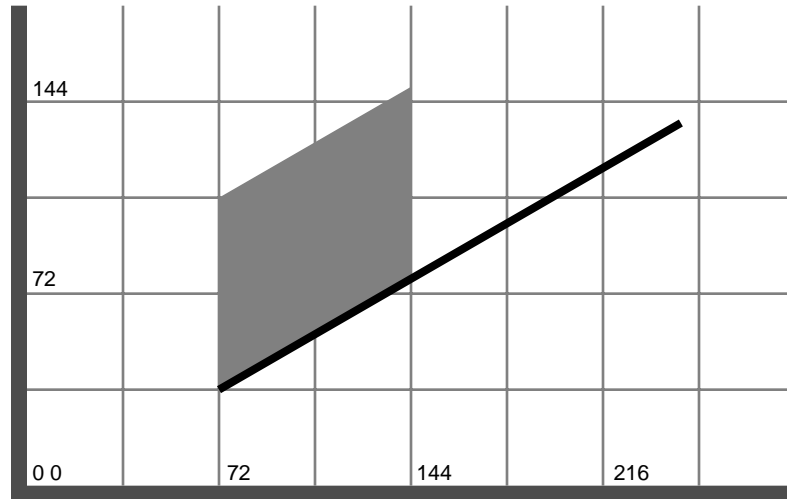
figure 10-2

Had the matrix been `[1 -1 0 1 0 0]` `concat`, the shift would have been down 45°. Unfortunately, as you can see, the degree of the desired shift is not what is entered into the matrix. There is a calculation based on the desired angle θ (degree) that needs to be done for the second and third numbers. The tangent of θ is what is required for the second and third positions of the matrix array. Appendix B contains a table of tangent values for each degree for convenient reference.

To rotate the x axis counterclockwise 30°, `tan30`, which equals 0.5774, is entered as the second number of the matrix array. In the following example, the square is skewed 30° by the `concat` operator. The black line is rotated 30° by using `rotate` as a reference.



10-7



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat_1.eps
%%BoundingBox:72 36 252 150

72 36 translate

gsave      % gray box
    .5 setgray
    [1 0.5774 0 1 0 0] concat
    0 0 moveto 0 72 lineto 72 72 lineto 72 0 lineto
    closepath fill
grestore

gsave      % black line
    3 setlinewidth
    30 rotate
    0 0 moveto 200 0 lineto stroke
grestore

```

If both axes were skewed the same degree and direction, it would appear that we had made the equivalent of a 30° rotation. Not quite, because

```
[1 .5774 -.5774 1 0 0] concat
```

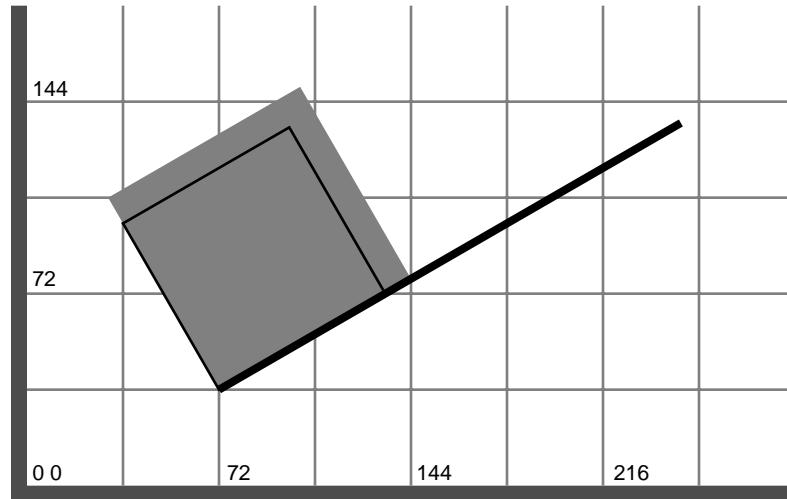
is not the same as

```
30 rotate
```

Even though a 30° rotation has occurred, the square is no longer the same size. Note the difference in the width and height of the square in `concat_1.eps`. The width is now greater than the height. This is easily seen in the next example. The same square is double skewed for a 30° rotation and a normal 30° rotation.



10-8



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat_2.eps
%%BoundingBox:30 36 252 150

72 36 translate
/square {0 0 moveto 0 72 lineto 72 72 lineto 72 0 lineto
         closepath} def
gsave
  .5 setgray
  [1 .5774 -.5774 1 0 0] concat
  square fill
grestore

gsave
  30 rotate
  square stroke
  3 setlinewidth
  0 0 moveto 200 0 lineto stroke
grestore

```

The proper method of rotation with the `concat` operator can be accomplished with this syntax:

```
[cosθ sinθ -sinθ cosθ 0 0] concat
```

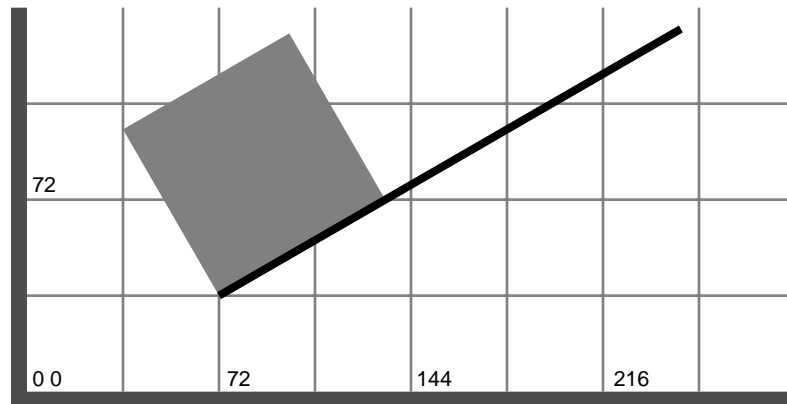
which is in this case

```
[cos30° sin30° -sin30° cos30° 0 0] concat
```

The values used for this operation can be found in appendix B. A rotation example follows:



10-9



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat&rotate.eps
%%BoundingBox:34 36 252 140

72 36 translate

gsave      % 30 degrees
    .5 setgray
    [.866 .5 -.5 .866 0 0] concat
    0 0 moveto 0 72 lineto 72 72 lineto 72 0 lineto
    closepath fill
grestore

gsave
    3 setlinewidth
    30 rotate
    0 0 moveto 200 0 lineto stroke
grestore

```

A combined scale and rotation can be performed with the `concat` operator in one matrix array. In the next example, the box is drawn as 1 unit by 1 unit. Using `concat`, we can scale it to 72 by 72 and rotate it 30°.

The syntax for this matrix is

```
[s*cosθ s*sinθ s*-sinθ s*cosθ 0 0] concat
```

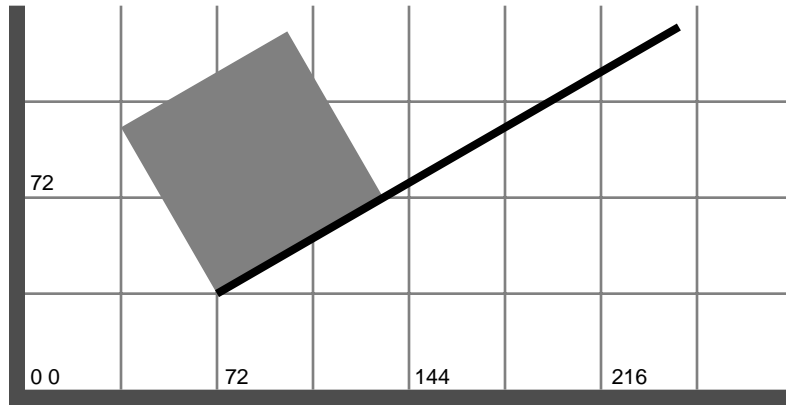
where s is the scale factor. Calculated, it will be:

```
[62.352 36 -36 62.352 0 0] concat
```

An example follows:



10-10



```

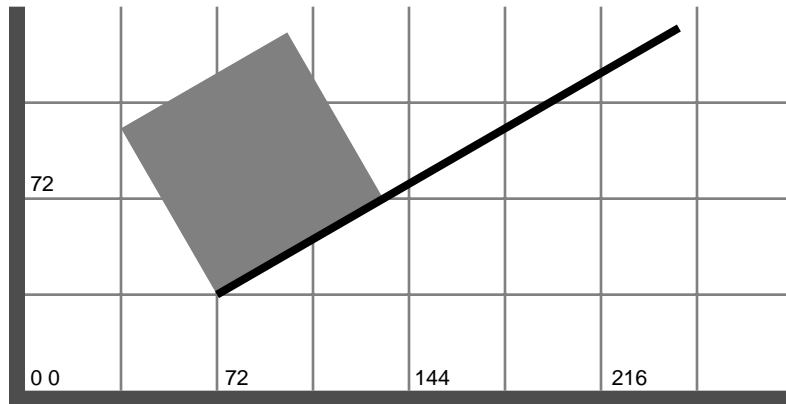
%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat_r&s.eps
%%BoundingBox:34 36 252 140

72 36 translate

gsave
    .5 setgray
    [62.352 36 -36 62.352 0 0] concat
    0 0 moveto 0 1 lineto 1 1 lineto 1 0 lineto
    closepath fill
grestore

gsave
    3 setlinewidth
    30 rotate
    0 0 moveto 200 0 lineto stroke
grestore
    
```

Finally, the fifth and sixth numbers of the matrix array perform a translation. In the next example, the `72 36 translate` is used only by the rotated line. The square is positioned by the `concat` operator.





10-11

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat_all.eps
%%BoundingBox:34 36 252 140

gsave
  .5 setgray
  [62.352 36 -36 62.352 72 36] concat
  0 0 moveto 0 1 lineto 1 1 lineto 1 0 lineto
  closepath fill
grestore

gsave
  72 36 translate
  3 setlinewidth
  30 rotate
  0 0 moveto 200 0 lineto stroke
grestore
```




advanced type

The previous chapter on type only covered the basics of working with type. There are a number of powerful PostScript operators for modifying fonts and kerning fonts. Kerning is performing custom character spacing for better legibility, form, or special effect. For example, the lower case *o* is spaced differently in the words *Do* and *To*. Careful attention to kerning would tuck the *o* under the *T* for proper letterspacing.

11.1 *modifying existing fonts*

As discussed in chapter 7, the outlines of fonts are stored as 1 point outlines. To be more precise, they are stored within a matrix that scales it to a 1 point square. This matrix is independent of the CTM, but acts in much the same way. In chapter 10, we manipulated the CTM with the `concat` operator. The font matrix is manipulated with the `makefont` operator. The next two PostScript fragments are equivalent:

```
/Helvetica-Bold findfont 100 scalefont setfont
```

```
/Helvetica-Bold findfont [100 0 0 100 0 0] makefont setfont
```

The `makefont` operator expects a matrix array that acts in the same way as the matrix array used with `concat`. The first number of the array represents the x scale, the fourth number represents the y scale. The other four remaining numbers also act the same as explained in section 10.4.



learn

11-1

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:makefont_1.eps
%%BoundingBox:0 0 340 75
```

```
/Helvetica-Bold findfont
[150 0 0 100 0 0] makefont setfont
```

```

0 0 moveto (T) show

/Helvetica-Bold findfont
[125 0 0 100 0 0] makefont setfont
100 0 moveto (T) show

/Helvetica-Bold findfont 100 scalefont setfont
188 0 moveto (T) show

/Helvetica-Bold findfont
[75 0 0 100 0 0] makefont setfont
255 0 moveto (T) show

/Helvetica-Bold findfont
[50 0 0 100 0 0] makefont setfont
308 0 moveto (T) show

```

All the *T*s above are Helvetica-Bold 100 point, at least in height. The first is scaled 150%, the second 125%, the third is normal, the fourth is condensed 75%, and the fifth is condensed 50%. The Helvetica-Narrow font on the LaserWriter Plus is condensed in this fashion (see section 15.3). It is not a separate drawing of Helvetica like Helvetica Condensed, but a mathematically condensed version of the font.

In the next example, the third number of the matrix array is given a value and the first and fourth are set to 100. The third number of the matrix skews the *y* axis left and right. The number unfortunately does not represent the angle that the matrix is skewed (see 10.4). It is the product of this equation:

$$y * \tan\theta$$

See the preceding chapter for detailed information on matrix manipulation.



11-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:makefont_2.eps
%%BoundingBox:0 0 360 75

/Helvetica-Bold findfont      % -45
[100 0 -100 100 0 0] makefont setfont
75 0 moveto (T) show

/Helvetica-Bold findfont      % -20
[100 0 -36.4 100 0 0] makefont setfont
105 0 moveto (T) show

/Helvetica-Bold findfont 100 scalefont setfont

```

```

150 0 moveto (T) show           % normal
/Helvetica-Bold findfont       % 20
[100 0 36.4 100 0 0] makefont setfont
195 0 moveto (T) show

/Helvetica-Bold findfont       % 45
[100 0 100 100 0 0] makefont setfont
225 0 moveto (T) show

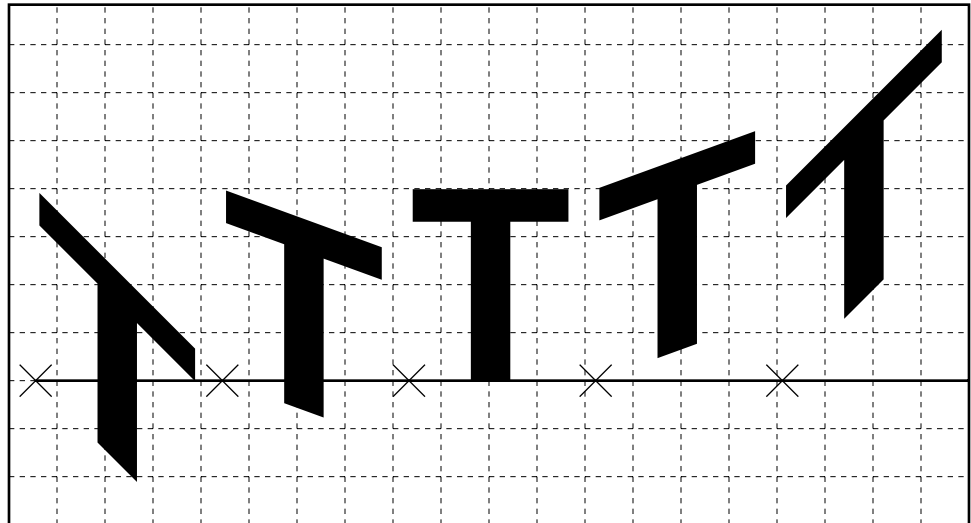
```

All the *T*s above are again Helvetica-Bold 100 point in size. The first *T* is obliqued -45° , the second is -20° , the third is normal, the fourth is at 20° , and the fifth is at 45° .

In the next example, the letterforms get stretched more dramatically when the x axis is skewed. Again, the type is the equivalent of 100 point. Now the second digit of the matrix array is changed. The number is arrived at in the same way as before, except the angle movement will be up and down.

This example is written differently to help keep track of where the current point is located for each *T*. An X is drawn at each *T*'s origin for a point of reference. Even though the origins of the *T*s make a horizontal row, the baselines of the *T*s in effect rotate because of the skewing of the x axis.

All the *T*s below are again Helvetica-Bold 100 point. The first *T* is obliqued -45° , the second is -20° , the third is normal, the fourth is at 20° , and the fifth is at 45° .



learn

11-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:makefont_3.eps
%%BoundingBox:0 0 360 195

/x    {-6 -6 moveto 6 6 lineto stroke
      -6 6 moveto 6 -6 lineto stroke} def

.5 setgray           % background
0 0 moveto 360 0 lineto 360 195 lineto 0 195 lineto
closepath fill

```

```

10 50 translate

1 setgray                                % baseline
0 0 moveto 370 0 rlineto stroke

0 setgray .5 setlinewidth

/Helvetica-Bold findfont                % -45
[100 -100 0 100 0 0] makefont setfont
x 0 0 moveto (T) show

70 0 translate
/Helvetica-Bold findfont                % -20
[100 -36.4 0 100 0 0] makefont setfont
x 0 0 moveto (T) show

70 0 translate                          % normal
/Helvetica-Bold findfont 100 scalefont setfont
x 0 0 moveto (T) show

70 0 translate
/Helvetica-Bold findfont                % 20
[100 36.4 0 100 0 0] makefont setfont
x 0 0 moveto (T) show

70 0 translate
/Helvetica-Bold findfont                % 45
[100 100 0 100 0 0] makefont setfont
x 0 0 moveto (T) show

```

11.2 *letterspacing with ashow*

The four PostScript kerning operators are **ashow**, **widthshow**, **awidthshow**, and **kshow**. Each provides different opportunities. In chapter 18 on PostScript Level 2, there are four additional type operators.

ashow provides a means to add or subtract overall character spacing both vertically and horizontally.

The syntax for **ashow** is:

```
x y string ashow
```

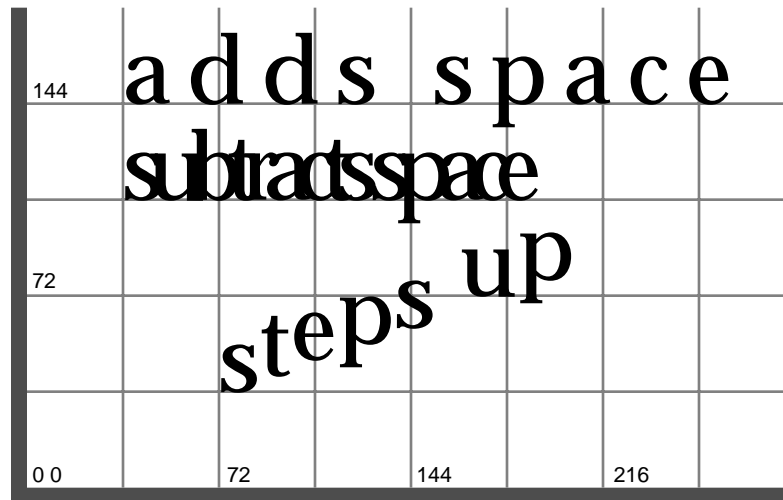
Where:

x adds or subtracts space horizontally in the string.

y adds or subtracts space vertically in the string.

string is the characters or words affected.

In the following example, a string is tightened, stretched, and shifted vertically. The vertical adjustment is typically used by non-roman fonts.



11-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:ashow_1.eps
%%BoundingBox:36 36 264 170

/Palatino-Roman findfont 36 scalefont setfont

36 144 moveto
6 0 (adds space) ashow

36 108 moveto
-6 0 (subtracts space) ashow

72 36 moveto
0 6 (steps up) ashow

```

11.3 *kerning with widthshow*

`widthshow` kerns a single character within a string. Its syntax is:

```
x y character string widthshow
```

Where:

x adds or subtracts horizontal space after the character.

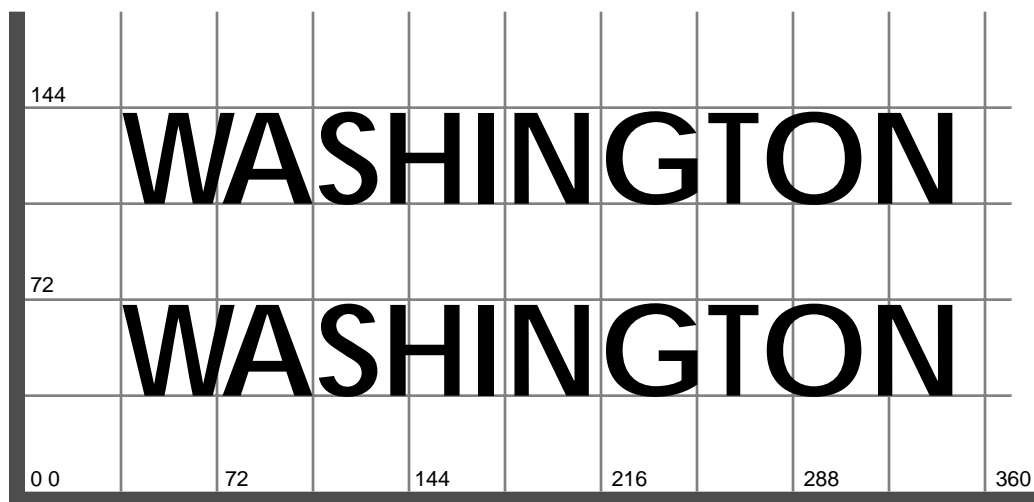
y adds or subtracts vertical space after the character.

character is either identified by its *decimal* or *octal character code* from the ASCII chart. See appendix A.

In the next example, the space is reduced after the *W*, which therefore brings the *A* closer. The only difference between the two *WASHINGTONs* is that the top *W* is identified by the decimal 87 and the bottom *W* by the octal 127. The default identification is decimal. If octal code is used, it needs the **8#** preface as a label.



11-5



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:widthshow_1.eps
%%BoundingBox:36 36 360 144

/AvantGarde-Demi findfont 48 scalefont setfont

36 108 moveto          % Decimal table
-6 0 087 (WASHINGTON) widthshow

36 36 moveto           % Octal table
-6 0 8#127 (WASHINGTON) widthshow

```

11.4 *kerning with awidthshow*

awidthshow is the combination of **ashow** and **widthshow**. It both kerns a single character within a string and adds or subtracts overall character spacing both vertically and horizontally. Its syntax is:

```
xc yc character xs ys string widthshow
```

Where:

x^c adds or subtracts horizontal space after the character.

y^c adds or subtracts vertical space after the character.

character is either the decimal or octal character code for the character.

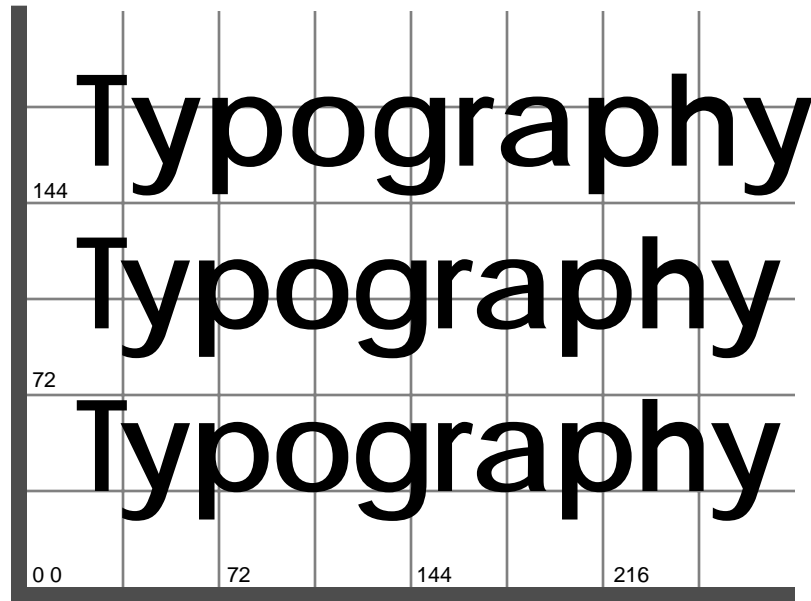
x^s adds or subtracts horizontal space in the string.

y^s adds or subtracts vertical space in the string.

In the next example, the word *Typography* on top is set normally with **show**. In the middle and bottom, the space after the *T* is reduced to kern the *y* underneath and overall character spacing is tightened by 1 point. Again, the character kerned is identified by its decimal or octal character code.



11-6



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:awidthshow 1.eps
%%BoundingBox:18 30 288 200

/AvantGarde-Demi findfont 48 scalefont setfont

18 158 moveto
(Typography) show

% using Octal table
18 97 moveto
-3 0 8#124 -1 0 (Typography) awidthshow

% using Decimal table
18 36 moveto
-3 0 084 -1 0 (Typography) awidthshow

```

11.5

kshow

The `kshow` operator has a much different strategy for kerning than the previous three PostScript operators. Its syntax is:

```
procedure string kshow
```

The `procedure` is executed after the positioning of each character. In the next example, two different procedures are performed on the same word. In the top word, the character spacing of *RAINBOW* is accomplished by using the procedure `-3 0 rmoveto`. After *R* is positioned, the new current point is moved to the right of its character width. The procedure `-3 0 rmoveto` shifts the current point left 3 points.

Other manipulations besides the adjustment of the current point can occur within

the procedure. With the *RAINBOW* on the bottom, the procedure also changes the current gray between each character setting. Any number of actions can be performed after each character.

The intention of this operator, however, is to give the opportunity to individually kern the characters of a string. Thus its name, *kernshow*. To accomplish this individual kerning after each character of the string (except the last), two numbers are pushed onto the stack. In the case of *RAINBOW*, after the *R* is set, the decimal codes for *R* and *A* are pushed onto the stack. They are 82 and 65. After the *A* is set, the decimal codes for *A* and *I* are pushed onto the stack. This occurs after each character of the string except for the last. These numbers can be used by the procedure to make a custom spacing decision based on which two characters are beside each other. This comparison is not needed after the last character. If unused, these numbers are left on the stack:

```
82 65 65 73 73 78 78 66 66 79 79 87
```

Since they are not used by the procedure in this example, we need to discard them. This is done with the `12 {pop} repeat`. The numbers can also be discarded within the *kshow* procedure.



learn

11-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:kshow_1.eps
%%BoundingBox:20 36 360 162

/AvantGarde-Demi findfont 72 scalefont setfont

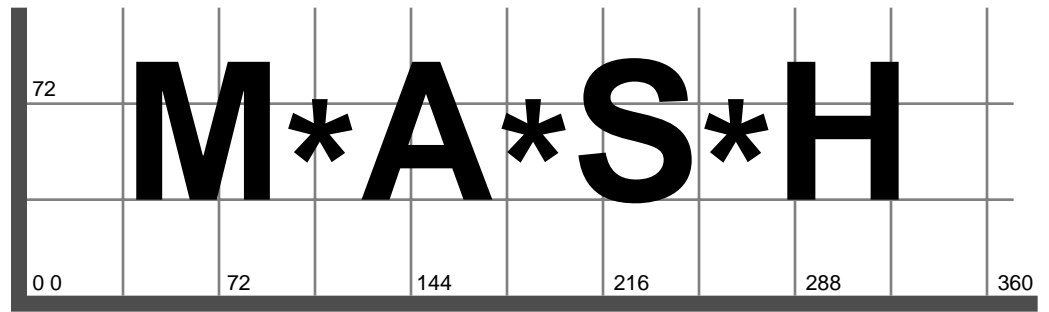
18 108 moveto
{-3 0 rmoveto} (RAINBOW) kshow
12 {pop} repeat

.9 setgray
18 36 moveto

{currentgray .12 sub setgray pop pop} (RAINBOW) kshow

```

Another example follows:



11-8

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:M*A*S*H.eps
%%BoundingBox:36 36 360 90

/Helvetica-Bold findfont 72 scalefont setfont

% procedure lowers the asterisk & then returns to baseline
/putAsterisk      {0 -14 rmoveto (*) show 0 14 rmoveto
                  pop pop} def

36 36 moveto
{putAsterisk} (MASH) kshow

```




the image operator / scan

The **image** operator is used to control painting scanned pictures. It can also be used to create a variety of patterns and blends. Discussion of the **image** operator will be divided over two chapters. This chapter concentrates on scanned pictures and chapter 13 will focus on creating patterns and blends. In chapter 14, the **imagemask** operator, a variation of **image**, will be explained.

12.1 *introduction*

The **image** operator can receive the data for the picture it creates from a number of sources. The most obvious source would be the data from a scanner of some kind. In sections 12.3 and 12.4, the traditional use of the **image** operator is explained with first a very simple picture of a shamrock and then a scanned picture.

The other source of data can be provided by the programmer. The provided data is usually some arbitrary characters that can create a pattern or the means to create a smooth blend of color. See chapter 13.

In all cases, either scanned data for a picture or arbitrary characters for a pattern, a data acquisition procedure of some kind is needed. A data acquisition procedure controls how the **image** operator receives or creates the information needed to paint a picture.

The syntax of **image** is:

```
width height bits matrix proc image
```

Where:

- width** is the width of the picture in cells or pixels.
- height** is the height of the picture in cells or pixels.
- bits** is the number of bits of information per cell or pixel. It can be 1, 2, 4, or 8.
- matrix** is the picture's coordinate system.
- proc** is the data acquisition procedure that obtains or creates the picture data.
- image** is the PostScript operator.

12.2 *simple digitized pictures*

Scanned pictures can be thought of as a complex paint by number coloring project dividing a picture into little squares. Each square, or pixel, would have a code assigned to it based on the value or the color it represents. The pixels would be arranged into rows and columns that would be used to describe the picture's width and height. When the picture is painted by the `image` operator, `image` uses one of four coding schemes to paint the pixel a particular value. Depending how the picture was originally scanned, the picture can be painted as a 1-bit, 2-bit, 4-bit, or 8-bit picture. This is the coding scheme for a black and white picture. Color pictures would be multiples of this. The difference between each of these coding schemes is the number of different grays they can produce. The size of the file increases as the number of bits increase.

These are the possible bits and what they can represent:

- 1-bit 2 grays. Actually either black or white. Value is either 0 or 1.
- 2-bit 4 grays. Black, white, and 2 intermediate grays.
Value is either 0, 1, 10, or 11 (counting from decimal 0 to 3 in binary).
- 4-bit 64 grays. Black, white, and 62 intermediate grays.
- 8-bit 256 grays. Black, white, and 254 intermediate grays.

The bits that we are using to describe value are binary bits. Binary is a numbering system that only uses 0 and 1. In 1-bit coding, the value of the pixels will be either a 0 or a 1. 0 is the code for black, 1 the code for white. We'll come back to the other bits later. Looking further at a 1-bit picture, figure 12-1 represents a picture divided into 256 pixels. There are 16 rows and 16 columns.

1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	1
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0
0	0	0	0	1	1	1	0	0	1	1	1	0	0	0	0
1	0	0	1	1	1	1	0	0	0	1	1	1	0	0	1
1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1

figure 12-1

If you follow the coloring assignments and paint the 0s black and the 1s white, a picture will emerge. See figure 12-2.

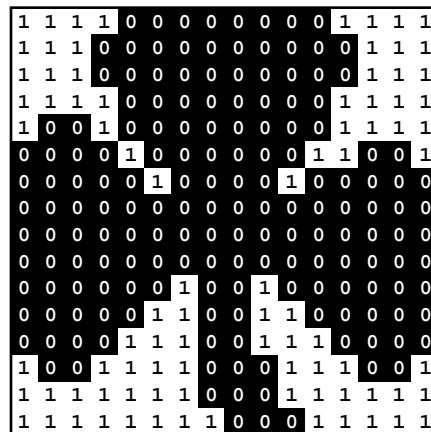


figure 12-2

It's the job of the scanner to get all this for you. Otherwise, it would be a major job to construct a picture. The scanner uses the intensity of light reflected back from a picture to assign a value to each picture. The number of bits the scanner's software is set for will determine how the light is divided.

Appendix A is an ASCII table of five different coding schemes, each a different way of counting from 0 to 255. In particular, we are interested in ASCII, hexadecimal, and binary code assignments. The ASCII table contains our alphabet within the 256 entries. Note that there isn't a visible character for all 256. *Hexadecimal* counts from 0 to 256 using only the numbers 0-9 and the characters A-F. The binary counts from 0 to 255 in binary (using 0 and 1).

The `image` operator uses the binary table to give pixels a value. It stores the pixel value as either ASCII or hexadecimal data.

As mentioned before, 0 and 1 are the two possible values for a bit. In the 1-bit coding scheme, 0 and 1 represent black and white. Each entry of the binary table of appendix A contains 8 bits. Therefore, each entry of the table can represent 8 pixels of a picture using the 1-bit coding scheme. The first 8 pixels starting at the lower left of figure 12-2 have the values 1 1 1 1 1 1 1 1. Consulting appendix A, 11111111 will be found at decimal 255. There isn't a visible character in the ASCII table and it's FF in hexadecimal.

Therefore, the first 8 pixels of figure 12-2 can be saved as either ASCII 255 or hexadecimal FF. See figure 12-3 below. This coding would continue for every row of the picture.

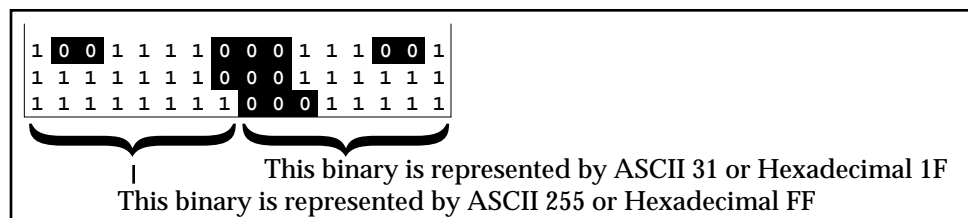
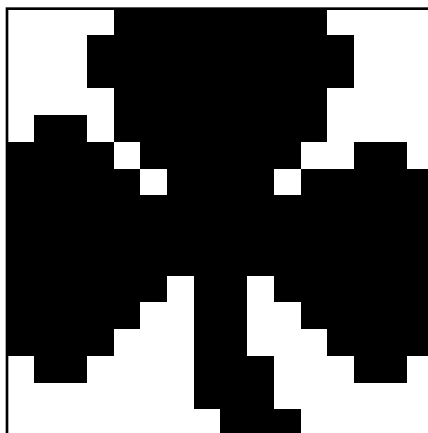


figure 12-3

The ASCII table contains the characters that may be typed from a standard computer keyboard plus many other special characters whose location may vary from one system to another such as \tilde{E} or β . There are a total of 256 possible characters. The difficulty in using the ASCII table in designing the `shamrock_1.eps` example was having a visible ASCII character for many of the particular binary patterns needed. Therefore, in the next example hexadecimal was

used. However, many scanners save their files using the ASCII table because the file will be half the size. At decimal 97, there is the ASCII character *a* (1 byte of storage), and the hexadecimal 61 (2 bytes of storage).

The following is an example of how figure 12-2 would be written as a PostScript program. It is not a scanned picture, but it is written in the same way a scanned picture would be written, only on a smaller scale. Section 12.3 will demonstrate how the same technique is applied to a scanned photograph.



12-1
image demo

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrock_1.eps
%%BoundingBox:0 0 160 160

/picStr 2 string def

/shamrock {
    16 16 1 [.1 0 0 .1 0 0]
    {currentfile picStr readhexstring pop} image
} def

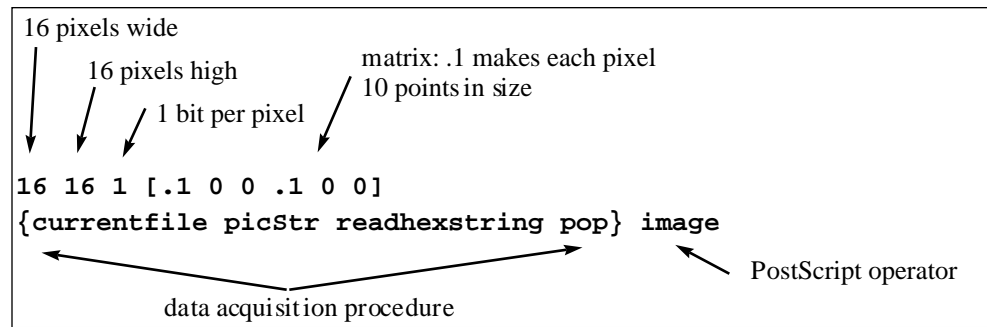
shamrock
FF 1F FE 3F 9E 39 0E 70 06 60 02 40 00 00 00 00
00 00 04 20 08 19 90 0F F0 0F E0 07 E0 07 F0 0F

% frame
0 0 moveto 160 0 lineto 160 160 lineto 0 160 lineto closepath
stroke

```

In `shamrock_1.eps` above, the procedure `/shamrock` is defined by the `image` operator and its operands. Figure 12-4 labels the arguments (required information for an operator) for the `image` operator.

figure 12-4



Let's look closer at each argument for `image`:

The `width` and `height` are the number of cells wide and high for the picture. In the example above, the picture's width and height is 16 by 16 pixels.

The `bits` value is 1 since the picture is a 1-bit picture.

Next is the matrix array which determines the size of the picture. This matrix array operates differently with `image` than for `concat` (section 10.4) and `makefont` (section 11.1). With `concat` and `makefont`, if we wanted an object larger, we would have used a larger number in the first and fourth position of the array. Here it is the opposite; a smaller number is used. The calculation is different with `image` because of a difference in how it handles pictures. It can be explained in this way. Had the matrix array been `[1 0 0 1 0 0]`, the shamrock would be as shown in figure 12-5.

figure 12-5



Figure 12-5 is this size because the 1 in the `[1 0 0 1 0 0]` matrix array means each pixel is actual size. The 16x16 pixel width and height will measure 16x16 points. A 16x16 pixel width and height using a `[.1 0 0 .1 0 0]` matrix is 160x160 points. This is arrived at by dividing 16 by .1 equaling 160, and 160 divided by 16 equals 10. With a dimension of 160x160, and the cells width and height being 16x16, we then know the individual pixels will be 10 points square. If the matrix array had been `[.2 0 0 .2 0 0]` the pixels would be 5 points square.

`{currentfile picStr readhexstring pop}` is the data acquisition procedure for the `image` operator. `currentfile` identifies that the data for the picture will follow after `shamrock` is used in the program. `/picStr` defines a space large enough for a string two characters long. `picStr` is a temporary holding place for the data for each row of the picture. The first two characters that make up the first row of the shamrock are the two hexadecimal characters `FF` and `1F`. They represent the two sets of binary characters `11111111` and `00011111`. `readhexstring` identifies that the data is in hexadecimal. `pop` clears the `picStr` so the procedure can be repeated. The procedure is repeated until the 16 rows of data in increments of two are filled.

Writing the program in this way permits all the data for the picture to follow the PostScript commands that control its painting. In most cases, scanned pictures are very large files. Having the data follow a PostScript header permits easier editing of the program.

12.3

scanned pictures

This picture was obtained by using a scanner and saving the file in a PostScript format by its software. I have edited the PostScript from its original form supplied by the scanning software. The original file had definitions to accommodate a number of different scanning situations and I wish to focus on the basic similarity with the previous shamrock example.

The program that produced the picture follows, minus the picture data to save space. It would have taken up about eight pages.



12-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:relief.eps
%%BoundingBox:0 0 224 225

/height 225 def
/width 224 def
/nheight height neg def
/picstr 112 string def

/makePicture
{ width height 4 [width 0 0 nheight 0 height]
{ currentfile picstr readstring pop } image } def

gsave
{1 exch sub} settransfer
width height scale

makePicture
~>fl>> a a > a ee]a ~]a TM TM πógfUTVUEDDUgfyzo'>fl

- 25.6K of picture data, etc -

î>î]a]ö™óyx°ÃÃfi~>~> , <Ã öE~><°ÃÖÃ™°>ô~
grestore

```

This program is basically the same as `shamrock_1.eps` but with a few differences. First, many of the values have been given names such as `height` and `width`. This section of the program is written by the scanning software so the same file beginning can be used for every file it creates.

`height` defines the height of the picture to be 225.

`width` defines the width of the picture to be 224.

`nheight` defines `height` to have a negative value. The reason for this is many scanners digitize from the top down as opposed to from the bottom up. Having the transformation matrix be `[width 0 0 nheight 0 height]` adjusts for this by flopping the picture.

`picStr`, as discussed at the end of section 12.2, is a holding space for a string. In this definition, the holding place is `112`. This would hold enough data for one row of the picture.

`makePicture` defines the `image` operation as `shamrock` did for the `shamrock_1.eps` example earlier.

```
width height 4 [width 0 0 nheight 0 height]
```

is the same as

```
224 225 4 [224 0 0 -255 0 255].
```

The picture is a 4-bit picture (64 grays) and the matrix array maps it, so it is very small. To get it to the right size,

```
width height scale
```

scales the picture up to `224 225`.

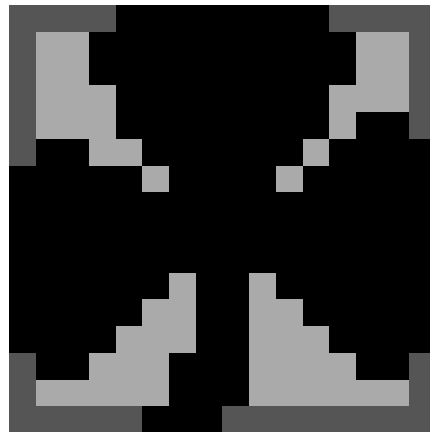
`{1 exch sub} settransfer` inverts the picture. Without this line, the picture would appear as a negative. This compensates for the way the scanner scans the picture.

12.4 *a second version of shamrock*

The previous example, `relief.eps`, is a 4-bit picture. To demonstrate how 4-bit and 8-bit pictures work with their data, the next example is a version of `shamrock` written as a 2-bit picture. A 4-bit and 8-bit version would be similar.



12-3



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrock_2.eps
%%BoundingBox:0 0 160 160

/picStr 4 string def

/shamrock {16 16 2 [.1 0 0 .1 0 0]
           {currentfile picStr readhexstring pop} image} def

shamrock
55 40 55 55
6A A0 2A A9
42 A0 2A 81
00 A8 2A 00
00 28 28 00
00 08 20 00
00 00 00 00
00 00 00 00
00 00 00 00
00 20 08 00
42 80 02 00
6A 00 00 81
6A 00 00 A9
68 00 00 29
68 00 00 29
55 00 00 55
    
```

Figure 12-6 enlarges the first two rows of `shamrock_2.eps` for a better look at the bit values. 10 is a light gray, 01 is a dark gray, 00 is black, and if there had been a 11 it would be white.

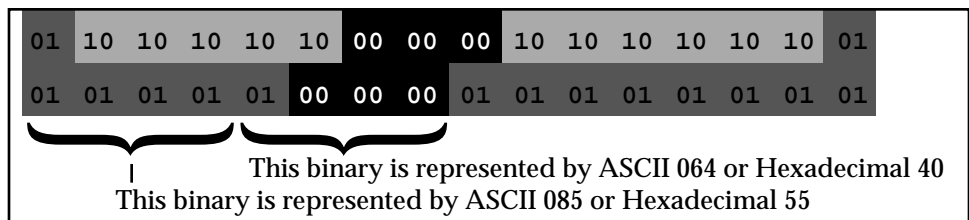


figure 12-6

In a 1-bit picture, 8 bits is the data for 8 pixels. In a 2-bit picture, 8 bits is the data for 4 pixels. With 4-bit, it's 2 pixels and with 8-bit, it's one pixel. This is covered more in the next chapter.

When a picture is scanned, the trade-off will be the size of the file versus the quality of the picture. Figure 12-7 is a photograph divided into four vertical segments, 1-bit, 2-bit, 4-bit, and 8-bit in order. The 1-bit section is about 4.5 K and the 8-bit is just over 26K.



figure 12-7

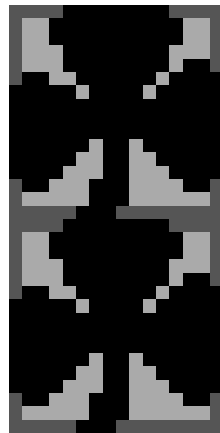


the image operator / patterns

In this second chapter on the `image` operator, it will be shown how `image` can be creatively used for making different kinds of patterns and for making fountains, also known as blends or ramps. See also the next chapter on `imagemask` which can also be used for making patterns.

13.1 *creating patterns with the image operator*

`shamrock_2.eps` could be written in such a way as to produce various patterns. In `shamrock_2.eps` the data for `shamrock` was only used once. However, if the `height` (or number of rows) were to be doubled and a name given to the data for `shamrock`, the picture could be used twice to fill the space.



learn

13-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrockTwice_1.eps
%%BoundingBox:0 0 80 160

```

```

/shamrock <55 40 55 55 6A A0 2A A9 42 A0 2A 81 00 A8 2A 00
00 28 28 00 00 08 20 00 00 00 00 00 00 00 00 00
00 00 00 00 00 20 08 00 42 80 02 00 6A 00 00 81
6A 00 00 A9 68 00 00 29 68 00 00 29 55 00 00 55> def

```

```

16 32 2 [.2 0 0 .2 0 0] { shamrock } image

```

Had the same thing been done with `shamrock_1.eps`, disaster would strike.



learn

13-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrockTwice_2.eps
%%BoundingBox:0 0 80 160

/picStr 2 string def

/shamrock { 16 32 1 [.2 0 0 .2 0 0]
            {currentfile picStr readhexstring pop} image} def

shamrock
FF 1F FE 3F 9E 39 0E 70 06 60 02 40 00 00 00 00
00 00 04 20 08 19 90 0F F0 0F E0 07 E0 07 F0 0F

% frame
0 0 moveto 160 0 lineto 160 160 lineto 0 160 lineto closepath
stroke

```

It looks like this because `readhexstring` will gobble up everything after `shamrock` is used, including the `0 0 moveto 160 0 lineto 160 160 lineto 0 160 lineto closepath stroke` looking for picture data. An error will occur because the data available and the defined picture size do not match.

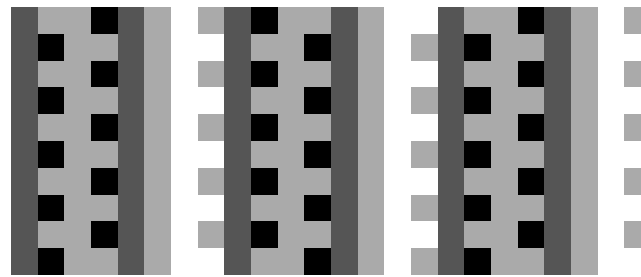
13.2

imageWord.eps

Similar to `shamrockTwice_1.eps`, the following is a PostScript program using the `image` operator to make a pattern:



13-3



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:imageWord.eps
%%BoundingBox:0 0 240 100
    
```

```

/word (John) def
24 10 2 [.1 0 0 .1 0 0] { word } image
    
```

In the picture above, the binary for the word *John* is repeatedly used until a rectangle 24 by 10 is formed as shown in figure 13-1.

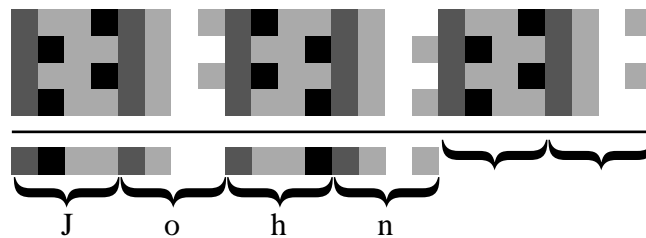


figure 13-1

The binary for the word *John* is:

- 01001010 or J
- 01101111 or o (two gray then two white)
- 01101000 or h
- 01101110 or n

The word just once would look like this:



Looking at the program again, the strip above is used over and over again until the picture rectangle is completed.

If `/word` above were to be redefined to `(John s)`, it would look like figure 13-2.



figure 13-2

It appears striped because the 6 characters of (John S) divide evenly into the 24 pixel width of the picture.

The previous examples of `imageWord.eps` use the picture data to make a 2-bit image, meaning every pixel of the picture has 2 bits of information, or four possible values. A pixel is an individual square of value. For example, the binary for *J* is 01001010 and it will be divided into four parts containing 2 bits each as 01 00 10 10. The number 01 represents a dark gray, and 10 a light gray, 00 represents a black, and 11 is white. A 2-bit picture therefore can only have 4 levels of gray: black, white, and two grays.

Should *J* be used in a 1-bit picture, *J* would look like this:



01001010 as a 1-bit picture is black, white, black, black, white, black, white, black. A 1-bit picture has 2 possible levels — black and white. The monitor of the Macintosh Plus and SE computers is a 1-bit screen. The pixels on the screen are either black or white. MacPaint creates 1-bit pictures.

A 4-bit picture would have 64 gray levels. The binary for *J* in this case would be the information for two pixels of a picture. One would be 0100 and the other would be 1010, representing 2 grays out of a possible 64. It would look like this:



An 8-bit picture would have 256 gray levels. The entire 01001010 would be used to represent a single gray level. Black is 00000000 and white is 11111111. In between, there would be 254 different combinations of 0s and 1s for a total of 256 levels of gray.

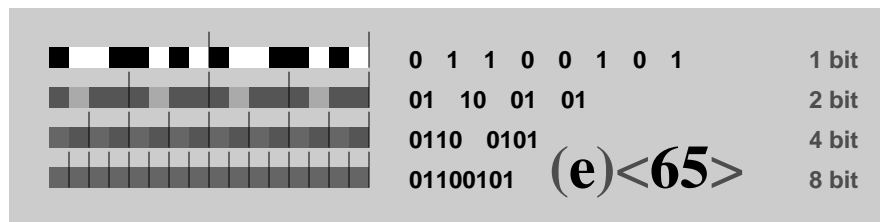
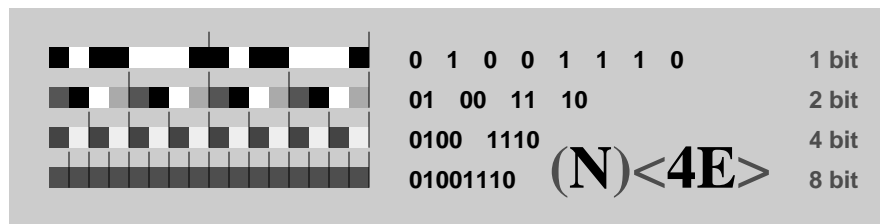
13.3

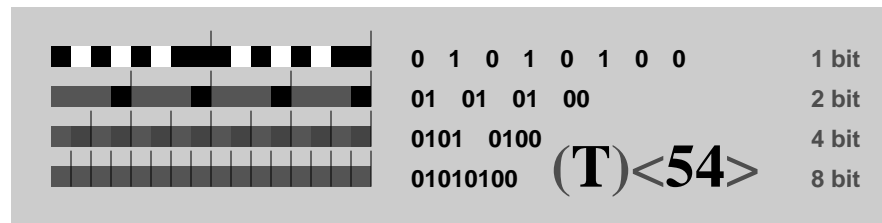
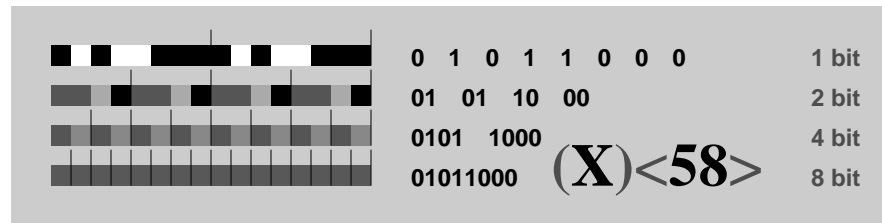
1248.eps

The following are charts for *N*, *e*, *X*, and *T* showing how the binary for each character would be painted as a 1, 2, 4, and 8-bit picture. The vertical line marks off the number of times the letter is used. The PostScript program listing for `1248-132.eps` can be found at the end of appendix A.



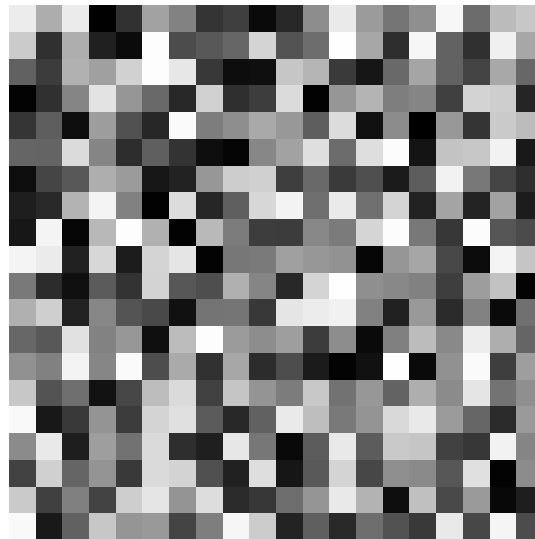
13-4





13.4 *random patterns*

Another example of a data acquisition procedure is the program that creates a random pattern within a certain range of grays.



learn

13-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:pattern_1.eps
%%Creator:John F Sherman
%%CreationDate:July 1989
%%BoundingBox:0 0 200 200

/str 512 string def
/pattern
  {/light exch def /dark exch def
  /diff light dark sub def

```

```

20 20 8 [.1 0 0 .1 0 0]
{0 1 511 {str exch rand diff mod dark add put}
for str} bind image } def

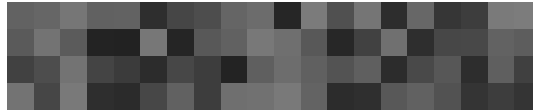
```

```

222 srand
0 255 pattern

```

Next is a version of the `pattern` procedure that limits the values to between 33 and 125. We'll next explain this example line by line. A review of the discussion of the `for` and `put` operators in section 8.3 might be helpful.



13-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:expattern1.eps
%%BoundingBox:0 0 200 40

```

```

/str 20 string def

```

```

/pattern
  {/light exch def /dark exch def
  /diff light dark sub def
  20 4 8 [.1 0 0 .1 0 0]
  {0 1 19 {str exch rand diff mod dark add put} for
  str }image} def

```

```

178 srand
33 125 pattern

```

This pattern is an 8-bit, 20x4 picture. Since it is an 8-bit picture, each row is created by 20 characters of data. A total of 80 characters will be needed for the entire picture. As explained in section 12.2, each character of the ASCII table represents a gray value somewhere between 0 and 255. 0 is black and 255 is white. That means character 33 would be the equivalent of a `.128 setgray` or 87.2% black and character 125 would be 51.2% black. The job of the `pattern` data acquisition procedure is to randomly create a string of 20 characters that are always between 33 and 125. I use this range because it is a portion of the ASCII table with visible characters.

The first line

```

/str 20 string def

```

defines a string 20 characters in length, initially with the value of (00000000000000000000)

Next begins the definition for the `pattern` procedure.

The first line of the procedure definition is

```

/light exch def /dark exch def

```

which defines the top two values on the operand stack to be `light` and `dark`. In this example, looking ahead to the last line of the program, `light` will be defined as 125 and `dark` will be 33. 125 will be on top of the stack with 33 below it.

The second line of the procedure defines `diff` as the subtraction of `dark` from `light`, which will in this case be 92.

Next, `20 4 8 [.1 0 0 .1 0 0]` are the first arguments for the `image` operator, setting up the picture to be 20 by 4 pixels, with each pixel to have an 8-bit value and the matrix array determining the size of the picture.

Next is the data acquisition procedure:

```
0 1 19 {str exch rand diff mod dark add put} for
```

This line creates a string 20 characters in length, each of which is a character that falls between 33 and 125 on the ASCII chart. Breaking this line down, after the first count of the `for` loop, 0 then `str` (which is (00000000000000000000)) will be on top of the stack. `exch` will switch those positions so that 0 is on top. Zero will be used later as the index for the `put` operator. `rand` creates a random number, which will be 2991646 on the first go around. (Don't worry for now about how I know that. See chapter 16.) The random numbers produced here are a reproducible sequence of numbers based on the 178 `srand` a bit later in the program. That is, the random sequence will always be the same unless 178 is changed (it may be a different repeatable sequence on your machine). Next, `diff` was defined earlier to be 92. After that is the operator `mod`, which returns to the stack the remainder of a division. In this case, it is the division of 2991646 by 92, which has a remainder of 82 (rounded off). Then comes `dark`, defined earlier as 33 and `add` which adds the 82 and 33 giving you 115. Checking the ASCII table, 115 is the character `s`. The `str` looks like this after the first count of `for`:

```
(00000000000000000000) 0 115 put which becomes
(s00000000000000000000)
```

This process is repeated until `str` is filled with 20 characters.

Then `str` is used as data for the first row of the picture by the `image` operator.

Specifically, the first 20 characters in `str` would be:

```
sFy.+Ga=oqyc,/XaR4=4
```

This was arrived at by printing the string instead of using it as data for the image operator.



13-7

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:seepattern1.eps
%%BoundingBox:0 0 180 30

/Times-Bold findfont 16 scalefont setfont
/str 20 string def

/pattern
  {/light exch def /dark exch def
  /diff light dark sub def
  0 1 19 {str exch rand diff mod dark add put} for} def
178 srand

33 125 pattern % create 20 between 33 & 125

7 10 moveto str show
```

```
.25 setlinewidth %frame
0 0 moveto 180 0 lineto 180 30 lineto 0 30 lineto
closepath stroke
```

This program would look like figure 13-3.

figure 13-3



Had only the one line, `sFy.+Ga=oqyc,/XaR4=4`, been used for the picture, a repeating pattern would emerge (figure 13-4) because the same string would be used over and over until the width and height were satisfied.

figure 13-4



However, the `pattern` data acquisition procedure continually creates a new 20 character string. The procedure had first created:

```
sFy.+Ga=oqyc,/XaR4=4
```

which is used to paint 20 pixels in the first row. The procedure then creates:

```
AMsD;,G=#_m`S_,IW-_@
```

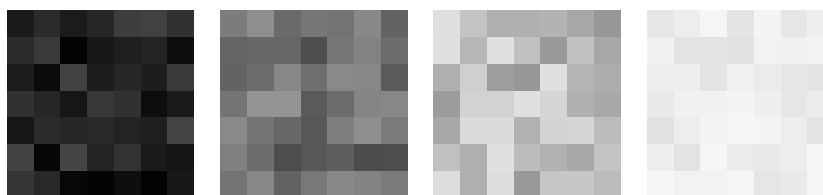
for the 20 pixels of the second row, looking like figure 13-5 at this point.

figure 13-5



And so on until the picture is completed.

The next example shows how `pattern` can look at different value ranges.



learn

13-8

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:pattern_by4.eps
%%Creator:John F Sherman c 1989
%%CreationDate:July 1989
%%BoundingBox:0 0 310 70
```

```
/str 512 string def
```

```
/pattern
```

```
{/light exch def /dark exch def
```

```
/diff light dark sub def
```

```
7 7 8 [.1 0 0 .1 0 0]
```



```

    {0 1 511 {str exch rand diff mod dark add put} for
    str} bind image } def
222 srand

0 75 pattern

80 0 translate
75 150 pattern

80 0 translate
150 225 pattern

80 0 translate
225 250 pattern

```

13.5 *variations on the patterns*

Variations of the `pattern` procedure can be achieved by changing the bit value for the image operator. The confinement of the picture to be within certain ranges of gray diminishes as the bits level gets smaller, but different textures are produced. Below, the first row of patterns are 1-bit, followed by 2, 4, and 8-bit.



learn

13-9

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:patternVar1.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 400 400

```

```

/str 512 string def
/pattern
  {/light exch def /dark exch def
  /diff light dark sub def
  7 7 bits [.1 0 0 .1 0 0]
  {0 1 511 {str exch rand diff mod dark add put} for
  str} bind image} def

17341734 srand
/bits 8 def

gsave      % bottom row
  0 75 pattern
  80 0 translate 75 150 pattern
  80 0 translate 150 225 pattern
  80 0 translate 225 250 pattern
grestore

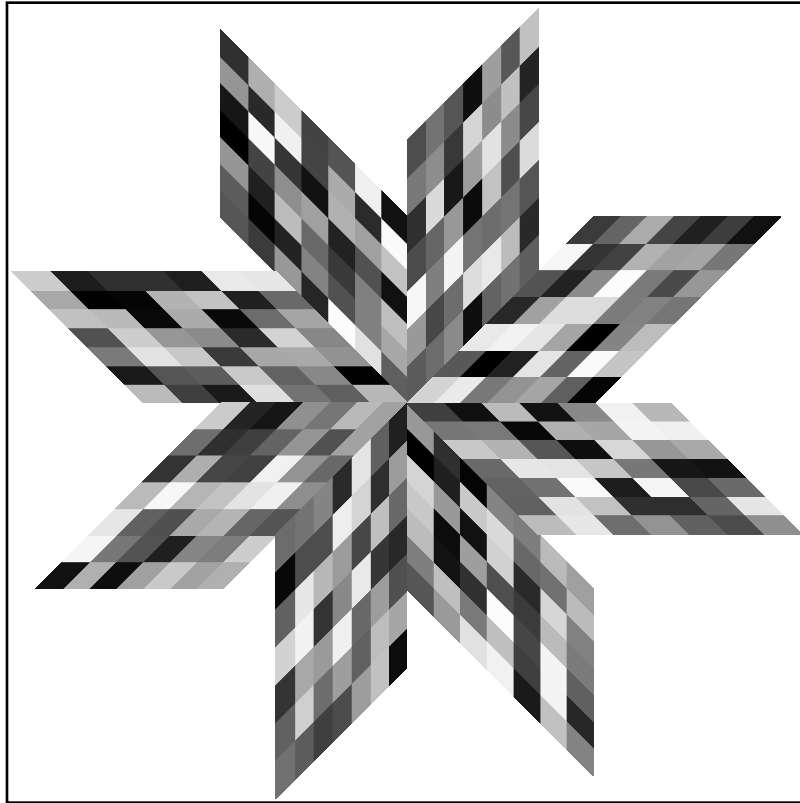
/bits 4 def
gsave      % second row
  0 80 translate 0 75 pattern
  80 0 translate 75 150 pattern
  80 0 translate 150 225 pattern
  80 0 translate 225 250 pattern
grestore

/bits 2 def
gsave      % third row
  0 160 translate 0 75 pattern
  80 0 translate 75 150 pattern
  80 0 translate 150 225 pattern
  80 0 translate 225 250 pattern
grestore

/bits 1 def
gsave      % top row
  0 240 translate 0 75 pattern
  80 0 translate 75 150 pattern
  80 0 translate 150 225 pattern
  80 0 translate 225 250 pattern
grestore

```

You can change the matrix as explained earlier in section 10.4. In the next example, having the matrix be $[\ .1\ 0\ -.1\ .1\ 0\ 0]$ skews the square pattern 45° to the right. Then the skewed square is used eight times as it is rotated in 45° increments.



learn

13-10

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:patternVar2.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 300 300

/str 512 string def
/pattern
  {/light exch def /dark exch def
  /diff light dark sub def
  7 7 8 [.1 0 -.1 .1 0 0]
  {0 1 511 {str exch rand diff mod dark add put} for
  str} bind image} def

17341734 srand
0 0 300 300 rectstroke
150 150 translate

gsave
  8 {45 rotate 0 255 pattern} repeat
grestore

```

13.6

fountains

Fountains, also known as ramps or blends, can be created with the `image` operator. The data acquisition procedure in this case creates a single string ordered from the

ASCII chart's beginning to end to create a transition of value. The first character of the chart is black, the last is white. The data acquisition procedure is a convenient means to order all 256 characters.

We could see that string by printing it instead of using it as the data for the `image` operator. This is basically the same program that was seen back in section 8.3 when the `for` and `put` operators were explained.



13-11

```
!PS-Adobe-2.0 EPSF-1.2
%%Title:seefountain_1.eps
%%BoundingBox:0 0 460 30

.25 setlinewidth
0 0 460 30 rectstroke
/Helvetica-Condensed findfont 4 scalefont setfont

/str 256 string def
0 1 255 { str exch dup put } for

5 10 moveto str show

str stringwidth pop 5 add
/strLength exch def

% mark beginning
1 setlinewidth
5 10 moveto 0 20 rlineto stroke

strLength 10 moveto 0 20 rlineto stroke
```

To show the complete string in this example, it's set with 4 point type because of its length. The vertical lines to the left and right mark the beginning and end of the string. The `!` is the first visible character at the 33rd position. The gaps are characters without a drawing within the 256 character set.

A fountain is basically a 256x1 picture scaled to whatever size is required. See figure 13-6.

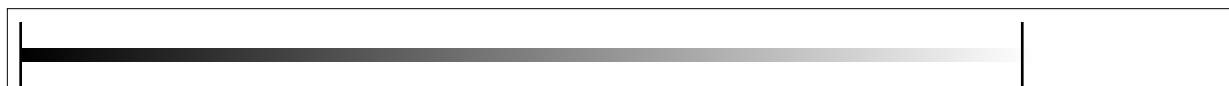


figure 13-6

The scaling of the fountain is handled by both the `image` operator's matrix array and the `scale` operator. In the following example of a basic fountain, first note the picture is 256 wide and 1 high. Second, note the matrix array of `[256 0 0 1 0 0]`. We know from chapter 10 on the matrix that this array will scale the picture by 256 in the `x` and by 1 in the `y`.

The matrix of `[256 0 0 1 0 0]` compresses the 256x1 picture to a 1 by 1 point square. Without the `170 170 scale`, we probably would have a difficult time

seeing the fountain. The `scale` operator scales the fountain to whatever size desired. Since we are beginning with a 1x1 point square, the size of the fountain will also be its scale factor. Below, the `170 170 scale` also means the fountain is 170x170 points in size.



13-12

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_1.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 170 170

/str 256 string def
0 1 255 { str exch dup put } for

170 170 scale

256 1 8 [ 256 0 0 1 0 0 ] {str} image

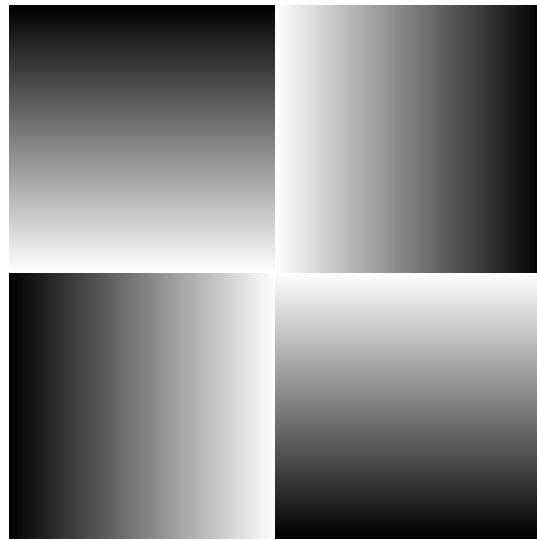
```

13.7 *changing the fountain's direction*

The direction of the fountain can be controlled by using either a 1x256 or 256x1 picture and the matrix array used with the `image` operator. In the next example, note the changes made to the `[256 0 0 1 0 0]` matrix in the other three fountains.



13-13



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_x4.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 200 200

100 100 scale
/str 256 string def
0 1 255 { str exch dup put } for

gsave          % lower left
    256 1 8 [ 256 0 0 1 0 0 ] { str } image
grestore

gsave          % lower right
    1 0 translate          % remember 100 100 scale
    1 256 8 [1 0 0 256 0 0 ] { str } image % different
grestore

gsave          % upper left
    0 1 translate
    1 256 8 [ 1 0 0 -256 0 256] { str } image % different
grestore

gsave          % upper right
    1 1 translate
    256 1 8 [ -256 0 0 1 256 0 ] { str } image % different
grestore

```

13.8

other fountains

There are several techniques in creating a fountain to permit different transitions of gray. Generally the reason is to compensate for different printing situations.



13-14

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_2.eps
%%BoundingBox:0 0 170 170

/str 256 string def
0 1 255 {str exch dup 255 div sqrt 255 mul cvi put} for

/fountain
{ gsave
  /ury exch def /urx exch def
  /lly exch def /llx exch def

  llx lly ury add translate
  urx llx sub ury lly sub scale
  256 1 8 [ 256 0 0 1 0 1 ]{str} image
  grestore } def

0 0 170 170 fountain

```



13-15

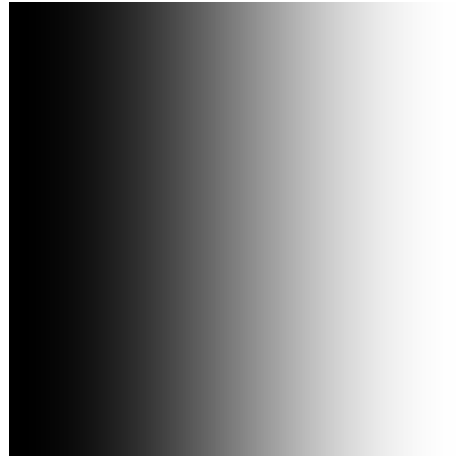
Using a variation of `seefountain_1.eps` from the beginning of section 13.6, the rate of change can be demonstrated by printing the string. `fountain_1.eps` is on the bottom for reference. It is also cropped on the right so we can see the beginning more easily. The vertical lines mark the beginning.

```
| #'*/-/2479;=?ACEGIJLNOQRTUWXZ[^_abcdefghijklmnopqrstuvwxyz{}}~
```

```
| !"#%&'()*+,-./0123456789;<=>?@ABCDEFGHIJKLM
```



13-16



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_3-1.eps
%%BoundingBox:0 0 170 170

/str 256 string def
    0 1 255
    { str exch dup 255 div 180 mul cos neg
      2 div .5 add 255 mul cvi put} for

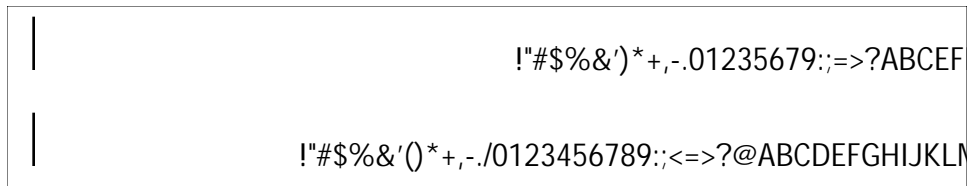
/fountain
{ /ury exch def /urx exch def
  /lly exch def /llx exch def
  gsave
  llx lly translate
  urx llx sub ury lly sub scale
  256 1 8 [ 256 0 0 1 0 0 ]
  {str} image
  grestore} def

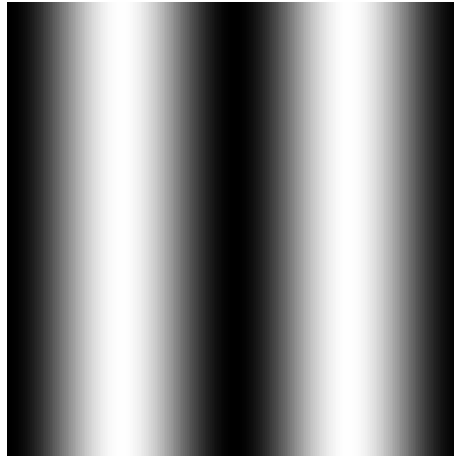
0 0 170 170 fountain
    
```



13-17

`fountain_2.eps` was a shift to the left. You can see below that `fountain_3-1.eps` is a shift to the right compared to `fountain_1.eps`.





13-18

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_3-2.eps
%%BoundingBox:0 0 170 170

/rate 720 def
/str 256 string def
  0 1 255
  { str exch dup 255 div rate mul cos neg
    2 div .5 add 255 mul cvi put} for

/fountain
{ /ury exch def /urx exch def
  /lly exch def /llx exch def
  gsave
  llx lly translate
  urx llx sub ury lly sub scale
  255 1 8 [ 255 0 0 1 0 0 ]
  {str} image
  grestore
} def

0 0 170 170 fountain

```

13-19

This is a variation of `fountain_3-1.eps` on the previous page. Note the new variable titled `rate` used within the `for` procedure. In `fountain_3-1.eps` it was 180. If `rate` is 360, the fountain would be black to white to black. Below is a portion of the string.

```

!%*.39>C!OU[agmsz  ¥«-[]...  CE iløß  œø æ °Ł  }vpjd^XRLFA:61,#  #',16;AFLRX^djpv}  C=fi

```





creative uses of imagemask

The `imagemask` operator is a variation of the `image` operator. It differs in that it can only be used to paint 1-bit pictures, it can reverse the picture, and the picture is painted with the current gray. Another feature of this operator is that the portion of the picture that is not painted is transparent. Otherwise it works much in the same way as `image`.

`imagemask` can be used to make patterns (see chapter 13). One of the suggested uses of `imagemask` is the creation of bit-map fonts. Since the unpainted part of the letterforms is transparent, they would not interfere with images underneath them.

14.1

syntax

The syntax of `imagemask` is:

```
width height invert matrix proc image
```

Where:

<code>width</code>	is the width of the picture in cells or pixels.
<code>height</code>	is the height of the picture in cells or pixels.
<code>invert</code>	if <code>true</code> , values of 1 are painted with the current color; if <code>false</code> , values of 0 are painted with the current color.
<code>matrix</code>	is the picture's coordinate system.
<code>proc</code>	is the data acquisition procedure that obtains or creates the picture data.
<code>imagemask</code>	is the PostScript operator.

14.2

shamrock revisited

When the `invert` boolean is `false`, the 0s are painted with the current color and the 1s are clear. A *boolean* is something that is either true or false. When the `invert` boolean is `true`, the opposite is the case. The 1s are painted and the 0s are clear. See figure 14-1.

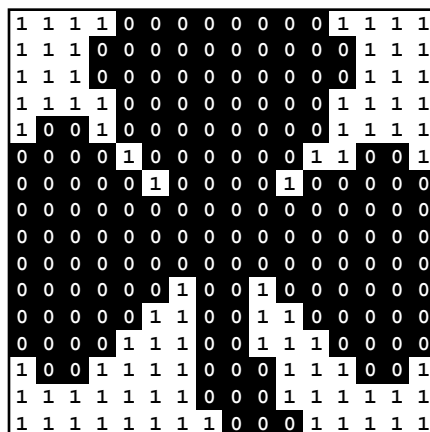


figure 14-1

Using the `shamrock_1.eps` example from chapter 12, the primary differences of `imagemask` can be demonstrated. In the next PostScript program example, the picture is placed on top of a gray square. Had the `image` operator been used, that gray square would have been covered with opaque white.



14-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrock_3.eps
%%BoundingBox:0 0 160 160

/S    <FF 1F FE 3F 9E 39 0E 70 06 60 02 40 00 00 00 00
      00 00 04 20 08 19 90 0F F0 0F E0 07 E0 07 F0 0F> def

.5 setgray
0 0 moveto 160 0 lineto 160 160 lineto 0 160 lineto
closepath fill

0 setgray
16 16 false [.1 0 0 .1 0 0] {S} imagemask

Had true been used in the imagemask argument,
16 16 true [.1 0 0 .1 0 0] {S} imagemask
    
```

the same data would look like figure 14-2. The gray shamrock is the gray showing through where the 0s are and the black is the painted 1s.

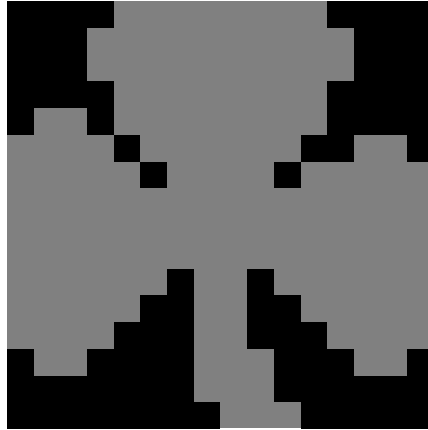
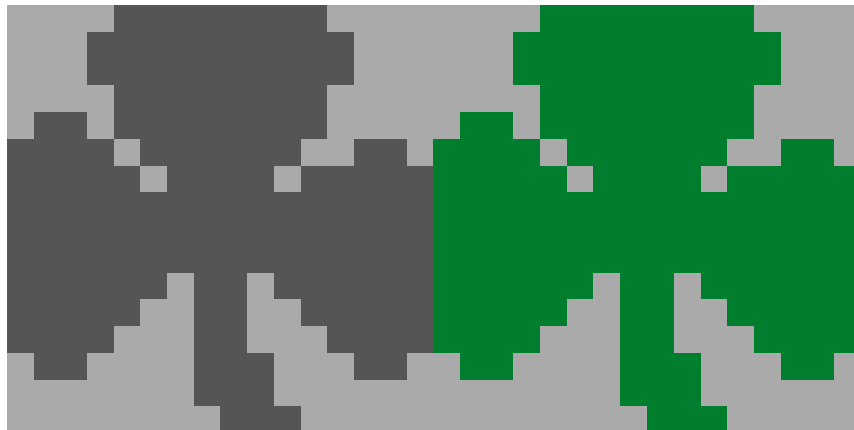


figure 14-2

In the next example, different colors are used for the picture. The background square is `.666 setgray`. The first shamrock is `.333 setgray`. For the shamrock to the right, the current color is made to `1 0 1 0.3 setcmykcolor`, a dark green.



14-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shamrock_5&6.eps
%%BoundingBox:0 0 320 160

/S <FF 1F FE 3F 9E 39 0E 70 06 60 02 40 00 00 00 00
  00 00 04 20 08 19 90 0F F0 0F E0 07 E0 07 F0 0F> def

.666 setgray % background
0 0 moveto 320 0 lineto 320 160 lineto 0 160 lineto
closepath fill

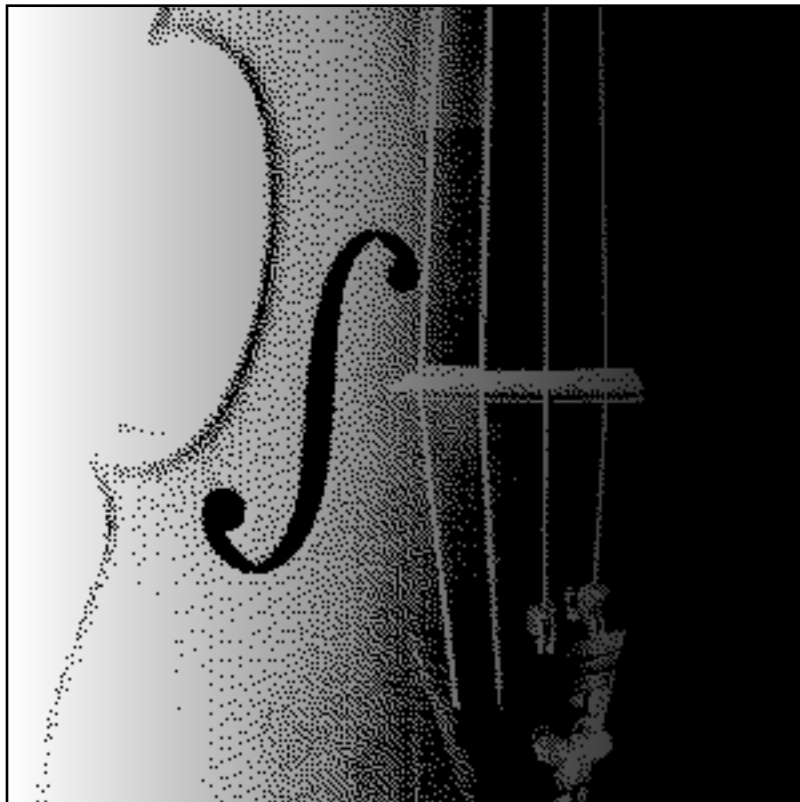
.333 setgray
16 16 false [.1 0 0 .1 0 0] {S} imagemask

160 0 translate
1 0 1 .3 setcmykcolor
16 16 false [.1 0 0 .1 0 0] {S} imagemask

```

14.3 *with a fountain*

An interesting effect with `imagemask` can be achieved with a fountain (see previous chapter, sections 13.6 – 13.8) underneath the picture. Following are two designs set up basically the same. Each has the same fountain underneath the picture of the violin painted with `imagemask`. In the first example, `violinSql.eps`, the violin is painted in the default value of black.



learn PS

14-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:violinSql.eps
%%BoundingBox: 0 0 300 300

gsave
  300 300 scale
  /str 256 string def
  0 1 255 { str exch dup put } for
  256 1 8 [ -256 0 0 1 256 0 ] { str } image
grestore

gsave
  0 setgray
  /picstr1 38 string def
  /readdata {currentfile exch readhexstring pop} def
  /beginimage {{picstr1 readdata} imagemask} def
  300 300 scale

```

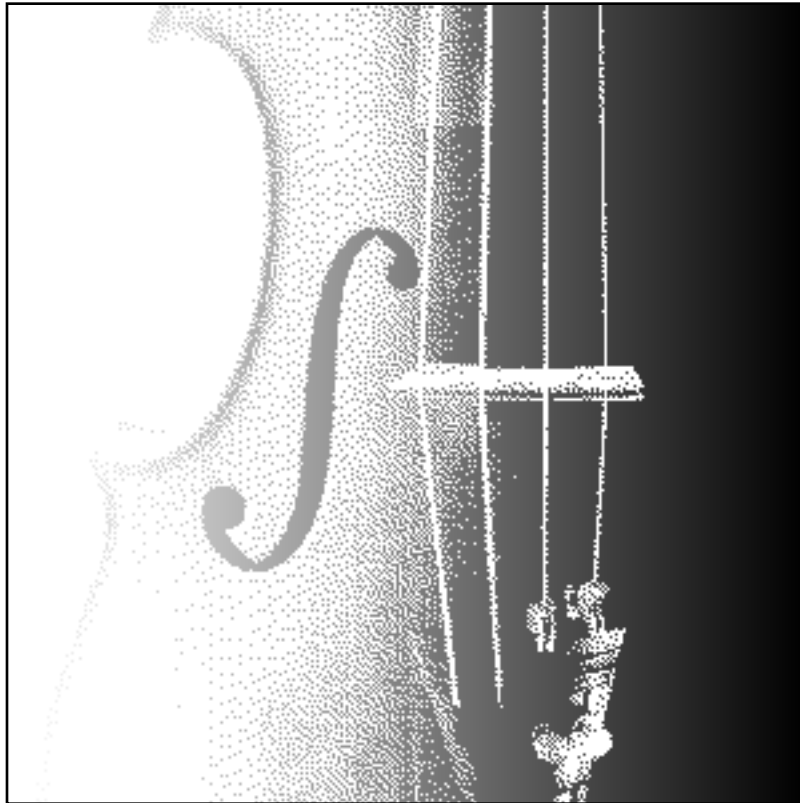
```

    300 300 false [300 0 0 300 neg 0 300]
beginimage
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF69FEA800C00002000020000
... PICTURE DATA ...
FFF7FFFFFFFFFFFFFFFF7FFFAFBD54ED6AA52222004410000000363800000
grestore

0 0 300 300 rectstroke

```

In this second example, two changes have been made to the file. First, `true` is used as the invert boolean for `imagemask`, making the opposite pixels be painted 1s. Second, the current gray is made white with `1 setgray`. In this way, whatever is underneath will give the violin its color. This creates the effect that the violin is being painted as a fountain. The white is a mask covering the fountain below it.



learn

14-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:violinSq2.eps
%%BoundingBox: 0 0 300 300

gsave
    300 300 scale
    /str 256 string def
    0 1 255 { str exch dup put } for
    256 1 8 [ -256 0 0 1 256 0 ] { str } image
grestore

```

```

gsave
  1 setgray
  /picstr1 38 string def
  /readdata {currentfile exch readhexstring pop} def
  /beginimage {{picstr1 readdata} imagemask} def
  300 300 scale
  300 300 true [300 0 0 300 neg 0 300]
beginimage
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF69FEA800C00002000020000
... PICTURE DATA ...
FFF7FFFFFFFFFFFFFFFF7FFFAFBD54ED6AA52222004410000000363800000
grestore

0 0 300 300 rectstroke

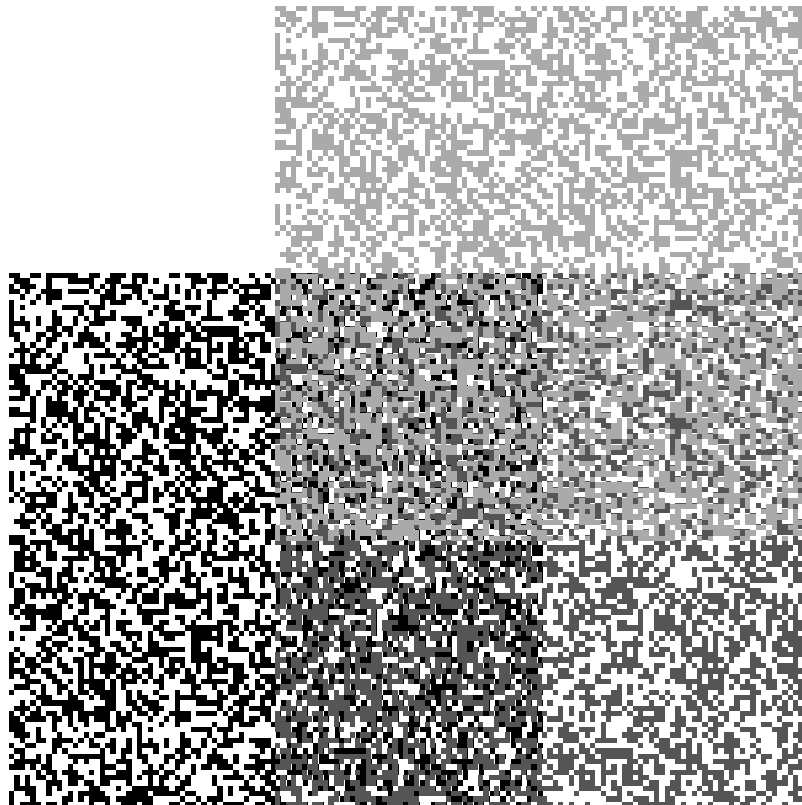
```

14.4 *as a pattern*

Random patterns can be made with a similar procedure used to make patterns in section 13.4.

```
0 1 511 {str exch rand 255 mod put} for
```

This procedure converts `str` into a string of 512 randomly selected characters. Three overlapping squares are made, each a different value.





14-5

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:imagemaskPat_1.eps
%%BoundingBox:0 0 300 300

/str 512 string def
0 1 511 {str exch rand 255 mod put} for

100 100 true [.5 0 0 .5 0 0] {str} imagemask

100 0 translate
.333 setgray
100 100 true [.5 0 0 .5 0 0] {str} imagemask

0 100 translate
.666 setgray
100 100 true [.5 0 0 .5 0 0] {str} imagemask
```




creating a font

You can create a unique PostScript font and use it in your PostScript programs. The PostScript examples in this chapter will cover how to modify an existing font, the creation of a mono-spaced font, and then a variable-spaced font. We will be creating type 3 fonts. The creation of type 1 fonts, however, will not be covered. Type 1 fonts are Adobe's previously encrypted font format that was recently published. These fonts contain procedures, commonly known as hints, that ensure high quality output on all resolutions of printers, especially small type on low resolution devices.

15.1 *anatomy of a font program*

A PostScript font program is the creation of a font dictionary containing the values for a number of font characteristics, some of which are contained in their own dictionary within the larger font dictionary. Earlier in chapter 3, you may remember, we defined `square` to be the name of a procedure that draws a square. Within a font dictionary, `/a` is the name of a procedure that draws the character `a`. There is more to it than just this, but essentially a font is a dictionary of drawings for each character of the alphabet. There are also other attributes of the font defined in the file, such as the `FontType`, `FontMatrix`, `FontBBox`, the character Encoding, and its font metric.

The `FontType` tells the PostScript interpreter how the font was made so it knows how to handle the font. The different font types are:

- 0 Kanji or other very large font sets.
- 1 Adobe's encrypted and hinted fonts.
- 2 Obsolete.
- 3 User defined fonts.
- 4 Cartridge and disk-based encrypted fonts.
- 5 ROM-based encrypted fonts.

The `FontMatrix` is the coordinate system within which the font was designed. Usually the font is designed within a 1000x1000 unit space. It can be thought of as the page the font was designed on. In figure 15-1, the characters `g` and `M` are placed on what could be the 1000x1000 unit space. Each grid line represents 100 units. A matrix array of `[.001 0 0 .001 0 0]` will reduce the 1000x1000 unit space to a 1x1 unit square. In other words, what was drawn to be 100 is now .1 and what was 1000 is 1. Therefore, when `72 scalefont` is used with the font, the 1x1 unit square is scaled by a factor of 72.

Figure 15-2 explains `FontMatrix` further. The first large square grid represents

the 1000x1000 unit space (reduced 10% to save space) on which the font was drawn. Next to it, labeled with an **A** and with an arrow pointing at it (it's small) is that 1000x1000 unit space reduced to a 1x1 unit space by the application of the `[.1 0 0 .1 0 0] FontMatrix`. **B** labels what would happen with a 72 `scalefont`, **C** labels a 300 `scalefont` enlargement for a better view.

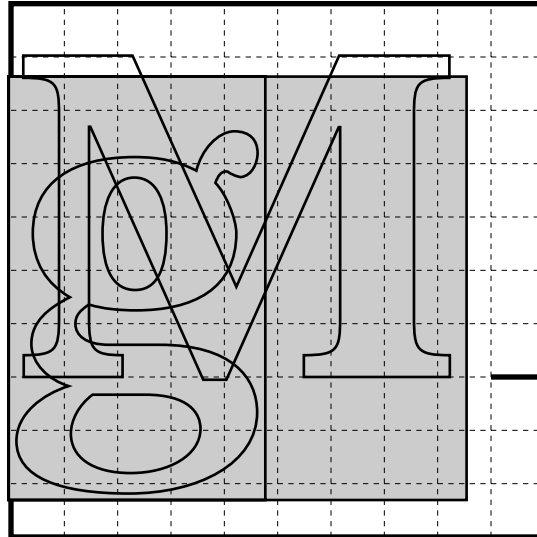


figure 15-1

The `FontBBox` is the smallest rectangle into which all the characters of the font will fit. This is depicted by the gray rectangle in figure 15-1.

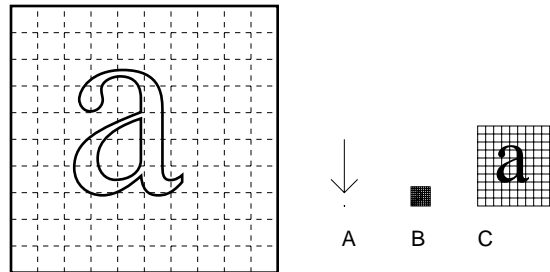


figure 15-2

The character **Encoding** determines at what decimal location the character name will reside on the ASCII chart. See appendix A. PostScript has a standard **Encoding** named `StandardEncoding` that already gives every entry of the array a name. In it, `/one` is the name for the number 1 at decimal 49, `/a` is the name for the character a at decimal 97, and so on. The `StandardEncoding` can be copied or you can create your own scheme. In the next section, the `StandardEncoding` is copied, saving the trouble of having to name all the character procedures. Later, in section 15.4, custom names are given to several of the characters and the name for a null procedure (doesn't do anything) to the rest.

The `CharProcs` is a dictionary containing all the procedure definitions that draw the individual characters named in the `Encoding` array.

Finally, there is the `BuildChar` procedure that uses the information previously listed in the program to actually draw the character.

15.2

font template

This next section has a listing of a font file to show how the above information is organized into a file. All that is missing are your drawings for individual characters and your choice for a font name. Not all 256 entries are included. The others can be found in appendix A.



15-1

```

%!PS-Font
%%Title:font_template.ps

/newfont 10 dict def      % create dictionary
newfont begin            % begin filling dictionary

/FontType 3 def          % user defined font
/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 1000 1000] def

/Encoding 256 array def % copy standard encoding
StandardEncoding Encoding copy pop

/CharProcs 20 dict def  % match #with # of char proc
CharProcs begin         % dictionary of char procedures

    /space { } def      % word space
    /ampersand { % character drawing for & } def
    /asterisk { % character drawing for * } def

    /zero { % character drawing for 0 } def
    /one { % character drawing for 1 } def
    /two { % character drawing for 2 } def
    /three { % character drawing for 3 } def
    /four { % character drawing for 4 } def

    /A { % character drawing for A } def
    /B { % character drawing for B } def
    /C { % character drawing for C } def
    /D { % character drawing for D } def
    /a { % character drawing for a } def
    /b { % character drawing for b } def
    /c { % character drawing for c } def
    /d { % character drawing for d } def

    /yen { % character drawing for ¥ } def
    /paragraph { % character drawing for ¶ } def
    /germandbls { % character drawing for ß } def
end

/BuildChar % creates char from info above
{1000 0 0 0 1000 1000 setcachedevice
exch begin
Encoding exch get
CharProcs exch get
exec end} def

```

```
end
```

```
% give your font a name replacing YourName
/YourName newfont definefont pop
```

15.3 *modifying an existing font*

It's possible to modify an existing font by duplicating its font dictionary and then placing a new entry in the copy. The most common change is to change the font's matrix. By doing so, the entire font can be condensed or expanded. The font Helvetica-Narrow found in many laser printers is produced by mathematically condensing Helvetica. Helvetica Condensed, on the other hand, is a separate drawing from Helvetica.

In this example, we will make our own Palatino Bold Narrow. The first section of code duplicates the Palatino-Bold dictionary and discards the existing `FID`. A new one will be made with `definefont`. While `dictCopy` is on top of the dictionary stack, the `FontMatrix` is replaced with a new array. The next line defines `Palatino-BNarrow` as the font with `definefont`.



15-2

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:Palatino-BNarrow.eps
%%BoundingBox:18 36 373 156

/Palatino-Bold findfont
dup length dict /dictCopy exch def
  {1 index /FID ne
    {dictCopy 3 1 roll put}
    {pop pop}
  }ifelse }forall

dictCopy /FontMatrix [.0005 0 0 .001 0 0] put

/Palatino-BNarrow dictCopy definefont pop
```

```
/Palatino-BNarrow findfont 70 scalefont setfont
18 36 moveto (Palatino Bold Narrow) show
```

```
/Palatino-Bold findfont 70 scalefont setfont
18 108 moveto (Palatino B) show
```

15.4 *shapeFont as mono-spaced font*

To create our own font from scratch, we'll need to set up all the necessary information. A mono-spaced font is simpler than a variable-spaced font because every character uses the same amount of space. First, the font dictionary is set up with `/newfont 10 dict def`. `newfont` is an arbitrary name; use whatever you want. It will later be replaced with the actual font name. The `newfont` dictionary must have four entries: `FontType`, `FontMatrix`, `FontBBox` and `Encoding`. These should be understood from section 15.1. The `Encoding` this time will be a nonstandard array of 256 names. The line

```
0 1 255 {Encoding exch /.notdef put} for
```

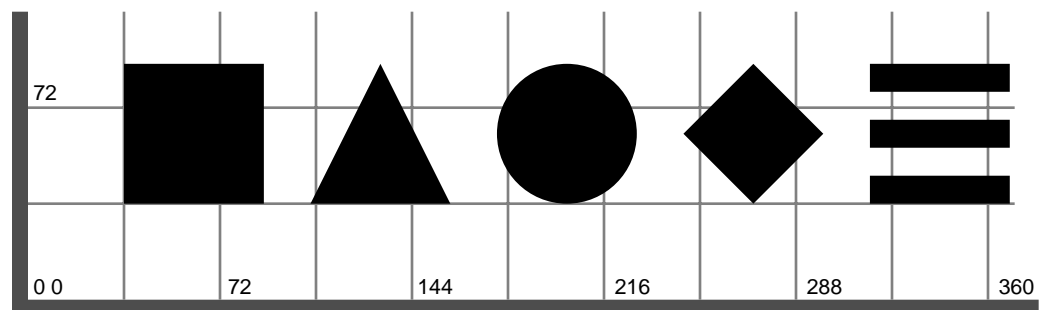
replaces each position of the “blank” `Encoding` made by the previous line

```
/Encoding 256 array def
```

with the key `/.notdef`. This is a quick way to give each entry of `Encoding` a name. We plan to name only 5 of the 256 positions. If a character without a drawing is used, the `.notdef` procedure (which does nothing) will be used to prevent an error. Next, the names for characters we want in `Encoding` are put into the position we want that corresponds to a key of the keyboard. Basically, ASCII 97 is the character `a` in a standard font and is the character `square` in `shapeFont`.

`CharProcs` is a dictionary that matches the name in `Encoding` with a procedure. The procedure definitions should look familiar to you. Note what the procedure `.notdef` is defined as.

Finally, there is the definition for `BuildChar`. `BuildChar` is what gets called when a character needs to be painted. Font caching, performed by the `setcachedevice` operator is used to gain efficiency in printing. When a character is drawn by `BuildChar`, its image is temporarily stored in a cache. Should the character be used again, it's there, ready to go.



15-3

```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:shapeFont.eps
%%BoundingBox:0 0 385 150
```

```

/newfont 10 dict def
newfont begin

/FontType 3 def
/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 1000 1000] def

/Encoding 256 array def          % give name
0 1 255 {Encoding exch /.notdef put} for
Encoding 97  /square put        % keyboard a
Encoding 98  /triangle put      % keyboard b
Encoding 99  /circle put       % keyboard c
Encoding 100 /diamond put      % keyboard d
Encoding 101 /lines put        % keyboard e

/CharProcs 6 dict def
CharProcs begin                  % proc def for names above
  /.notdef { } def
  /square
    {0 0 moveto 750 0 lineto 750 750 lineto
     0 750 lineto closepath fill} def
  /triangle
    {0 0 moveto 375 750 lineto 750 0 lineto
     closepath fill} def
  /circle
    {375 375 375 0 360 arc closepath fill} def
  /diamond
    {375 0 moveto 750 375 lineto 375 750 lineto
     0 375 lineto closepath fill} def
  /lines
    {150 setlinewidth
     0 75 moveto 750 0 rlineto stroke
     0 375 moveto 750 0 rlineto stroke
     0 675 moveto 750 0 rlineto stroke} def
  end

/BuildChar
  {1000 0 0 0 750 750 setcachedevice
  exch begin
  Encoding exch get
  CharProcs exch get
  exec
  end} def
end                                % end newfont

/ShapeFont newfont definefont pop

/ShapeFont findfont 70 scalefont setfont

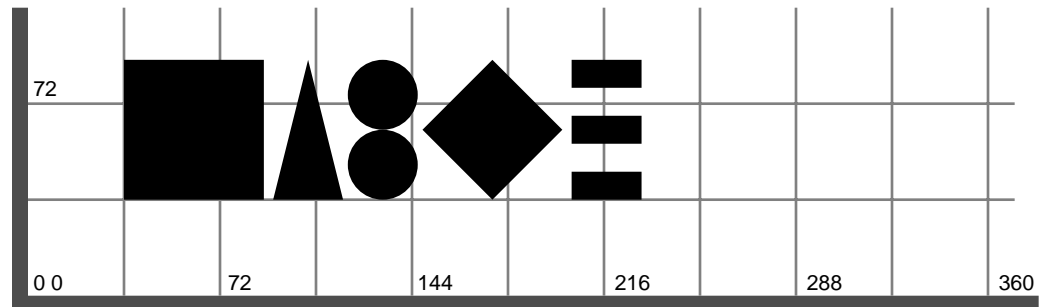
36 36 moveto (abcde) show

```


15.5

shapeFont as variable-spaced font

shapeFont as a variable-spaced font is written much the same as in the previous section. In this version of the font, every character can have its own unique width. The primary difference in this font file from the last section is the addition of the **Metrics** and **BBox** dictionaries. These are used for character spacing and setting the boundaries of font caching respectively. These are used in the **BuildChar** procedure. If the values are changed in the **Metrics** dictionary, the character spacing of this font will be different. The **BBox** dictionary provides more precise information for font caching.



15-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:shapeFont2.eps
%%BoundingBox:0 0 385 150

/newfont 10 dict def
newfont begin

/FontType 3 def
/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 750 750] def

/Encoding 256 array def
0 1 255 {Encoding exch /.notdef put} for
Encoding 97 /square put           % keyboard a
Encoding 98 /triangle put        % keyboard b
Encoding 99 /circles put         % keyboard c
Encoding 100 /diamond put       % keyboard d
Encoding 101 /lines put         % keyboard e

/Metrics 6 dict def
Metrics begin
  /.notdef 0 def
  /square 800 def
  /triangle 400 def
  /circles 400 def
  /diamond 800 def
  /lines 400 def
end

/BBox 6 dict def

```

```

BBox begin
  /.notdef [0 0 0 0] def
  /square [0 0 750 750] def
  /triangle [0 0 375 750] def
  /circles [0 0 375 750] def
  /diamond [0 0 750 750] def
  /lines [0 0 375 750] def
end

/CharProcs 6 dict def
CharProcs begin
  /.notdef { } def
  /square
    {0 0 moveto 750 0 lineto 750 750 lineto
    0 750 lineto closepath fill} def
  /triangle
    {0 0 moveto 375 0 lineto 187.5 750 lineto
    closepath fill} def
  /circles
    {187.5 562.5 187.5 0 360 arc closepath fill
    187.5 187.5 187.5 0 360 arc closepath fill} def
  /diamond
    {375 0 moveto 750 375 lineto 375 750 lineto
    0 375 lineto closepath fill} def
  /lines
    {150 setlinewidth
    0 75 moveto 375 0 rlineto stroke
    0 375 moveto 375 0 rlineto stroke
    0 675 moveto 375 0 rlineto stroke} def
end

/BuildChar
  {0 begin
  /char exch def
  /fontdict exch def
  /charname fontdict /Encoding get char get def
  fontdict begin
    Metrics charname get 0
    BBox charname get aload pop setcachedevice
    CharProcs charname get exec
  end
  end } def

/BuildChar load 0 3 dict put
/UniqueID 1 def
end

/ShapeFont2 newfont definefont pop

/ShapeFont2 findfont 70 scalefont setfont

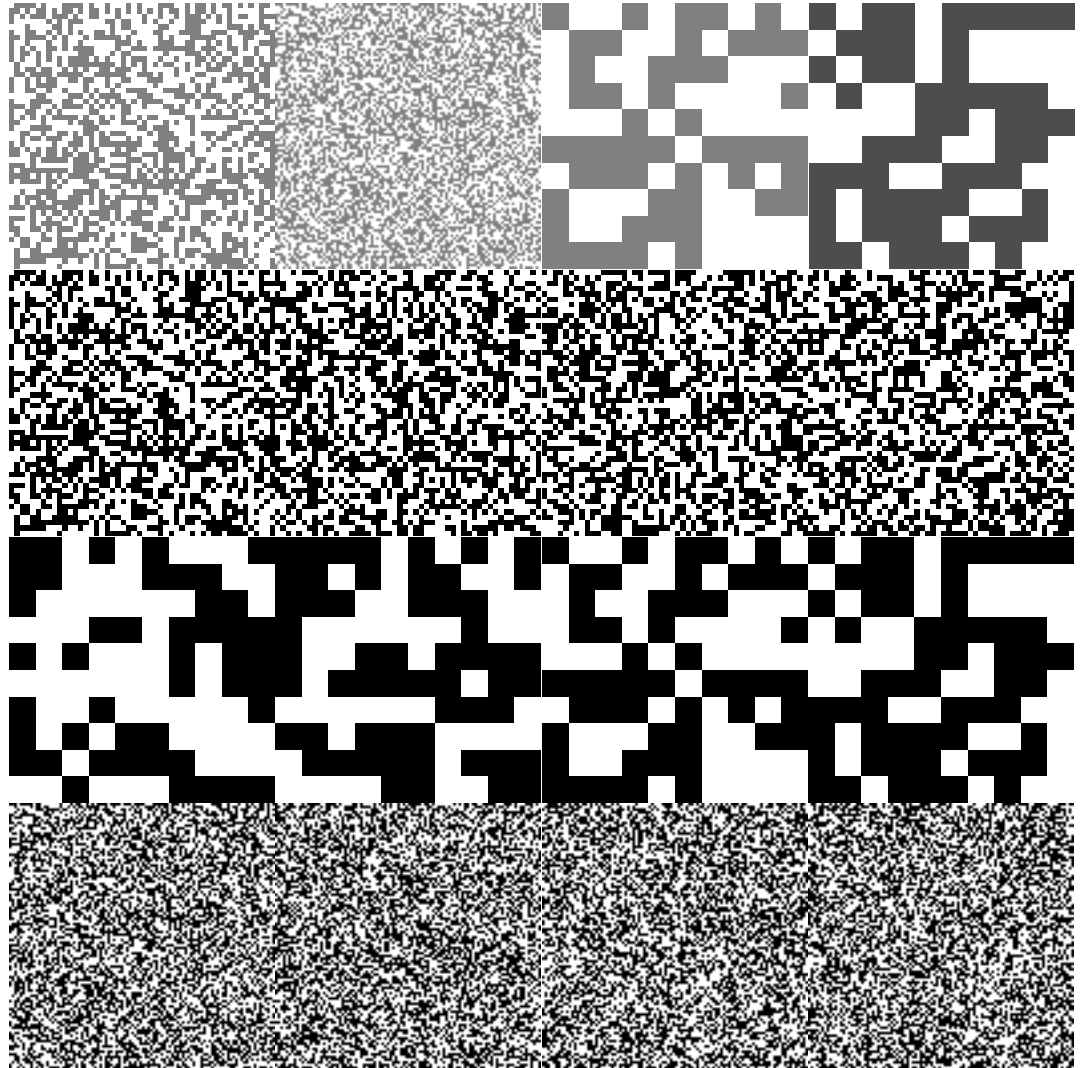
36 36 moveto (abcde) show

```

15.6

the radBit-Roman font

The radBit-Roman font is an example of how the PostScript font machinery can be used to create interesting patterns. It is a mono-spaced font with no gap between the characters.



15-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:radBit_font.eps
%%Creator:John F Sherman
%%CreationDate:April 1990
%%DocumentFonts:radBit-Roman
%%BoundingBox:0 0 400 400

/newfont 10 dict def
newfont begin

/FontType 3 def

```

```

/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 1000 1000] def

/Encoding 256 array def
StandardEncoding Encoding copy pop

/CharProcs 30 dict def
CharProcs begin

    /.notdef { } def
    /str 512 string def
    /rBit {
        0 1 511
        {CharProcs /str get exec exch rand 255 mod put}
        for CharProcs
        /str get exec } def

    /space {0 0 moveto newpath} bind def

    /zero {48 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /one {49 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /two {50 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /three{51 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /four {52 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /A {65 srand
        10 10 true [.01 0 0 .01 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /B {66 srand
        10 10 true [.01 0 0 .01 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /C {67 srand
        10 10 true [.01 0 0 .01 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /D {68 srand
        10 10 true [.01 0 0 .01 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

```

```

/a    {97 srand
      50 50 true [.05 0 0 .05 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/b    {98 srand
      50 50 true [.05 0 0 .05 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/c    {99 srand
      50 50 true [.05 0 0 .05 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/d    {100 srand
      50 50 true [.05 0 0 .05 0 0]
      {CharProcs /rBit get exec}imagemask} bind def
end

/BuildChar
  {1000 0 0 0 1000 1000 setcachedevice
  exch begin
  Encoding exch get
  CharProcs exch get
  exec end} def
end

/radBit-Roman newfont definefont pop

%% BEGIN DESIGN

/radBit-Roman findfont 100 scalefont setfont

gsave
  0 0 moveto (1234) show
  0 100 moveto (ABCD) show
  0 200 moveto (abcd) show
  .5 setgray
  0 300 moveto (a1C) show .3 setgray (D) show
grestore

```

15.7 *how to use your font*

There are two ways you can use your font. The first way is to include the definition of the font in the beginning of your program. This has been done in the previous examples. The advantage of this is that your program can easily be run on any PostScript printer. Everything is there in the PostScript file ready to go.

The second way to use your font is to use it only as a font file and download it to your laser printer's RAM. Thereafter, your PostScript files that use the font will not need to contain the definition for the font. The main advantage to downloading your font is that you can simplify your PostScript files. The disadvantages are that you can lose track of which files use which fonts and there isn't a convenient way to let Macintosh applications use them.



15-6

To be able to download a font, you need to add a line of code to the beginning of your font file:

```
serverdict begin 0 exitserver
```

This line at the beginning of your file will let your font exist in your printer's RAM until it is turned off. Your file would look like this.

```

%!PS-Font
%%Title:download_rFont.ps

serverdict begin 0 exitserver

/newfont 10 dict def
newfont begin

/FontType 3 def
/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 1000 1000] def

/Encoding 256 array def
0 1 255 {Encoding exch /.notdef put} for

Encoding 33 /33 put
Encoding 65 /65 put
Encoding 97 /97 put
Encoding 122 /122 put

/CharProcs 7 dict def
CharProcs begin
  /.notdef{ } def
  /char {10 setlinewidth rand 1000 mod rand 1000 mod
moveto
          rand 1000 mod rand 1000 mod lineto stroke} def

/33 {33 {CharProcs /char get exec} repeat} bind def
/65 {65 {CharProcs /char get exec} repeat} bind def
/97 {97 {CharProcs /char get exec} repeat} bind def
/122 {122 {CharProcs /char get exec} repeat} bind def
end

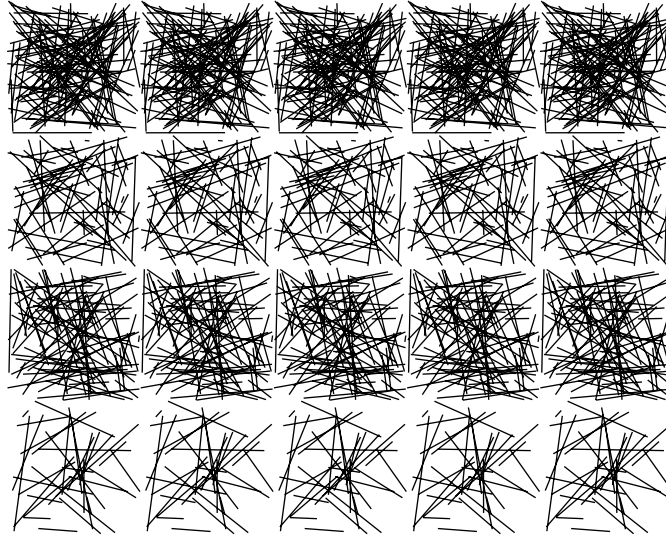
/BuildChar
  {1000 0 0 0 1000 1000 setcachedevice
  exch begin
  Encoding exch get
  CharProcs exch get
  exec
  end} def
end

/rFont newfont definefont pop

```

If the previous program has been sent to the LaserWriter, the following program

can be sent to the printer and work correctly.



15-7

```

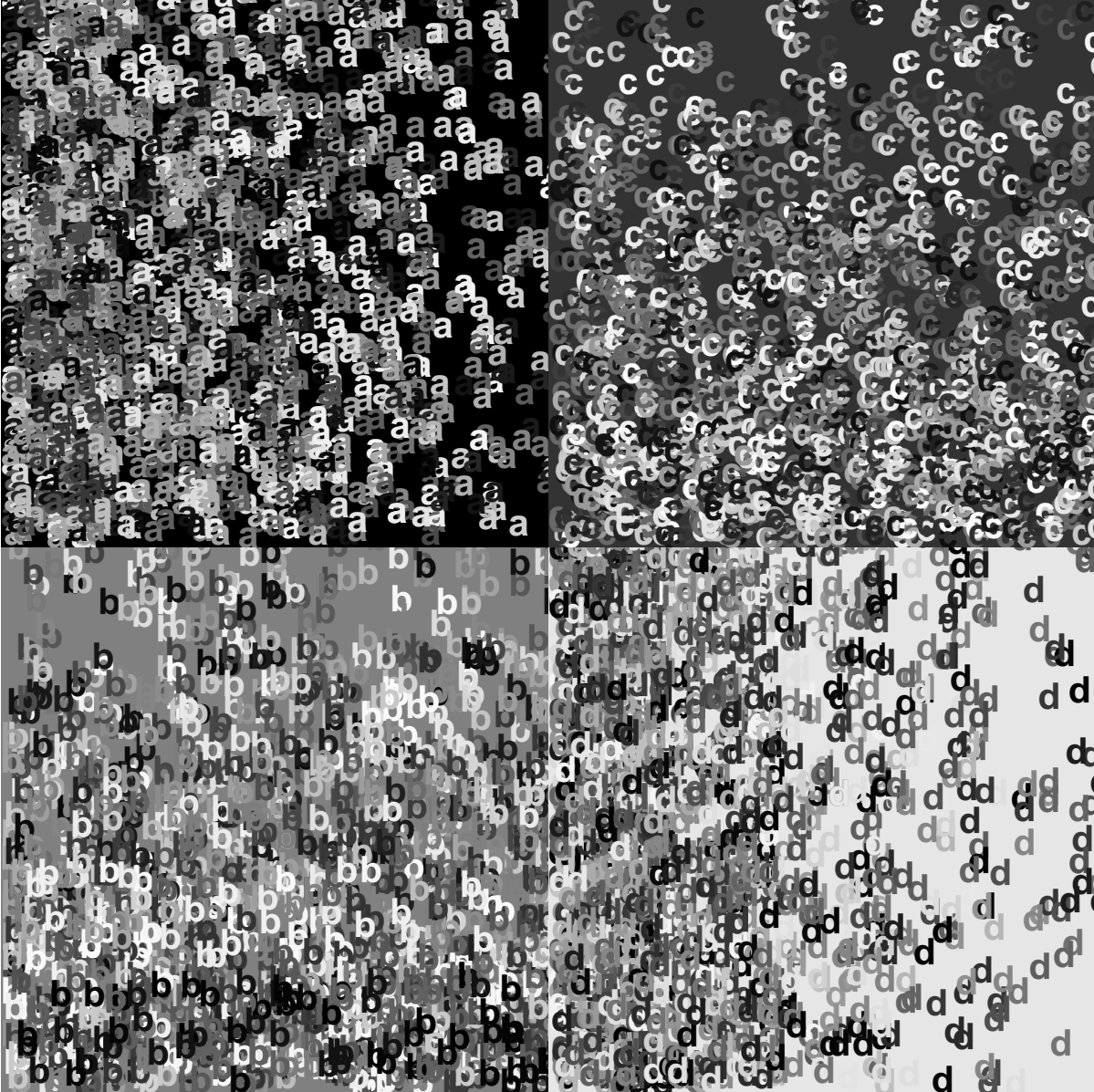
%!PS-Adobe-2.0 EPSF-1.2
%%Title:rFont.eps
%%BoundingBox:25 25 275 225

/rFont findfont 50 scalefont setfont

25 25 moveto (!!!!) show
25 75 moveto (aaaa) show
25 125 moveto (AAAA) show
25 175 moveto (zzzz) show

```

Generally, you'll find it easier to keep the font definition within the PostScript file.





creative uses of random numbers

The random number generation of PostScript has a number of creative applications. Most common are random numbers for an `x y` location for a `moveto` or `lineto` operator or to set a gray value. The sequence of random numbers can be repeated if the seed number that begins the sequence is the same.

16.1

rand

The simplest random number program would be

```
rand =
```

rand creates and pushes a random number onto the operand stack. The equal sign (=) pops the number off the stack and sends it to the standard output file. Depending on the application you are using at the time, the standard output file will be different. On the Macintosh, if you are using *SendPS*, the standard output file will be a text file on your hard disk named after your printer. If you are using *Downloader*, the output file is a brief display in a window on the monitor. On the NeXT, the standard output file will be the Console window found in the Tools menu under the main Workspace menu. By running the following,

```
rand =  
rand =  
rand =  
rand =  
rand =  
rand =  
rand =
```

you will send seven numbers to the standard output file.

```
507111939  
1815247477  
1711656657  
1717468248  
1161144809  
1176904574  
1910786348
```

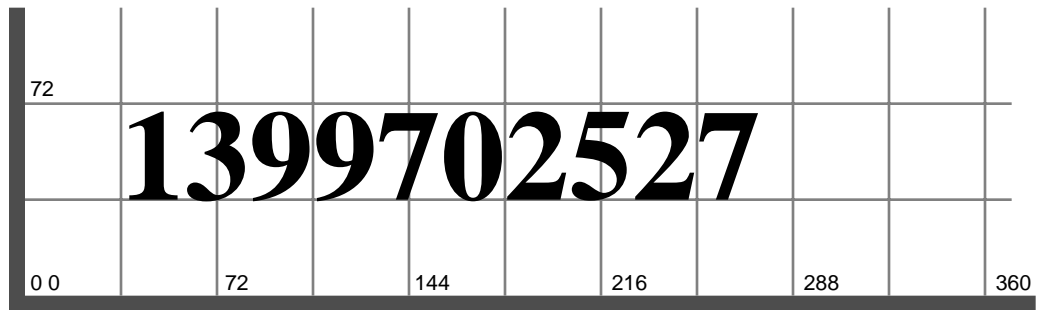
You won't get these numbers. It will be a different set every time. Another way to see the number created is to print it:



16-1



16-2



```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:seeRand1.eps
%%DocumentFonts:Times-Bold
%%BoundingBox:36 36 288 72

/Times-Bold findfont 48 scalefont setfont
/str 10 string def

36 36 moveto rand str cvs show
    
```

16.2

srand

In the previous example, every time the program is run, a different number will be generated. By using the operator `srand`, which can be understood as *seed*random number, a repeatable series of numbers can be created. The syntax of `srand` is:

`integer srand`

Where `integer` is the seed number.

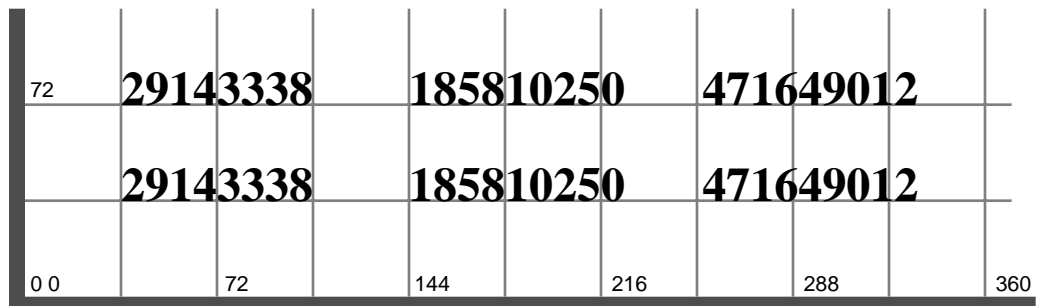


16-3

```

1734 srand
rand =
rand =
rand =
rand =
rand =
rand =
rand =
    
```

Every time this is sent to the PostScript interpreter, the same seven numbers will be generated. Following is an expanded version of `seeRand1.eps`.





16-4

```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:seeRand2.eps
%%DocumentFonts:Times-Bold
%%BoundingBox:36 36 335 90

/Times-Bold findfont 18 scalefont setfont
/str 10 string def

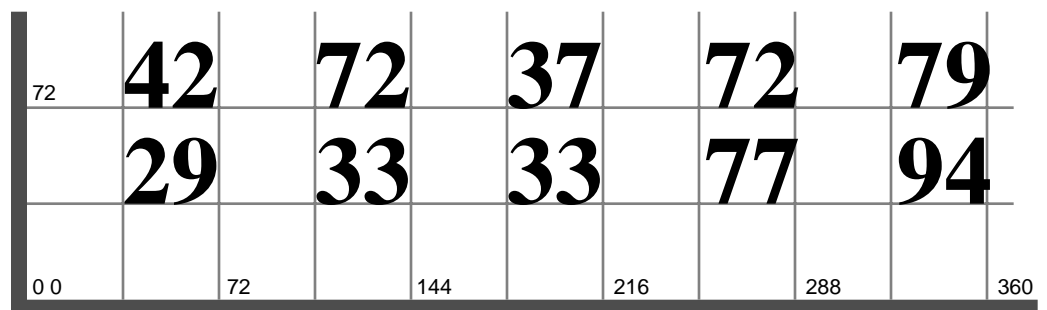
1734 srand
36 36 moveto rand str cvs show
144 36 moveto rand str cvs show
254 36 moveto rand str cvs show

1734 srand
36 72 moveto rand str cvs show
144 72 moveto rand str cvs show
254 72 moveto rand str cvs show

```

16.3 *random x y placement*

The PostScript random number generator can be used to supply values for the `moveto` or `lineto` operators. The random number created, however, needs to be confined to a certain range so that there is a reasonable expectation of where items will be located. Otherwise, you'll be off the page most of the time. Creating a random number that is always within a certain range can be done with the `mod` operator. `mod` divides the top two integers on the stack and pushes the division's remainder onto the stack. For example, `77 10 mod` would return 7. No matter what the first integer is, the remainder will always be between 0 and the divider minus 1. In the following example, 10 random numbers are generated between 0 and 100. A different set can be made if the `srand` value is changed.



16-5

```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:seeRand3.eps
%%DocumentFonts:Times-Bold
%%BoundingBox:36 36 362 100

/Times-Bold findfont 36 scalefont setfont
/str 20 string def
/range 100 def
/rNum {rand range mod} def

```

```

173468 srand

36 36 moveto rNum str cvs show
108 36 moveto rNum str cvs show
180 36 moveto rNum str cvs show
254 36 moveto rNum str cvs show
326 36 moveto rNum str cvs show

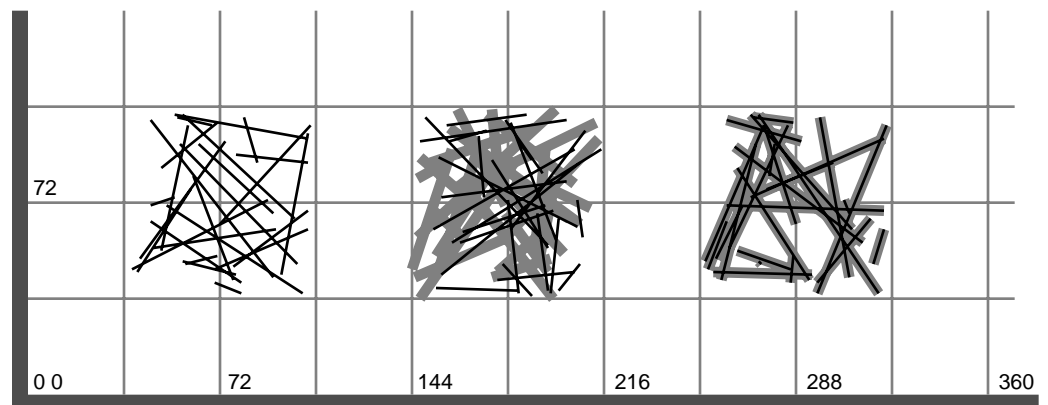
36 72 moveto rNum str cvs show
108 72 moveto rNum str cvs show
180 72 moveto rNum str cvs show
254 72 moveto rNum str cvs show
326 72 moveto rNum str cvs show

```

To take advantage of these numbers to make a graphic, you can write:

```
rNum rNum moveto rNum rNum lineto stroke
```

This will draw a line from a random start to a random end. The beginning and the end will be within whatever range is decided. The next step is to repeat the line so that various patterns or textures can be created. In the following example, the first cluster is an example of repeating the line of code above. The second cluster is an example of using the repetition twice with a change in line weight and value. The third cluster of lines demonstrates how by using `srand` with the same seed before drawing the cluster of lines, the same pattern of lines can be made and overlapped.



16-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:randLines.eps
%%BoundingBox:36 36 324 108

/range72 def
/rNum {rand range mod} def

36 36 translate % first cluster
25 {rNum rNum moveto rNum rNum lineto stroke} repeat

108 0 translate % second cluster
.5 setgray
4 setlinewidth

```

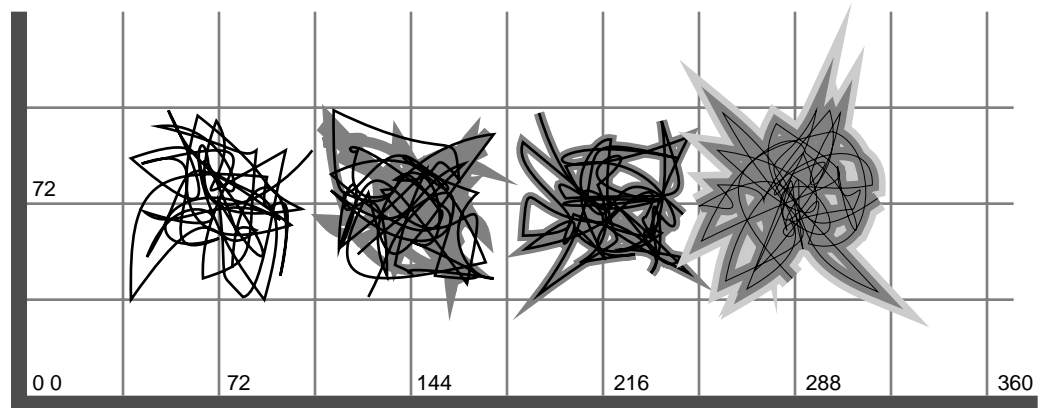
```

25 {rNum rNum moveto rNum rNum lineto stroke} repeat
0 setgray
1 setlinewidth
25 {rNum rNum moveto rNum rNum lineto stroke} repeat

108 0 translate    % third cluster
173468 srand
.5 setgray
4 setlinewidth
25 {rNum rNum moveto rNum rNum lineto stroke} repeat
173468 srand
0 setgray
1 setlinewidth
25 {rNum rNum moveto rNum rNum lineto stroke} repeat

```

Following is a similar example using the `curveto` operator.



learn

16-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:randcurves.eps
%%BoundingBox:36 24 362 140

/n    {rand 72 mod} def

36 36 translate    % first scribble
n n moveto
25    {n n n n n n curveto} repeat stroke

72 0 translate    % second scribble
.5 setgray
4 setlinewidth
n n moveto
25    {n n n n n n curveto} repeat stroke
0 setgray
1 setlinewidth
n n moveto
25    {n n n n n n curveto} repeat stroke

72 0 translate    % third scribble

```

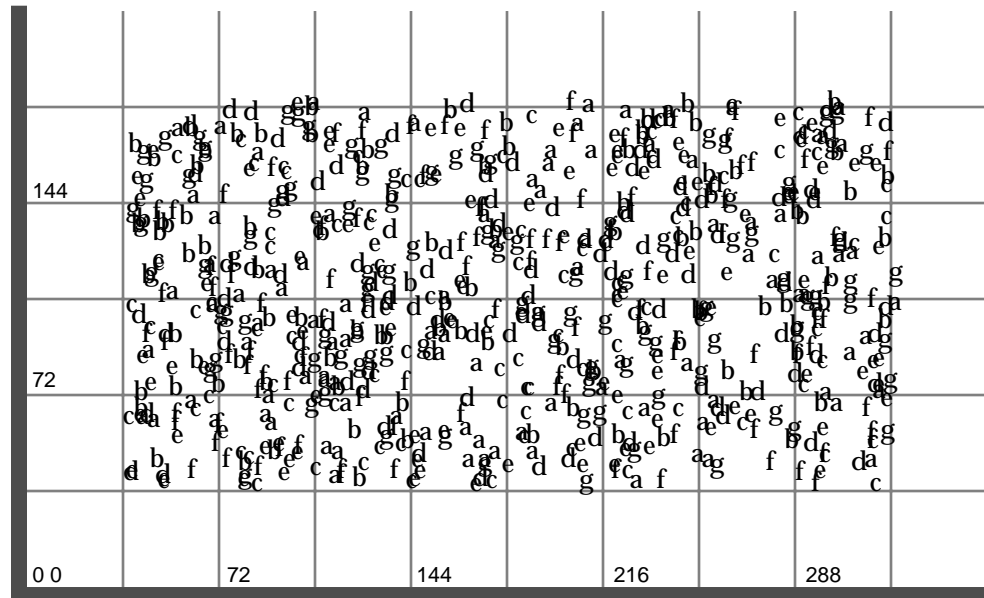
```

192837465 srand
    .5 setgray
    4 setlinewidth
    n n moveto
    25 {n n n n n n curveto} repeat stroke
192837465 srand
    0 setgray
    1 setlinewidth
    n n moveto
    25 {n n n n n n curveto} repeat stroke

72 0 translate % fourth scribble
1734173417 srand % bottom
    .8 setgray
    10 setlinewidth
    n n moveto
    25 {n n n n n n curveto} repeat stroke
1734173417 srand % middle
    .5 setgray
    5 setlinewidth
    n n moveto
    25 {n n n n n n curveto} repeat stroke
1734173417 srand % top
    0 setgray
    .25 setlinewidth
    n n moveto
    25 {n n n n n n curveto} repeat stroke
    
```

16.4 random placement of type

The random placement of type is essentially accomplished in the same way as in the previous section. Here we generate separate *x* and *y* values.





16-8

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:randAlpha1.eps
%%BoundingBox:36 36 330 184

/x {rand 288 mod} def % x will be between 0 & 288
/y {rand 144 mod} def % y will be between 0 & 144

/Palatino-Roman findfont 10 scalefont setfont
36 36 translate

100 {x y moveto (a) show} repeat
100 {x y moveto (b) show} repeat
100 {x y moveto (c) show} repeat
100 {x y moveto (d) show} repeat
100 {x y moveto (e) show} repeat
100 {x y moveto (f) show} repeat
100 {x y moveto (g) show} repeat

```

A second example of randomly placing type introduces a few helpful variations of the random number procedure. Having the procedure

```
/x {rand exch mod} def
```

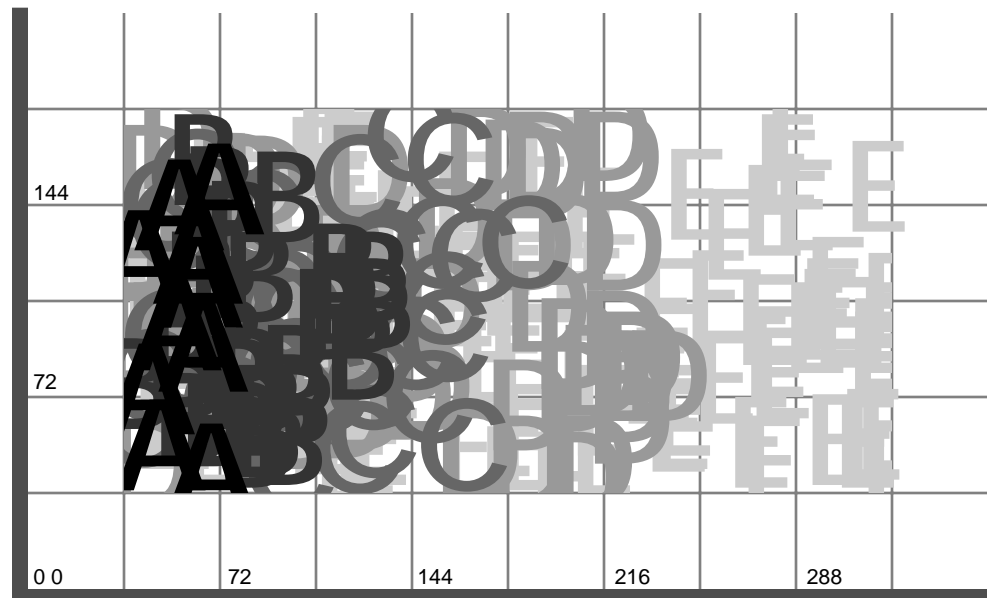
used in this way

```

100 x y moveto (A) show
200 x y moveto (B) show
300 x y moveto (C) show

```

can give flexibility in the program to produce variable ranges for the value of **x**. In the next example, this technique is used for the **x** value used by the **moveto** operator. The addition of the **12 sub** to the **/x** and **/y** procedures below shifts the values for **x** and **y** by **-12** so that the type will be cropped left and bottom.





16-9

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:randAlpha2.eps
%%BoundingBox:0 0 366 224

/x {rand exch mod 12 sub} def
/y {rand 130 mod 12 sub} def

/AvantGarde-Demi findfont 48 scalefont setfont

36 36 translate
0 0 moveto 288 0 rlineto 0 144 rlineto -288 0 rlineto
closepath clip newpath

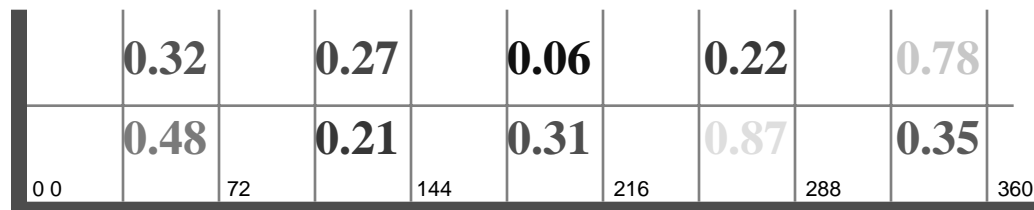
933478383 srand

.8 setgray
  100 {290 x y moveto (E) show} repeat
.6 setgray
  60 {200 x y moveto (D) show} repeat
.4 setgray
  35 {150 x y moveto (C) show} repeat
.2 setgray
  25 {100 x y moveto (B) show} repeat
0 setgray
  10 {40 x y moveto (A) show} repeat

```

16.5 *random gray values*

A randomly generated number can be used by the `setgray` operator as well. In this case, we need a number between 0 and 1. To get the number, a number between 0 and 100 is generated with `rand 100 mod` and then this number is multiplied by `.01` to shift the decimal point. The following example is similar to `seeRand2.eps` seen earlier in section 16.2. In this version, the generated number is duplicated so it can be used twice. The first is used by the `setgray` operator, the copy is converted to a string and painted. Therefore, the number appears in its gray value.



16-10

```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:seeRgray1.eps
%%DocumentFonts:Times-Bold
%%BoundingBox:36 18 360 70

/Times-Bold findfont 18 scalefont setfont
/str 10 string def

```



```

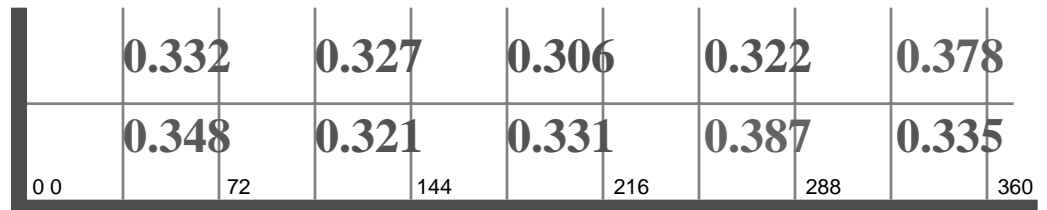
/rG {rand 100 mod .01 mul} def
6234572 srand

36 18 moveto rG dup setgray str cvs show
108 18 moveto rG dup setgray str cvs show
180 18 moveto rG dup setgray str cvs show
254 18 moveto rG dup setgray str cvs show
326 18 moveto rG dup setgray str cvs show

36 48 moveto rG dup setgray str cvs show
108 48 moveto rG dup setgray str cvs show
180 48 moveto rG dup setgray str cvs show
254 48 moveto rG dup setgray str cvs show
326 48 moveto rG dup setgray str cvs show

```

A variation of this procedure is to confine the gray values within an even tighter range by multiplying by `.001` instead of `.01` and then adding a number to fill the first decimal space. The number obtained from the random generation may be `0.027`, for example. By adding `0.3`, it will be `0.327`. See example below. By moving the decimal over three places instead of two, the number will always be `0.0xx`. Whatever number is then added determines the narrow range as long as it's between `0.1` and `0.9`. Or, add nothing if you want to keep it in the `0.001` to `0.099` range.



16-11

```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:seeRgray2.eps
%%DocumentFonts:Times-Bold
%%BoundingBox:36 18 370 70

/Times-Bold findfont 18 scalefont setfont
/str 10 string def
/rG {rand 100 mod .001 mul .3 add} def
6234572 srand

36 18 moveto rG dup setgray str cvs show
108 18 moveto rG dup setgray str cvs show
180 18 moveto rG dup setgray str cvs show
254 18 moveto rG dup setgray str cvs show
326 18 moveto rG dup setgray str cvs show

36 48 moveto rG dup setgray str cvs show
108 48 moveto rG dup setgray str cvs show
180 48 moveto rG dup setgray str cvs show
254 48 moveto rG dup setgray str cvs show
326 48 moveto rG dup setgray str cvs show

```



16-12

Another example is the listing for the design on the first page of this chapter.

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:quad_abcd.eps
%%BoundingBox:0 0 360 360

/Helvetica-Bold findfont 12 scalefont setfont

/n {rand exch mod} def
/nn {rand 200 mod} def
/g {rand 100 mod .01 mul} def
/box {0 0 moveto 180 0 lineto 180 180 lineto
      0 180 lineto closepath} def

173417 srand

0 180 translate box fill % upper left
gsave
  box clip newpath

  /a {180 n nn moveto (a) show} def
  /b {144 n nn moveto (a) show} def
  /c {108 n nn moveto (a) show} def
  /d {72 n nn moveto (a) show} def
  /e {36 n nn moveto (a) show} def

  25 {g setgray 25 {a} repeat} repeat
  20 {g setgray 25 {b} repeat} repeat
  15 {g setgray 25 {c} repeat} repeat
  10 {g setgray 25 {d} repeat} repeat
  10 {g setgray 25 {e} repeat} repeat
grestore

0 -180 translate box .5 setgray fill
gsave
  box clip newpath

  /a {nn 180 n moveto (b) show} def
  /b {nn 144 n moveto (b) show} def
  /c {nn 108 n moveto (b) show} def
  /d {nn 72 n moveto (b) show} def
  /e {nn 36 n moveto (b) show} def

  25 {g setgray 25 {a} repeat} repeat
  20 {g setgray 25 {b} repeat} repeat
  15 {g setgray 25 {c} repeat} repeat
  10 {g setgray 25 {d} repeat} repeat
  10 {g setgray 25 {e} repeat} repeat
grestore

180 180 translate box .2 setgray fill

```

```

gsave
  box clip newpath

  /a {nn 180 n moveto (c) show} def
  /b {nn 144 n moveto (c) show} def
  /c {nn 108 n moveto (c) show} def
  /d {nn 72 n moveto (c) show} def
  /e {nn 36 n moveto (c) show} def

  25 {g setgray 25 {a} repeat} repeat
  20 {g setgray 25 {b} repeat} repeat
  15 {g setgray 25 {c} repeat} repeat
  10 {g setgray 25 {d} repeat} repeat
  10 {g setgray 25 {e} repeat} repeat
grestore

0 -180 translate box .9 setgray fill
gsave
  box clip newpath

  /a {36 n nn moveto (d) show} def
  /b {72 n nn moveto (d) show} def
  /c {108 n nn moveto (d) show} def
  /d {144 n nn moveto (d) show} def
  /e {180 n nn moveto (d) show} def

  25 {g setgray 25 {a} repeat} repeat
  20 {g setgray 25 {b} repeat} repeat
  15 {g setgray 25 {c} repeat} repeat
  10 {g setgray 25 {d} repeat} repeat
  10 {g setgray 25 {e} repeat} repeat
grestore

```




some advanced programming ideas

There are a number of techniques that can be used to increase the performance of your PostScript programs. The techniques involve new ways of defining the procedures you write, redefining PostScript operators, and creating your own user dictionary of procedures. After this discussion, I'll explain several designs in detail.

17.1 *early binding*

PostScript is an extensible language, meaning **box**, as defined in this example,

```
/box { moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
      closepath } def
```

can be used like any other PostScript operator. You have expanded the vocabulary of the PostScript language. Procedures that you create are located in the **userdict** dictionary on a dictionary stack. PostScript operators are found in the **systemdict** dictionary below **userdict** on the stack. When a name or key is encountered, the **userdict** is looked into first, the **systemdict** second, going down the stack. Every time **box** is used, the PostScript interpreter looks first in the user dictionary for the meaning of **box**. What is found there is **moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto closepath**. Next, the **userdict**, then the **systemdict** dictionaries are searched for the value of **moveto**, **rlineto**, and **closepath**.

Things could be sped up if all this lookup activity could be simplified. This can be done with a process called early binding and it's done with the **bind** operator. All that looking up process described in the previous paragraph is called late binding. The **box** procedure using **bind** would be written like this:

```
/box { moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
      closepath } bind def
```

Now the lookup is performed when the procedure is first defined. When the key **box** is now encountered in a program, its value is executed without any lookup. In a short program, the performance difference is negligible. Most of the program examples in this book would not benefit from this. It is useful, however, in much larger programs that take more time to process.

If a procedure name is used within the definition of another procedure, **bind** is only needed in the second procedure. There is no added benefit in binding both procedures, nor is there a penalty. For example, the **bind** used in the **R_box** procedure that follows will also apply to the **box** procedure.

```

/box {0 0 moveto 8 0 rlineto 0 8 rlineto -8 0 rlineto
      closepath fill} def

/R_box {100 {10 0 translate box} repeat} bind def

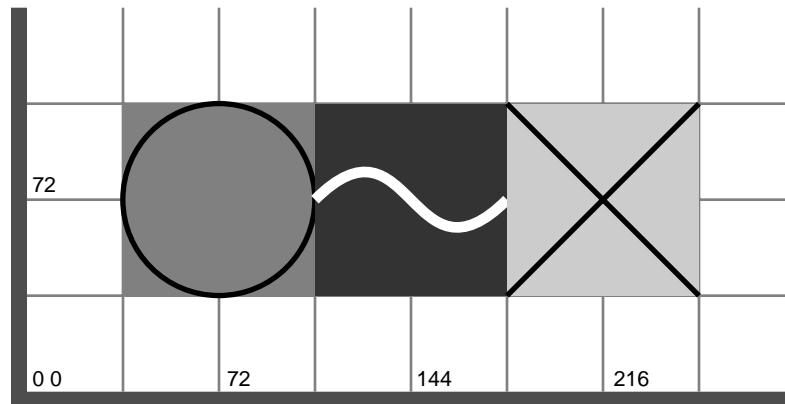
```

17.2

load

Another technique to increase the efficiency of your program is to reduce its size. Sometimes just the amount of information being sent to the printer is taking up all the time, not the PostScript interpreter processing what was received. Again, not much gain is going to be attained in a short program. However, if a PostScript program has a large number of `moveto`'s, `lineto`'s, and `stroke`'s for example, its size can be reduced significantly if they were replaced with `m`, `l`, and `s`. The operator `load` can be used to rename PostScript operators.

Actually, `lineto` is the key to the value of a PostScript operator that draws a line. The key to this operator value can be changed. `load` pushes the operator value onto the operand stack and `bind` can then be used to tightly associate a new key or name to the operator. Following is an example using both `bind` and `load`.



17-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:3boxDesigns.eps
%%BoundingBox:36 36 252 108

```

```

/bd {bind def} bind def
/lld {load def} bd
/lw /setlinewidth lld
/g /setgray lld
/m /moveto lld
/a /arc lld
/rl /rlineto lld
/l /lineto lld
/c /curveto lld
/cp /closepath lld
/s /stroke lld
/f /fill lld

```

```
% first box
.5 g 36 36 m 108 36 1 108 108 1 36 108 1 cp f
0 g 2 lw 72 72 36 0 360 a s

% second box
.2 g 4 lw 108 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
1 g 108 72 m 144 108 144 36 180 72 c s

% third box
.8 g 2 lw 180 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
0 g 180 36 m 72 72 r1 s 252 36 m -72 72 r1 s
```

17.3 *making a user dictionary*

The list of definitions used in the beginning of `3boxDesigns.eps` can be formed into a user dictionary. This is basically making your own laser printer prep file. The Macintosh Laser Prep file is a `userdict` dictionary named `md` (probably for `macintosh dictionary`). It needs to be in the LaserWriter when printing because Macintosh software uses the procedures defined in the `md` dictionary when it sends a file to the printer.

The `md` dictionary contains 250 procedure definitions. You can't open the Laser Prep file, but it can be examined in another way. Open any file and select Print from the "File" menu. Immediately after you click on OK in the Print dialog box, press command-k on the keyboard. A text file will be created on your disk drive of the `md` dictionary plus your file as it would be sent to the printer. A variation of this is pressing command-f. This will create the same text file minus the `md` dictionary. The files will be named `PostScript0`, `PostScript1`, and so on up to `PostScript9` and then start over at `PostScript0`.

This PostScript file, if sent to the laser printer, would reside in the printer's RAM just as the Laser Prep file does. Note that this is basically the same operation as downloading a font to the printer. See section 15.7.



17-2

```
%!PS-Adobe-2.0
%%Title:Sherm_dict.ps

serverdict begin 0 exitserver

/Sherm 13 dict def          % make dict with room for 13 def
Sherm begin                % put dict on top of dict stack

/bd   {bind def} bind def
/lld  {load def} bd
/lw   /setlinewidth ld
/g    /setgray ld
/m    /moveto ld
/a    /arc ld
/r1   /rlineto ld
/l    /lineto ld
/c    /curveto ld
/cp   /closepath ld
/s    /stroke ld
/f    /fill ld
```



17-3

```
end % remove as top dictionary
```

This version of `3boxDesigns.eps` could then be sent:

```
!PS-Adobe-2.0 EPSF-1.2
%%Title:withoutDict.eps
%%BoundingBox:36 36 252 108

Sherm begin % put dict on top of dict stack

% first box
.5 g 36 36 m 108 36 1 108 108 1 36 108 1 cp f

0 g 2 lw 72 72 36 0 360 a s

% second box
.2 g 4 lw 108 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
1 g 108 72 m 144 108 144 36 180 72 c s

% third box
.8 g 2 lw 180 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
0 g 180 36 m 72 72 r1 s 252 36 m -72 72 r1 s

end % remove from top of dict stack
```

However, that way of handling your dictionary can be inconvenient if you need to print your files at a number of different locations. To increase the portability of your files, your dictionary can be included in the file.



17-4

```
!PS-Adobe-2.0 EPSF-1.2
%%Title:withDict.eps
%%BoundingBox:36 36 252 108

/Sherm 13 dict def
Sherm begin

/bd {bind def} bind def
/ld {load def} ld
/lw /setlinewidth ld
/g /setgray ld
/m /moveto ld
/a /arc ld
/r1 /rlineto ld
/l /lineto ld
/c /curveto ld
/cp /closepath ld
/s /stroke ld
/f /fill ld

% first box
.5 g 36 36 m 108 36 1 108 108 1 36 108 1 cp f

0 g 2 lw 72 72 36 0 360 a s

% second box
```



```
.2 g 4 lw 108 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
1 g 108 72 m 144 108 144 36 180 72 c s

% third box
.8 g 2 lw 180 36 m 72 0 r1 0 72 r1 -72 0 r1 cp f
0 g 180 36 m 72 72 r1 s 252 36 m -72 72 r1 s

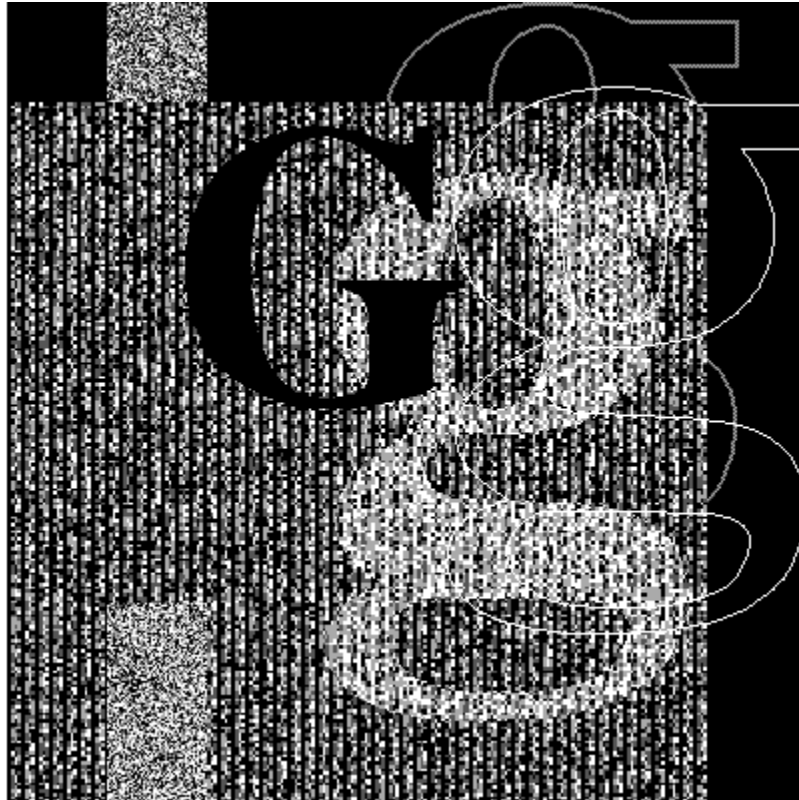
end
```

17.4 *3 designs explained, gDesign listing*

In this and the following two sections I'll explain three of my PostScript designs in more detail and present my approach to writing a program. I generally begin simply by deciding on a format and the germ of an idea. I also like working on a series, so one idea can feed into another. I'm always copying program fragments from other PostScript files to get things started. With this in mind, I generally try to write in such a way that this can easily be done.

This design can be found on the first page of chapter 7, "type basics."

We'll start with the standard EPS header info. The original design was intended to be larger. I began with `.5 .5 scale` to reduce the design by half. I also reflected the reduced size in the `%%BoundingBox:` comment. It used to be `0 0 800 800`.





17-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:gDesign.eps
%%Creator:John F Sherman
%%CreationDate:8 March 1989
%%DocumentFonts:Times-Bold
%%BoundingBox:0 0 400 400

.5 .5 scale          % reduced, originally designed for linotron
                    % remember to reflect change in BoundingBox

/Times-Bold findfont 800 scalefont setfont

% -----
This design uses the pattern procedure (see chapter 13) written as a 2-bit picture
pattern. As a 2-bit picture, the confinement of a value range is still there, but there
are also other values thrown in that give a sparkle effect. I also use an sx sy
scale within the pattern definition to determine the pattern's size. sx and sy
are defined and redefined four times before pattern is used throughout the
program.
% -----

/str 512 string def      % used by pattern proc

/pattern      % def for 200 200 random pattern
  {/light exch def /dark exch def
  /diff light dark sub def
  sx sy scale
  200 200 2 [200 0 0 200 0 0]
  {0 1 511 {str exch rand diff mod dark add put} for str}
  image } bind def

% background seen on top and right sides
0 0 moveto
800 0 lineto 800 800 lineto 0 800 lineto closepath fill

% 800 point Times-Bold g, outline with 50% gray 4 pt line
4 setlinewidth
.5 setgray
350 420 moveto (g) true charpath stroke

gsave          % large pattern
  /sx 700 def
  /sy 700 def
  0 50 pattern
grestore

gsave          % 1 by 2 near bottom
  /sx 100 def
  /sy 200 def
  100 0 translate
  25 50 pattern
grestore

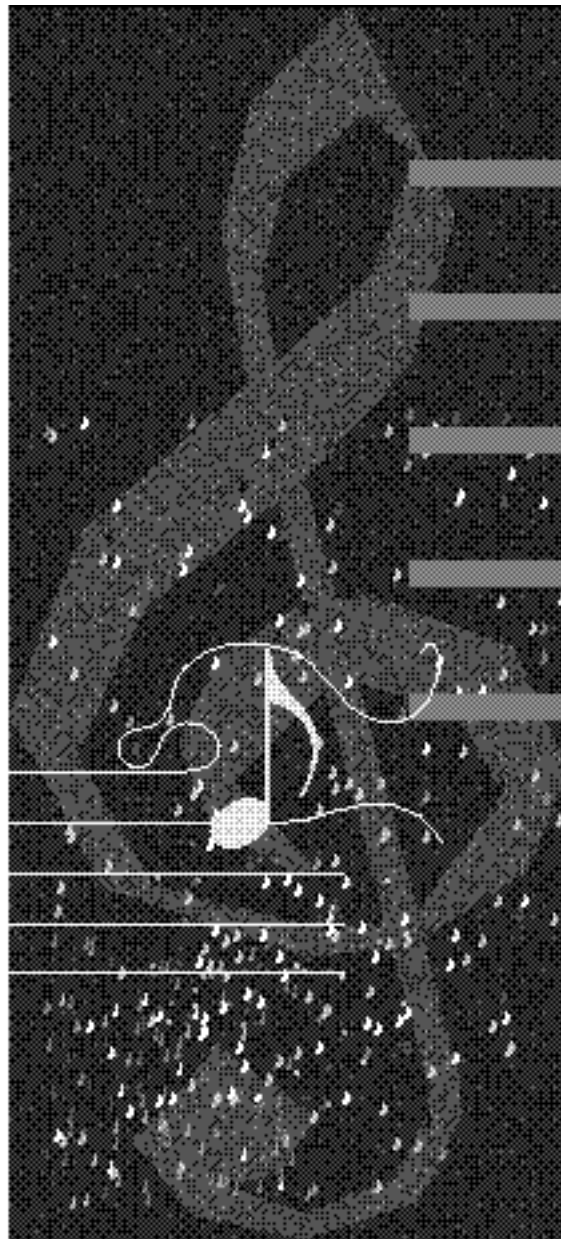
```

```
gsave          % 1 by 1 near top
  /sx 100 def
  /sy 100 def
  100 700 translate
  0 255 pattern
grestore

gsave          % pattern in g
  300 250 moveto (g) true charpath clip
  /sx 700 def
  /sy 700 def
  150 250 pattern % pattern for clip
grestore

1 setgray
2 setlinewidth
420 336 moveto (g) true charpath stroke

/Times-Bold findfont 400 scalefont setfont
0 setgray
160 400 moveto (G) show
```



17.5

sonataClef listing

Here's the standard EPS header info:

```
!PS-Adobe-2.0 EPSF-2.0
%%Title:sonataClef.eps
%%CreationDate:9 July 1988 rev 7 August 1989
%%Creator:John F Sherman
%%DocumentFonts:Sonata
%%BoundingBox:0 0 840 1850
```



17-6

```
% -----
```

The lines of code below set the page size for the Linotron. It is written in this way so that if it is printed on something other than a Linotron, such as a proof on an Apple LaserWriter, the file doesn't fail. The `statusdict` is a special dictionary that contains things such as the printer's default page size, the printer's name, whether or not the printer prints a test page, and a number of other persistent values.

`begin` makes `statusdict` the current dictionary. `/product load` returns or gets the value of `product` for the laser printer. In the case of an Apple LaserWriter, `/product load` returns `LaserWriter`; with a Linotron L300, it returns `Linotype`. Therefore, depending on which printer you're using, this line would read `LaserWriter Linotype eq` or `read Linotype Linotype eq`. `eq` looks to see if the top two items on the stack are equal and returns either a `true` or `false` (known as a boolean). On the LaserWriter we'll get a `false`, on the L300 we'll get a `true`.

In the next line, `{840 1850 12 1 setpageparams} if`, between the `{ }` is a procedure that sets the page size to 840 wide, 1850 high, and an offset of 12 points from the edge as a vertical page (0 is landscape). `if` is a control operator that will execute the page size procedure if the boolean from the previous line is `true`. If it's `false`, the page size doesn't change.

In short, if this file is being printed on a Linotron, use these page parameters. If not, disregard them.

```
% -----

statusdict
begin
    /product load (Linotype) eq
    {840 1850 12 1 setpageparams} if
end
```

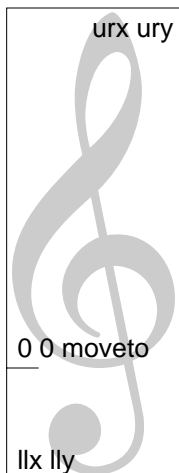
```
% -----
```

This next section of the program is a user dictionary named `Sherms` containing operator redefinitions and procedures. The purpose of `Sherms` is to hopefully gain some performance, but mostly it is a typing convenience. It is easier to type `m` instead of `moveto`, `l` for `lineto`, and so on. If the files were very large, performance would be more easily seen. See section 17.3.

```
% -----

/Sherms 35 dict def
Sherms begin

/bd    {bind def} bind def
/ld    {load def} bd
/m     /moveto ld
/c     /curveto ld
/r1    /rlineto ld
/l     /lineto ld
/s     /show ld
/S     /stroke ld
/r     /repeat ld
```



```

% -----
The purpose of the following lines of code is to obtain four numbers. The four
numbers are the bounding box of the clef character of the Sonata font. The
bounding box is the x y locations of the lower left and upper right corners. I need
these numbers so I can automatically create a pattern the same size as the clef
character's bounding box. Once I get the four numbers, I'll need to adjust them
because the lower left corner will have a negative y value. Note where the 0 0
moveto is located in this drawing. I determined the size of the clef by trial and
error using seeBBox1.ps found in section D.5.
% -----

/Sonata findfont [456.87 0 0 355 0 0] makefont setfont
0 0 m
    (&) true charpath % get char path
    flattenpath      % reduces curves to straight lines
    pathbbox        % get BBox numbers
    /ury exch def /urx exch def
    /lly exch def /llx exch def
newpath              % clear path

/str 512 string def
/y   lly neg ury add cvi def
/x   urx cvi def

/pattern {
    /dark exch def
    /lite exch def
    /dif dark lite sub def
    x y scale          % x and y from above, size of pattern
    x y 8 [ x 0 0 y 0 0 ]
    {0 1 511 {str exch rand dif mod lite add put} for str}
    image} bd

/n   {rand exch mod cvr} bd
/gray {100 n .01 mul .1 add} bd

/r1  {800 n 1200 n m (e) s} bd      % random notes
/r2  {400 n 400 n m (e) s} bd
/r3  {400 n 400 n m (h) s} bd
/r4  {400 n 400 n m (w) s} bd

% -----

newpath      % frame
    0 setgray
    0 0 m 0 1850 1 840 1850 1 840 0 1
    closepath
    6 setlinewidth S

222 srand

gsave      % pattern behind everything
    3 3 scale
    25 50 pattern

```

```

grestore

gsave
    20 setflat          % to prevent limitcheck error
    3 3 scale
    llx lly abs m      % place clef
    (&) true charpath clip
    50 100 pattern     % pattern in clip clef
grestore

% random notes -----

/Sonata findfont 18 scalefont setfont
500000000 srand

gsave
    50 50 translate
    20 {gray setgray 10 {r1} r } r
    10 {gray setgray 10 {r3} r } r
grestore

gsave
    200 200 translate
    5 {gray setgray 10 {r2} r } r
    5 {gray setgray 10 {r4} r } r
grestore

% staff -----

40 setlinewidth .5 setgray
600 1600 m 250 0 r1 S
600 1400 m 250 0 r1 S
600 1200 m 250 0 r1 S
600 1000 m 250 0 r1 S
600 800 m 250 0 r1 S

% curve -----

2 setlinewidth 1 setgray

0 700 m
250 700 l
280 700 318.4 710.1 316 740 c
313.6 769.5 271.6 784.6 246 770 c
218 754 228 708 194 708 c
175.1 708 158 728 166 748 c
177.1 775.9 206.8 758.9 226 776 c
252.2 799.4 239.4 833 264 858 c
305.7 900.3 357.2 903.2 414 886 c
480 866 484 798 554 778 c
596.4 765.9 632 792 644 848 c
654.7 898.2 626 902 616 880 c
S
0 625 m

```

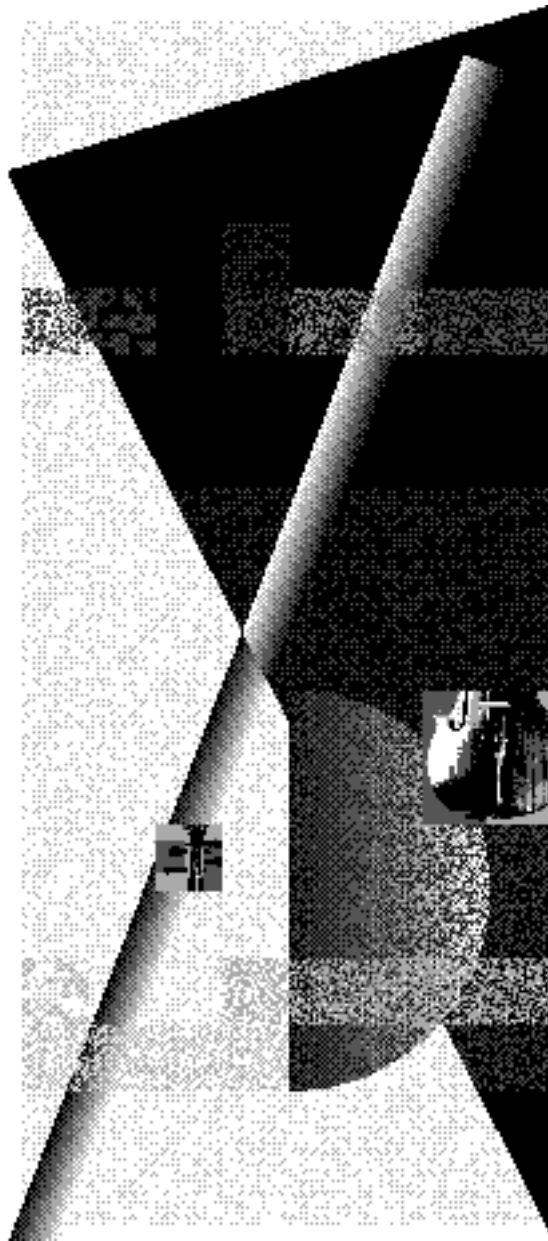
```
400 625 l
470.2 625 514 666 578 652 c
611.9 644.6 627 633 650 598 c
S
0 550 m 500 0 r1 S
0 475 m 500 0 r1 S
0 400 m 500 0 r1 S
S

% large note -----

/Sonata findfont 300 scalefont setfont
300 625 m .9 setgray
(e) s

end % for sherms dict

showpage
```

17.6 *symphony, opus 1 listing*

This design is the first panel of a triptych entitled *Symphony, opus 1*. It begins with the definition of radBit-Roman, a font explained in more detail in 15.6. This design includes two small scanned images. These scanned images were created using Icon on the NeXT for want of anything else at the time. If you're using the Mac, I'd use Adobe PhotoShop to make your pictures. Also, I would get a good text editor (I use QUED) if you now use MSWord or MacWrite to write your text files. Files with scanned images tend to be big; MSWord and MacWrite don't handle large files very well. The size of your file may also be limited by the amount of RAM you have in your computer.



17-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:symphony, opus 1.1
%%Creator:John F Sherman   University of Notre Dame
%%CreationDate: copyright Jan 1990
%%DocumentFonts:Sonata
%%BoundingBox:0 0 840 1850

statusdict % For Linotron printing
begin
    /product load (Linotype) eq
    {840 1850 12 1 setpageparams} if
end

/newfont 10 dict def
newfont begin

/FontType 3 def
/FontMatrix [.001 0 0 .001 0 0] def
/FontBBox [0 0 1000 1000] def

/Encoding 256 array def
StandardEncoding Encoding copy pop

/CharProcs 30 dict def
CharProcs begin

    /.notdef{ } def
    /str 512 string def
    /rBit {
        0 1 511{CharProcs
            /str get exec exch rand 255 mod put} for CharProcs
        /str get exec } def

    /space{0 0 moveto newpath} bind def

    /zero {48 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /one {49 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /two {50 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /three{51 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

    /four {52 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

```

```

/five {53 srand
      100 100 true [.1 0 0 .1 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/six {54 srand
     100 100 true [.1 0 0 .1 0 0]
     {CharProcs /rBit get exec}imagemask} bind def

/seven{55 srand
      100 100 true [.1 0 0 .1 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/eight{56 srand
      100 100 true [.1 0 0 .1 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/nine {57 srand
      100 100 true [.1 0 0 .1 0 0]
      {CharProcs /rBit get exec}imagemask} bind def

/A {65 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/B {66 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/C {67 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/D {68 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/E {69 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/F {70 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/a {97 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/b {98 srand
   100 100 true [.1 0 0 .1 0 0]
   {CharProcs /rBit get exec}imagemask} bind def

/c {99 srand

```

```

        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

/d      {100 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

/e      {101 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def

/f      {102 srand
        100 100 true [.1 0 0 .1 0 0]
        {CharProcs /rBit get exec}imagemask} bind def
end

/BuildChar
  {1000 0 0 0 1000 1000 setcachedevice
  exch begin
  Encoding exch get
  CharProcs exch get
  exec end} def
end

/radBit-Roman newfont definefont pop

% PROCEDURES -----
% These procedures are defined to simplify typing.

/bd     {bind def} bind def
/ld     {load def} bd
/m      /moveto ld
/sg     /setgray ld
/s      /show ld
/fStr 256 string def

% BEGIN DESIGN ONE -----
Since radBit-Roman is a mono-spaced font, I can set up a grid pattern based on its
point size of 100. A character and space are each 100 points square. Notice my m or
moveto commands are in 100 point intervals up the page. Later on you'll see
spaces used to break up the pattern.
% -----

/radBit-Roman findfont 100 scalefont setfont

gsave
  20 25 translate
  .9 sg
  0 0 m (12312312) s
  0 100 m (12312312) s
  0 200 m (12312312) s
  0 300 m (12312312) s
  0 400 m (12312312) s
  0 500 m (12312312) s

```

```

0 600 m (12312312) s
0 700 m (12312312) s
0 800 m (12312312) s
0 900 m (12312312) s
0 1000 m (12312312) s
0 1100 m (12312312) s
0 1200 m (12312312) s
0 1300 m (12312312) s
0 1400 m (12312312) s
0 1500 m (12312312) s
0 1600 m (12312312) s
0 1700 m (12312312) s
grestore

gsave      % fountain
-21 rotate
255 -1 0 { fStr exch dup put } for
70 1900 scale
256 1 8 [ 256 0 0 1 0 0 ] { fStr } image
grestore

% -----
This triangle shape was the product of an accident. I meant to make a rectangle. I
entered the coordinates wrong, but liked the result. The triangle shape is filled
black and is used as a clipping path for the reversed fountain and radBit-Roman
characters.
% -----

gsave      % large triangle
0 sg
0 1600 moveto 820 1850 lineto 820 0 lineto
closepath clip fill

.1 sg
20 725 m ( cdabcd) s
20 825 m ( cdabcd) s
20 925 m ( cdabcd) s
20 1025 m ( cdabcd) s

gsave
{1 exch sub} settransfer      % reverses fountain
-21 rotate
255 -1 0 { fStr exch dup put } for
70 1900 scale
256 1 8 [ 256 0 0 1 0 0 ] { fStr } image
grestore

.2 sg
20 225 m ( bca) s
20 325 m ( bca) s
20 425 m ( bca) s
20 525 m ( bca) s
grestore

```

```

% -----
One of the characteristics of radBit-Roman is that you can see through it. This is
because the imagemask operator is used instead of image. This next section
draws the half circle that clips the fountain with the radBit-Roman characters over
it.
% -----

gsave
  420 225 translate

  0 600 m 0 300 300 270 90 arc
  closepath clip newpath

  gsave
  255 -1 0 { fStr exch dup put } for
  300 600 scale
  256 1 8 [ 256 0 0 1 0 0 ]
  { fStr } image
  grestore

  .2 sg
  0 0 m (abc) s
  0 100 m (abc) s
  0 200 m (abc) s
  0 300 m (abc) s
  0 400 m (abc) s
  0 500 m (abc) s
grestore

20 225 m% 3
.8 sg(aaaa) s

20 325 m% 4
(A )s .7 sg (abcde) s

20 1325 m
.2 sg (aB a) s .5 sg (1b) s .3 sg (1b) s

320 1425 m
.1 sg (1) s

0 sg

% vio3

/picstr 25 string def
gsave
220 525 translate
100 100 scale
100 100 2
[100 0 0 100 neg 0 100]
{currentfile picstr readhexstring pop}
image
AAAAAAAAAAAAAAAAAAAAAAAAA9020020056401690255AAAAAAAAAAAAAAAAAAAA

```

```

AAAAAAAA90200500194015501846AAAAAAAAAAAAAAAAAAAAAAAAAAAA4
06

```

5K of scanned data

```

AAAAAAAA9A00340080081555555555555555AAAAAAAAAAAAAAAAAAAA9
400800815555555555555555AAAAAAAAAAAAAAAAAAAA9A003400800815
5grestore

```

% viol

```

/picstr 50 string def
gsave
620 625 translate
200 200 scale
200 200 2
[200 0 0 200 neg 0 200]
{currentfile picstr readhexstring pop}
image
555555555555555555555555555555558BFBBAA00FBBA4002D55101C000003000
00000000004000000000602AAAAAAAAAAAAA55555555555555555555
55

```

20K of scanned data

```

55555555AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A555555555555555555555555555555555555555555555555555555555555555
AAgrestore

```

showpage



PostScript Level 2

The PostScript language has evolved from its first introduction in 1985. When introduced, it was primarily conceived as a page description language for black and white laser printers. Since then, extensions have been made for color printers, computer displays and new technologies and features available for individual laser printers. PostScript Level 2 brings all these extensions and new image opportunities into one unified implementation.

This chapter will concentrate on some of the PostScript Level 2 operators that were originally created as extensions for Display PostScript and color printers. PostScript Level 2 is so new, not all printers will be equipped with it, so you may not be able to experiment with some of the operators presented in this chapter. If you have access to a NeXT, all the examples in this chapter can be tried and experimented with. They will work on both the black and white and color NeXT computers.

If you are a Macintosh user, you will either need a Level 2 laser printer to use all these examples or a color printer for the color examples. If you only have an Apple LaserWriter, some of the color examples will work, some will not. For example, `setrgbcolor` will work on an Apple LaserWriter, `setcmykcolor` will not. Some of the Display PostScript operators, such as `rectfill` and `rectstroke`, can be simulated with Level 1 operators.

18.1

rectfill

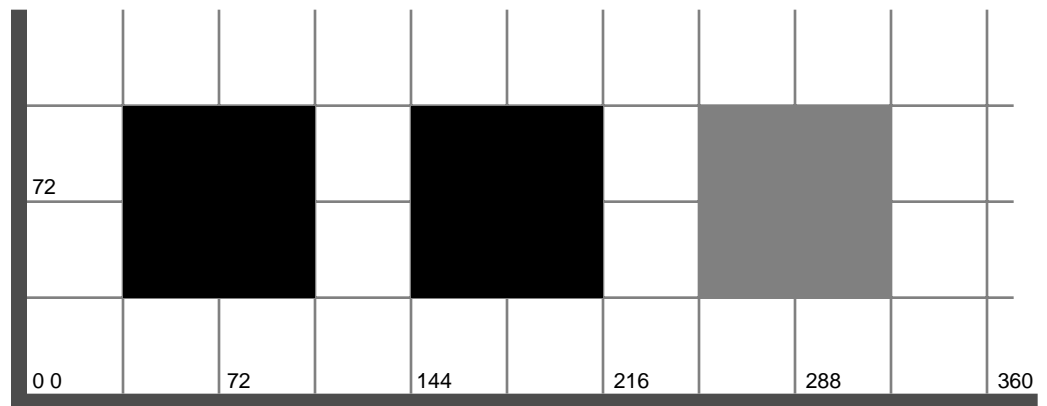
In PostScript Level 2, there are two new operators for drawing squares or rectangles. They are `rectfill` and `rectstroke`. They originate from the Display PostScript extensions.

`rectfill` is the combination of `moveto`, `rlineto`, and `fill`. Its syntax is:

```
x y width height rectfill
```

where `x y` is the location of the lower left corner of the rectangle on the page relative to the origin and `width height` is the distance from the `x y` to the upper right corner.

`x y` can be thought of as the equivalent of the current point made by a `moveto` when using an `rlineto`. Unlike `rlineto`, `rectfill` does not need the establishment of a current point. In the next example, three squares are drawn and painted. The first is drawn in the conventional way, the second and third are drawn using `rectfill`.



18-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rectfill_1.eps
%%BoundingBox:36 36 324 108

36 36 moveto          % first square
72 0 rlineto 0 72 rlineto -72 0 rlineto closepath fill

144 36 72 72 rectfill % second square

252 36 translate     % third square
.5 setgray
0 0 72 72 rectfill

```

As explained earlier, many laser printers will not be able to understand `rectfill`. If you are using such a printer, `rectfill_1.eps` can be rewritten to work on a Level 1 laser printer.



18-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rectfill_2.eps
%%BoundingBox:36 36 324 108

% simulate rectfill on a Level 1 laser printer

/rectfill      {
  /ury exch def /urx exch def
  /lly exch def /llx exch def
  llx lly moveto urx 0 rlineto 0 ury rlineto
  urx neg 0 rlineto closepath fill } def

36 36 moveto          % first square
72 0 rlineto 0 72 rlineto -72 0 rlineto closepath fill

144 36 72 72 rectfill % second square

252 36 translate     % third square
.5 setgray
0 0 72 72 rectfill

```

18.2

rectstroke

`rectstroke` is the combination of `moveto`, `rlineto`, and `stroke`. Its syntax is:

```
x y width height rectstroke
```

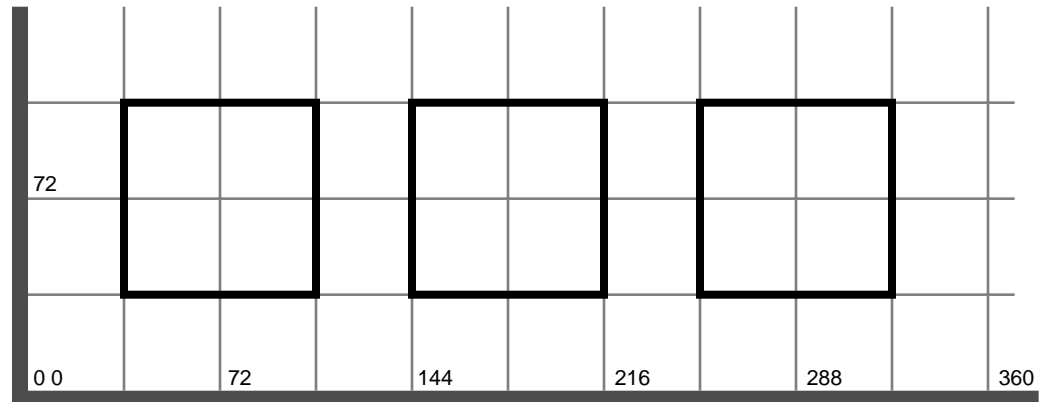
where `x y` is the location of the lower left corner of the rectangle on the page relative to the origin and `width height` is the distance from the `x y` to the upper right corner.

Unlike `rectfill`, an optional `matrix` can also be supplied.

```
x y width height matrix rectstroke
```

`matrix` will perform a transformation on the rectangle's line weight, but not on the path itself. This is because the transformation occurs after the construction of the path.

`rectstroke` is much like `rectfill`, as can be seen in the next example. This example is basically the same program as `rectfill_1.eps`.



18-3

```
!PS-Adobe-2.0 EPSF-1.2  
%%Title:rectstroke_1.eps  
%%BoundingBox:34 34 326 110
```

```
% first square  
3 setlinewidth  
36 36 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto  
closepath stroke
```

```
% second square  
144 36 72 72 rectstroke
```

```
% third square  
252 36 translate  
0 0 72 72 rectstroke
```

The following example is a rewrite of `rectstroke_1.eps` for a Level 1 laser printer. It is essentially a simple variation of `rectfill_2.eps` from the previous page. However, it is not a complete substitute for `rectstroke`. This simulation will not work for the later examples of `rectstroke` that include a `matrix`. A



18-4

more detailed explanation of the considerations involved in emulating Level 2 operators can be found in “Appendix D: Compatibility Strategies,” in the *PostScript Language Reference Manual*, second edition.

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rectstroke_2.eps
%%BoundingBox:34 34 326 110

% simulate rectstroke on a Level 1 laser printer

/rectstroke {
  /ury exch def /urx exch def
  /lly exch def /llx exch def
  llx lly moveto urx 0 rlineto 0 ury rlineto
  urx neg 0 rlineto closepath stroke } def

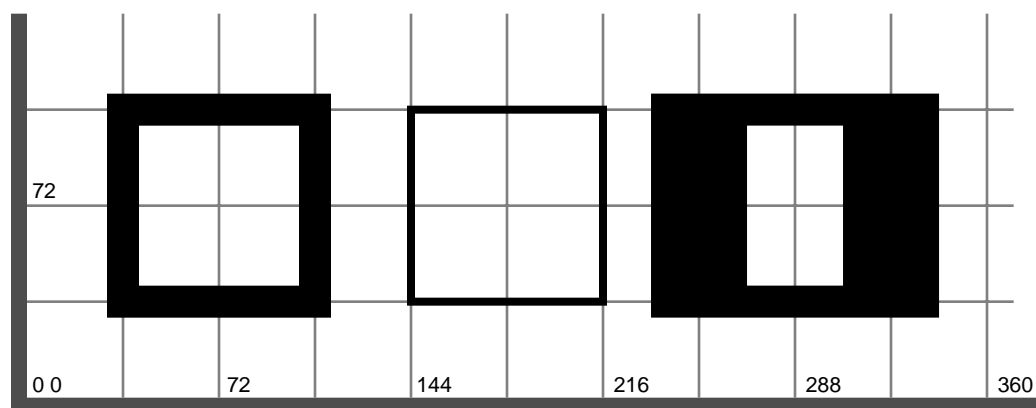
% first square
3 setlinewidth
36 36 moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
closepath stroke

% second square
144 36 72 72 rectstroke

% third square
252 36 translate
0 0 72 72 rectstroke

```

As outlined in the beginning of this section, a **matrix** can be included in the **rectstroke** arguments. The transformation affects the line weight of the rectangle. Following are some examples of how this variation of **rectstroke** can be used. Note that the transformations do not change the graphic state.



18-5

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rectstroke_3.eps
%%BoundingBox:30 30 342 114

3 setlinewidth

```

```

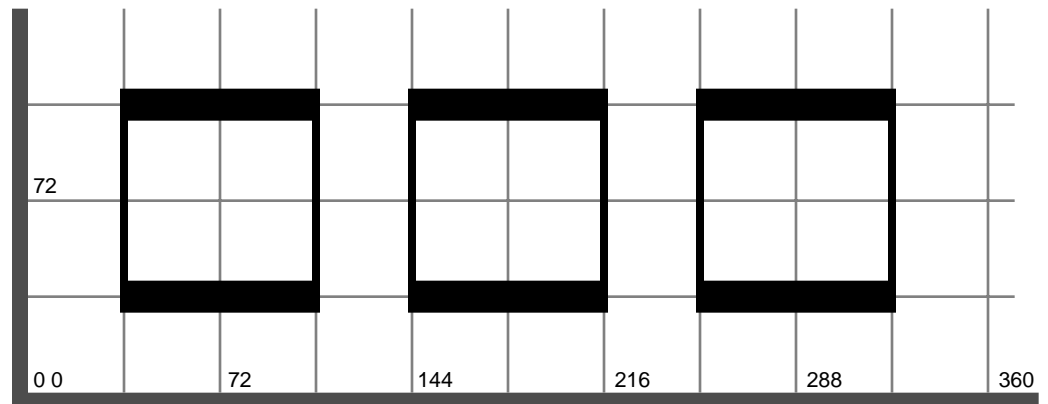
% first square
36 36 72 72 [4 0 0 4 0 0] rectstroke

% second square
144 36 72 72 [1 0 0 1 0 0] rectstroke

% third square
252 36 72 72 [12 0 0 4 0 0] rectstroke

```

The result of using a `matrix` with `rectstroke` can be simulated with Level 1 operators. In this example, the second and third rectangles accomplish the same result as with `rectstroke` used for the first.



18-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rectstroke_4.eps
%%BoundingBox:34 30 326 114

3 setlinewidth

% first square
36 36 72 72 [1 0 0 4 0 0] rectstroke

gsave % second square
  144 36 moveto
  72 0 rlineto 0 72 rlineto -72 0 rlineto closepath
  1 4 scale
  stroke
grestore

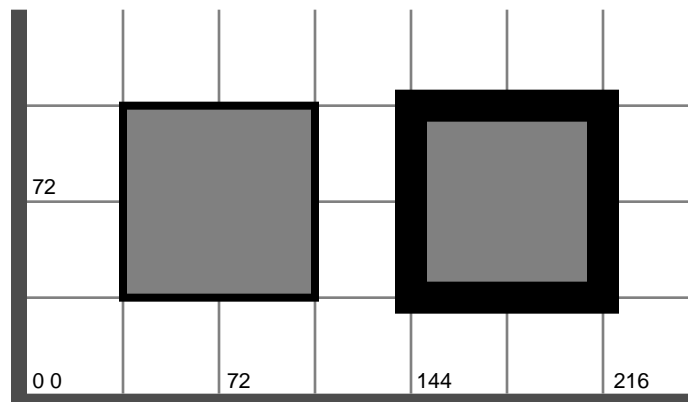
gsave % third square
  252 36 moveto
  72 0 rlineto 0 72 rlineto -72 0 rlineto closepath
  [1 0 0 4 0 0] concat
  stroke
grestore

```

The coordinates of the rectangles made by `rectfill` and `rectstroke` can be supplied as an array. These two lines are the same:

```
36 36 72 72 rectfill
and
[36 36 72 72] rectfill
```

The array can be given a name so that the same dimensions of a rectangle can be used several times.



18-7

```
!PS-Adobe-2.0 EPSF-1.2
%%Title:bothRect_1.eps
%%BoundingBox:34 30 222 114

/size [0 0 72 72] def
/m [4 0 0 4 0 0] def

3 setlinewidth

% first square
gsave
    36 36 translate
    .5 setgray
    size rectfill
    0 setgray
    size rectstroke
grestore

% second square
gsave
    144 36 translate
    .5 setgray
    size rectfill
    0 setgray
    size m rectstroke
grestore
```

18.3 *new type operators*

There are several new font and character operators in PostScript Level 2 that also

originate from the Display PostScript extensions. Some are for convenience, others provide new typographic control. The new operators are:

```
selectfont
xshow
yshow
xyshow
```

`selectfont` is considered a convenience operator in that it combines the functions of the `findfont`, `scalefont`, and `setfont` or the `findfont`, `makefont`, and `setfont` operators. Instead of writing,

```
/Times-Bold findfont 42 scalefont setfont
```

this can be used:

```
/Times-Bold 42 selectfont
```

Instead of writing,

```
/Times-Bold findfont [42 0 0 24 0 0] makefont setfont
```

this can be used:

```
/Times-Bold 42 [42 0 0 24 0 0] selectfont
```

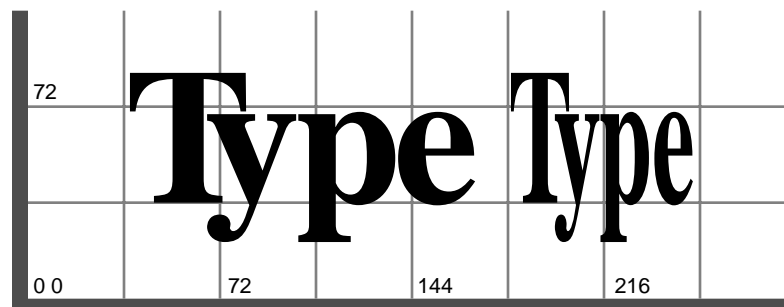
Examples of the uses of `selectfont` can be found in the type examples that follow.

`xshow`, `yshow`, and `xyshow` are new operators that provide new strategies for custom letterspacing. `xshow` permits the individual spacing of characters in a string along the x axis. `yshow` individually spaces characters along the y axis and `xyshow` is the combination of `xshow` and `yshow`. Their syntax are:

```
string spacingArray xshow
string spacingArray yshow
string spacingArray xyshow
```

where `spacingArray` is an array of numbers that is the spacing for each character.

In the following example, the first array, [30 34 37 0], is the spacing values for *T*, *y*, *p*, and *e* respectively. If the width of *T* is changed from 30 to 34, the *ype* would shift to the right 4 points. The *e* was given a width of 0 only because nothing is following. If no spacing information is provided for the last character, an error will occur.





18-8

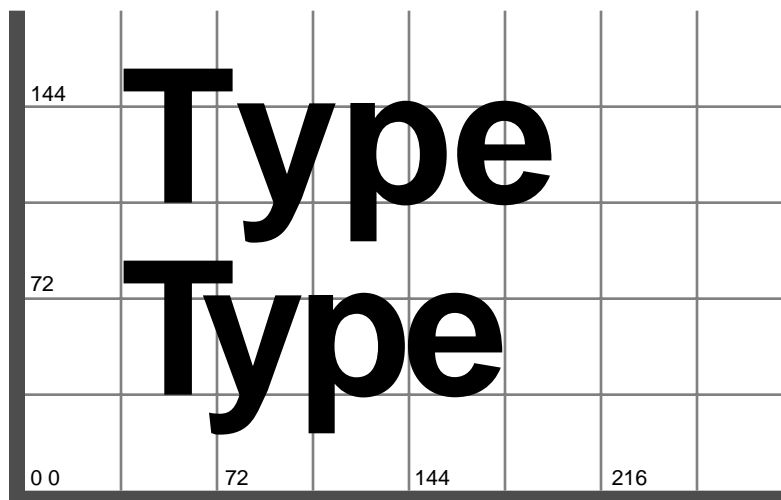
```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:xshow&.eps
%%BoundingBox:36 18 252 85

/Times-Bold 72 selectfont
36 36 moveto (Type) [30 34 37 0] xshow

/Times-Bold [36 0 0 72 0 0] selectfont
180 36 moveto (Type) [17 17 20 0] xshow
    
```

The following are examples of `xyshow`. In this case, the array is a pair of numbers for each character's x and y spacing within the string.



18-9

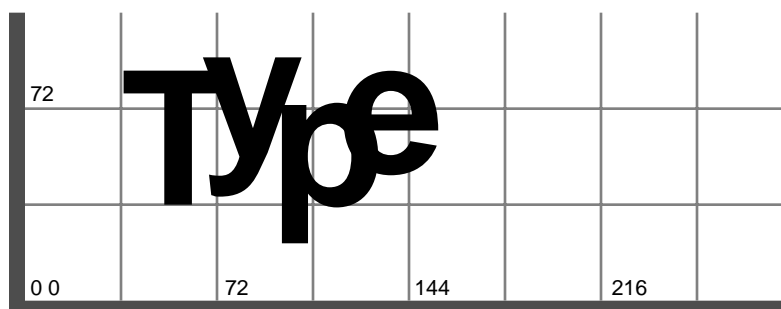
```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_xy1.eps
%%BoundingBox:36 18 198 164

/Helvetica-Bold 70 selectfont

36 108 moveto (Type) show      % Level 1

36 36 moveto (Type)
[30 0 36 0 40 0 0 0] xyshow   % Level 2
    
```





```

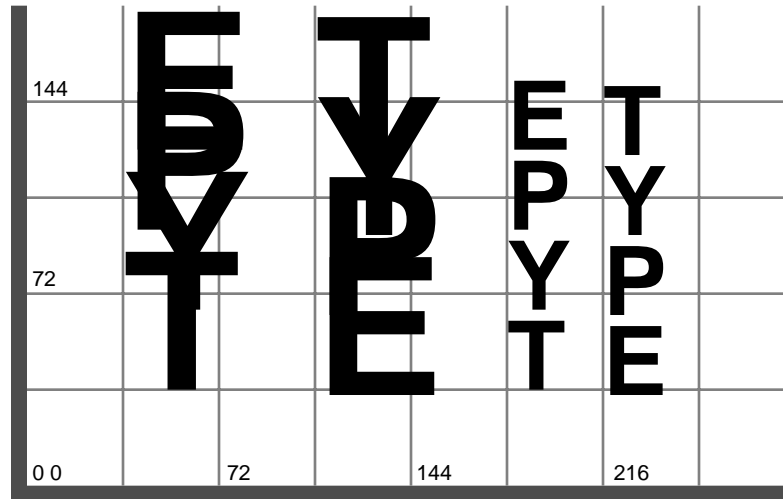
%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_xy2.eps
%%BoundingBox:36 12 156 90

/Helvetica-Bold 70 selectfont

36 36 moveto (Type)
[30 12 26 -18 26 12 0 0] xshow

```

Following is an example using `yshow`.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_y1.eps
%%BoundingBox:0 0 294 186

/Helvetica-Bold 72 selectfont

36 36 moveto (TYPE)
[30 30 30 30] yshow

108 124 moveto (TYPE)
[-30 -30 -30 -30] yshow

/Helvetica-Bold 36 selectfont

180 36 moveto (TYPE)
[30 30 30 30] yshow

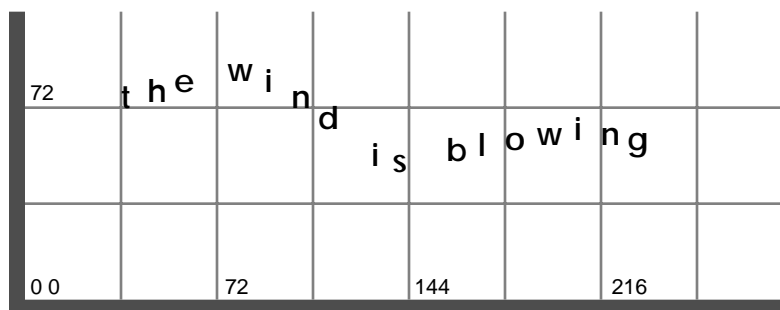
216 124 moveto (TYPE)
[-30 -30 -30 -30] yshow

```

Next is an example of using `xshow`. Note that the array is broken over four lines, each line for a word in the string.



18-12



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:windy.eps
%%BoundingBox:0 0 294 114

/AvantGarde-Demi 12 selectfont

36 72 moveto (the wind is blowing)
[10 2 10 4 10 6 10 -2
14 -4 10 -6 10 -8 10 0 10 -14
8 -2 10 2 10 2
12 2 10 2 12 2 14 2 10 -2 10 -2 0 0] xyshow
    
```

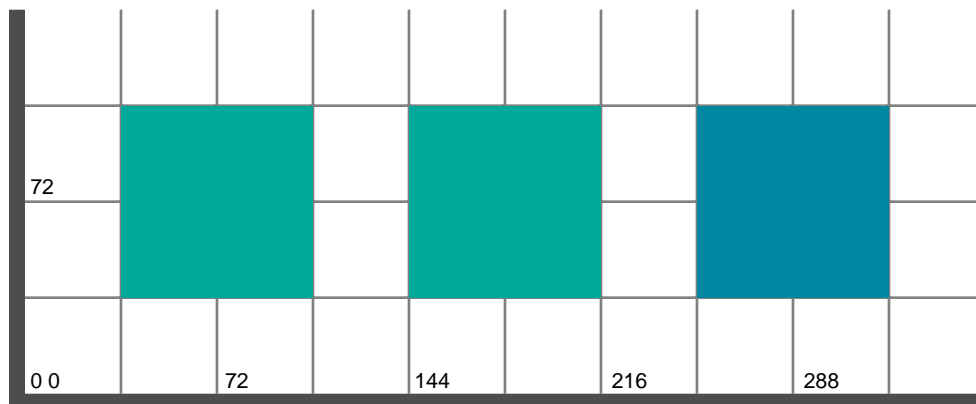
18.4 color operators

There are three operators used to specify a color. They are:

- hue saturation brightness sethsbcolor Level 1
- red green blue setrgbcolor Level 1
- cyan magenta yellow black setcmykcolor Level 2

Where in each case, the arguments are numbers between 0 and 1.

Each operator is based on a different method of identifying a color. The first two above are based on mixing light, the third is based on mixing inks. In the following example, the same color is made using each operator.





18-13

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:sameColor.eps
%%BoundingBox:0 0 366 150

/square      {moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
              closepath fill} def

.02 .66 .61 setrgbcolor
36 36 square

.486979 .969697 .66 sethsbcolor
144 36 square

.95 .31 .36 .03 setcmykcolor
252 36 square

```

`setrgbcolor` and `sethsbcolor` are both based on additive color mixing using light. Red, green, and blue light when mixed equal white. White on a TV or white clouds in the sky are the result of all colors of light mixed together. Figure 18-1 shows an additive color wheel.

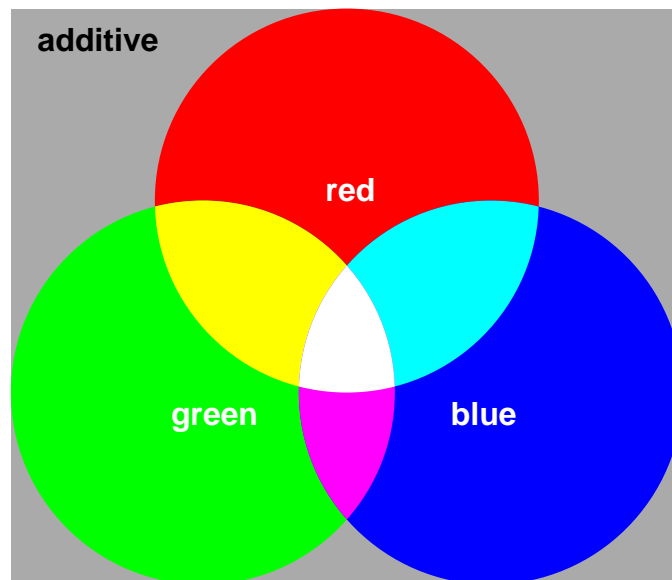


figure 18-1

`setcmykcolor` is based on subtractive color mixing using paint. Magenta, cyan, and yellow pigment when mixed equal black, at least in theory. Black is usually included in the mix because a good total black is hard to get without help. Figure 18-2 shows a subtractive color wheel.



color demo

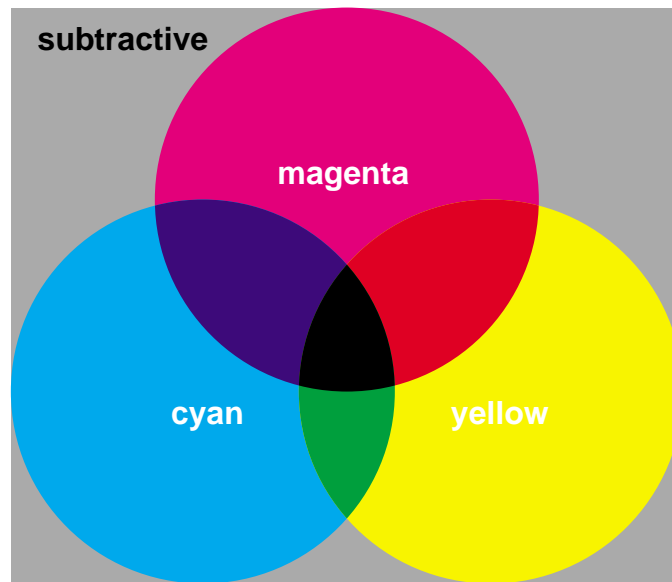


figure 18-2

Of the two methods, subtractive color mixing is the more intuitive way of specifying a color. It makes sense that mixing cyan and yellow paint will give green. It can be difficult, though, to visualize that red and green light will make yellow.

18.5 *color pictures*

Color pictures are made by the `colorimage` operator. Its syntax is:

```
width height bits matrix ds0...dsn s/mds comp colorimage
```

where:

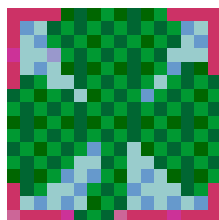
`width height bits matrix` are the same as with `image`.

`ds0...dsn` are one or more data sources. All the color components may be mixed together into one source or separated, for example, into individual red, green, and blue components.

`s/mds` is a boolean whether the data is from a single source (false) or multiple sources (true).

`comp` is a number representing the number of data sources.

The following example is a simplified version of a file made by Adobe PhotoShop.





18-14

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:cShamrock.eps
%%BoundingBox:0 0 160 160

/picstr1 16 string def
/picstr2 16 string def
/picstr3 16 string def
/picstr4 16 string def

/readdata {currentfile exch readhexstring pop} def

/shamrock{16 16 8 [.2 0 0 .2 0 0]
          {picstr1 readdata}
          {picstr2 readdata}
          {picstr3 readdata picstr4 readdata pop}
          true 3 colorimage} def

gsave
shamrock
CCCCCCCC000000CCCCCCCCCCCCCCCC666333333366996666633333333333
... color data ...

666666660033003300330033666666666666666663C423C603C423C60666
6grestore

```

18.6 *overview of new operators in Level 2*

PostScript Level 2 offers a number of new features and visual opportunities. It will take awhile for all of these new operators to make their way into printers and computer displays. Some things to look forward to are:

Patterns

PostScript Level 2 will have new operators to create patterns. These patterns can then be used as the paint for fills and strokes. Once a pattern `patternName` is created, it can be used by writing `patternName setpattern` within a program.

Forms

Forms can be used to create the repeating graphics on a multiple page document or as the name suggests, on a form to be filled out. Once the form `formName` is defined, it can be used by writing `formName execform` within a program.

Scanned pictures

Scanned pictures will be handled more efficiently and faster. Scanned pictures will be stored in a compressed form and be painted faster. Also, 12-bit per pixel black and white or 36-bit color pictures will be possible. A 12-bit picture has 4096 possible values.

Color

There will be expanded support to insure requested colors will remain consistent from different scanners, to different monitors, to various kinds of print output.



```

%!PS-Adobe-2.0 EPSF-2.0
%%Title:grass.eps
%%BoundingBox:0 0 300 300

/a {rand 50 mod} def      % number between 0 & 50
/b {rand 100 mod} def    % number between 0 & 100
/c {rand 150 mod} def    % & so on
/d {rand 200 mod} def    % see chap 16
/e {rand 250 mod} def
/f {rand 300 mod} def

17173434 srand           % seed random # to set sequence
.8 setgray               % 20% gray
0 0 300 300 rectfill     % rectangle, Level 2 operator

0 setgray                % set color to black

% draw 34 curved lines
% each different because of rand number

34 {0 0 moveto a b c d e f curveto stroke} repeat

```



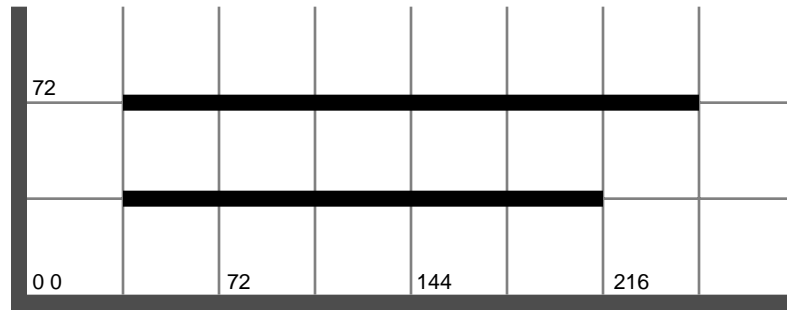
library of examples

The purpose of this chapter is to provide additional programming examples. Some demonstrate a different visual idea than presented in the text, others are examples of operators used in a different way.

a&M1.eps 244
a&M2.eps 245
a&M3.eps 246
arc&type_1.eps 227
arc&type_2.eps 228
arcto_2.eps 229
arcto_3.eps 230
bas-relief.eps 275
bkGrid.eps 253
box def1.eps 222
box def2.eps 223
box def4.eps 224
branch2.eps 278
branch3.eps 279
branch4.eps 280
clip 1.eps 247
clip 2.eps 248
closepath_2.eps 220
concat_3.eps 225
curvetoLoops.eps 270
curveto_2.eps 231
curveto_3.eps 232
dashPattern1.eps 217
dashPattern2.eps 218
dashPattern3.eps 219
excited1.eps 262
excited2.eps 263
exploded.eps 264
flower.eps 265
fountain 3-3.eps 241
fountain_LScreen.eps 259
fountainLine.eps 273
gesture.eps 261
grass.eps 212
grayChart.eps 271
grayChartRev.eps 272
linecap.eps 215
line.eps 214

petal2.eps 266
pie_chart2.eps 226
randJazz.eps 260
repeat_box1.eps 234
repeat_box2.eps 235
rotatedFount.eps 242
rotatedFount2.eps 243
setdash2.eps 216
setflat.eps 233
setlinejoin2.eps 221
setscreen1-2.eps 255
setscreen3-4.eps 256
setscreen5-6.eps 257
setscreen7-8.eps 258
slinky.eps 281
slinky2.eps 282
star clip.eps 249
star eoclip.eps 250
star eofill.eps 251
star fill.eps 252
star2.eps 274
stringwidth.eps 240
TCpos&rev.eps 283
theresa3.eps 254
3D-Line1.eps 267
3D-Line2.eps 268
3D-LineStar.eps 269
type_1.eps 236
type_2.eps 237
type_3.eps 238
type_4.eps 239
unencoding1.eps 276
unencoding2.eps 277

Here are two ways of drawing a line:
On top, the finish location is relative to the start location.
Below, start and finish locations are relative to the origin.

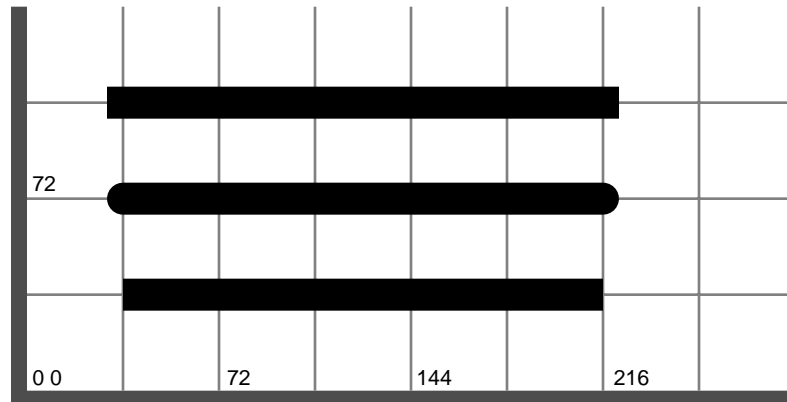


```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:line.eps
%%BoundingBox:36 30 252 75

6 setlinewidth

36 36 moveto 216 36 lineto stroke
36 72 moveto 216 0 rlineto stroke
```


The `setlinecap` operator is used to finish line endings. `0 setlinecap` is the default.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:linecap.eps
%%BoundingBox:32 32 220 112

12 setlinewidth

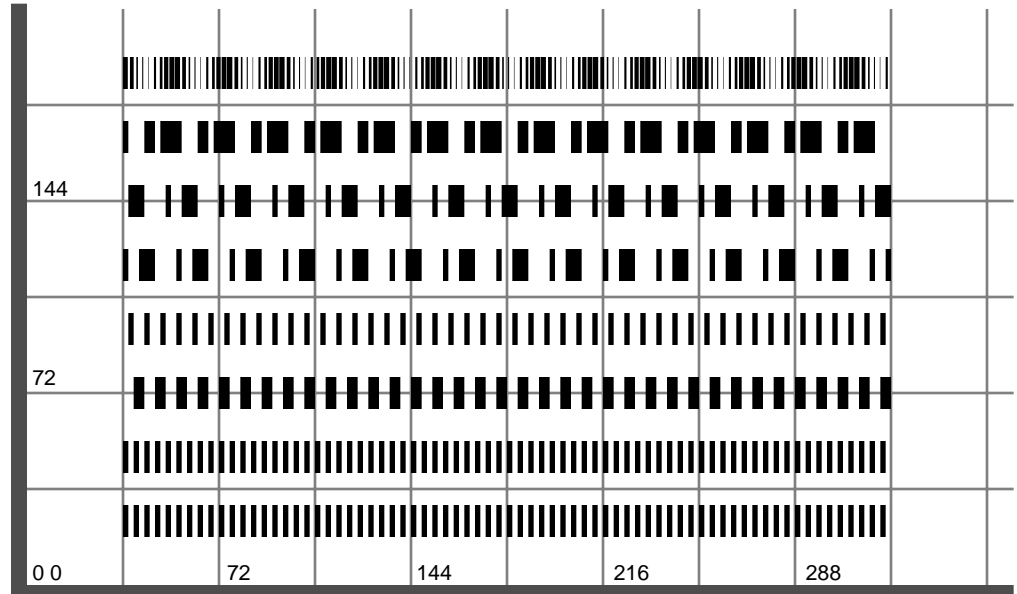
% bottom and default
36 36 moveto 216 36 lineto stroke

% middle
1 setlinecap
36 72 moveto 216 72 lineto stroke

% top
2 setlinecap
36 108 moveto 216 108 lineto stroke

```

The `setdash` operator creates a dashed line.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setdash2.eps
%%BoundingBox:36 18 324 198

6 setlinewidth 2 2 scale
[1] 0 setdash
18 12 moveto 144 0 rlineto stroke

[1] 2 setdash
18 24 moveto 144 0 rlineto stroke

[2] 2 setdash
18 36 moveto 144 0 rlineto stroke

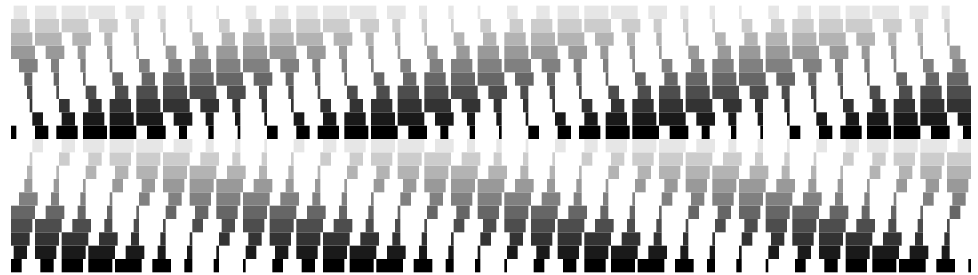
[1 2] 2 setdash
18 48 moveto 144 0 rlineto stroke

[1 2 3 4] 0 setdash
18 60 moveto 144 0 rlineto stroke

[1 2 3 4] 2 setdash
18 72 moveto 144 0 rlineto stroke

[4 3 2 1] 3 setdash
18 84 moveto 144 0 rlineto stroke

[.4 .7 .5 .3 .8 .2 .9 .1 1] 4 setdash
18 96 moveto 144 0 rlineto stroke
    
```



```

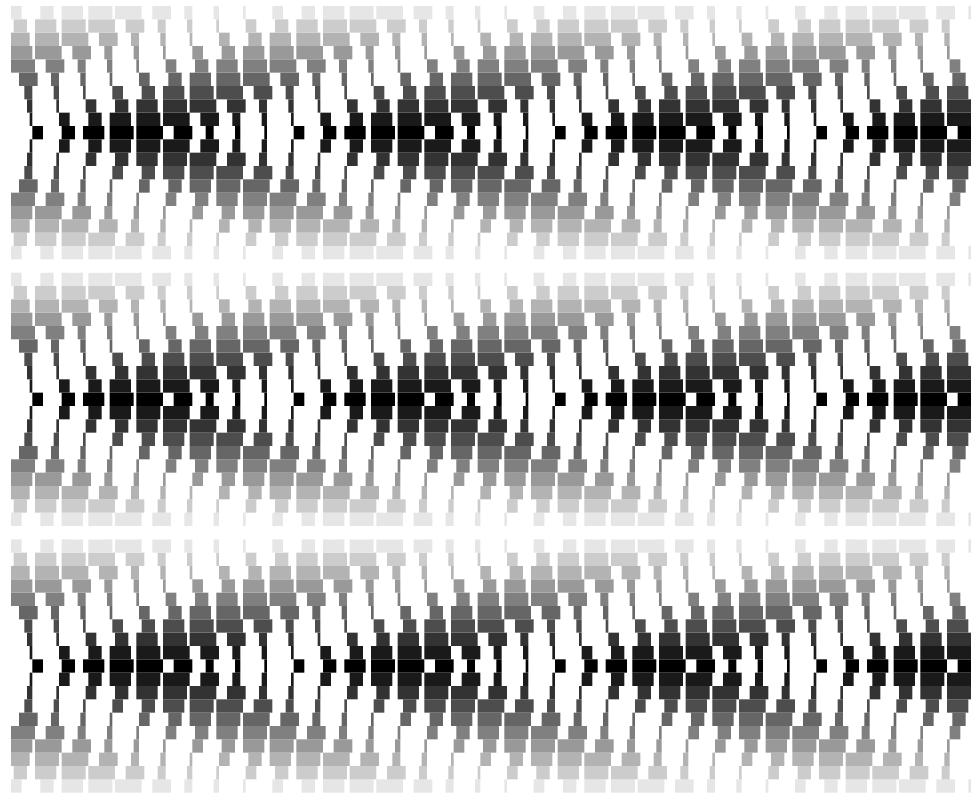
%!PS-Adobe-2.0 EPSF-1.2
%%Title:dashPattern1.eps
%%BoundingBox:0 0 360 100

/line {0 .5 translate 0 0 moveto 36 0 rlineto stroke} def
/dash [.4 .7 .5 .3 .8 .2 .9 .1 1] def

.5 setlinewidth
0 -2.5 translate
10 10 scale

0 setgray dash 0 setdash line
.1 setgray dash 1 setdash line
.2 setgray dash 2 setdash line
.3 setgray dash 3 setdash line
.4 setgray dash 4 setdash line
.5 setgray dash 5 setdash line
.6 setgray dash 6 setdash line
.7 setgray dash 7 setdash line
.8 setgray dash 8 setdash line
.9 setgray dash 9 setdash line
0 setgray dash 10 setdash line
.1 setgray dash 9 setdash line
.2 setgray dash 8 setdash line
.3 setgray dash 7 setdash line
.4 setgray dash 6 setdash line
.5 setgray dash 5 setdash line
.6 setgray dash 4 setdash line
.7 setgray dash 3 setdash line
.8 setgray dash 2 setdash line
.9 setgray dash 1 setdash line

```



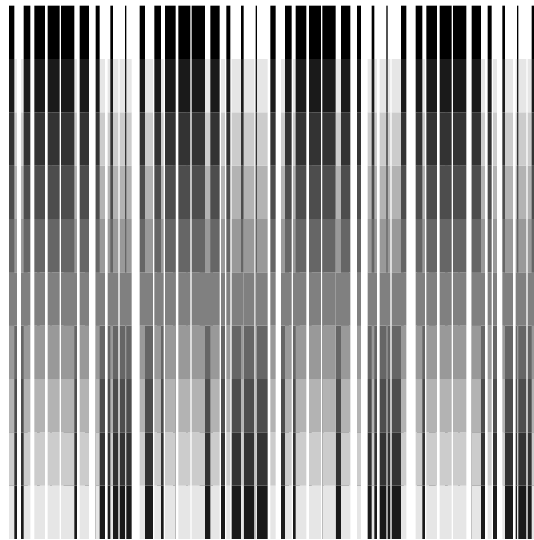
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:dashPattern2.eps
%%BoundingBox:0 0 360 300

/line {0 .5 translate 0 0 moveto 36 0 rlineto stroke} def
/dash [.4 .7 .5 .3 .8 .2 .9 .1 1] def
/x -1 def

.5 setlinewidth
0 -2.5 translate
10 10 scale

/pattern{
  0 .1 1 {} for
  10 {setgray dash x 1 add dup /x exch def setdash line}
repeat
  1 -.1 0 {} for
  10 {setgray dash x 1 sub dup /x exch def setdash line}
repeat
} def

3 {pattern} repeat
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:dashPattern3.eps
%%BoundingBox:0 0 200 200

/line {0 4 translate 0 0 moveto 40 0 rlineto stroke} def
/dashA [.4 .7 .5 .3 .8 .2 .9 .1 1] def
/dashB [.6 .3 .5 .7 .2 .8 .1 .9 1] def

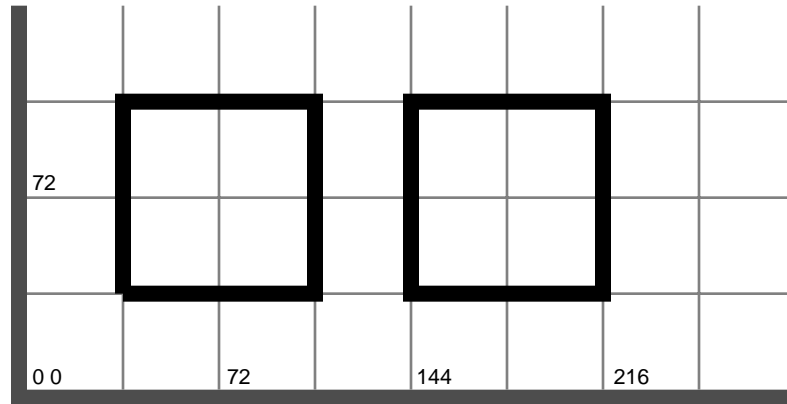
4 setlinewidth
0 -10 translate
5 5 scale

gsave
  1 -.1 0 {} for
    10 {setgray dashB 0 setdash line} repeat
grestore

gsave
  0 .1 1 {} for
    10 {setgray dashA 0 setdash line} repeat
grestore

```

The `closepath` operator is used to finish off the final corner of a path when the end of a path connects with its beginning. The difference `closepath` makes is especially noticeable when the paths are stroked with a thick line. Both squares below have been drawn the same size, but the first does not use `closepath` and the second does.



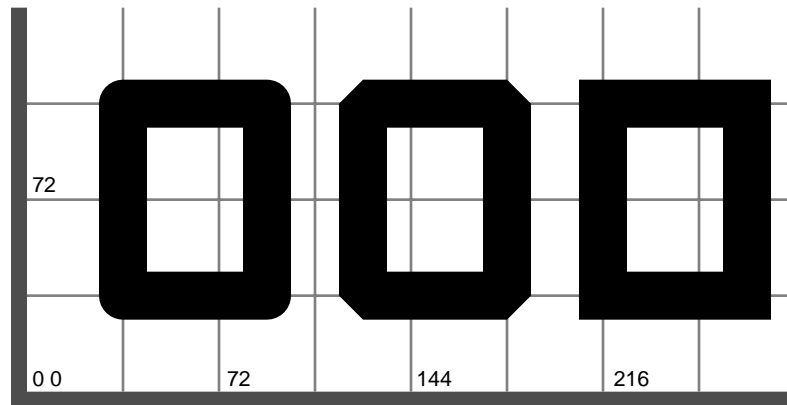
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:closepath_2.eps
%%BoundingBox:32 32 220 112

6 setlinewidth

36 36 moveto
36 108 lineto 108 108 lineto 108 36 lineto
36 36 lineto stroke

144 36 moveto
216 36 lineto 216 108 lineto 144 108 lineto
closepath stroke
```

The `setlinejoin` operator is used to finish corners. 0 `setlinejoin` is the default.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setlinejoin2.eps
%%BoundingBox:27 27 261 117

/rect {54 0 rlineto 0 72 rlineto -54 0 rlineto
       closepath stroke} def

18 setlinewidth

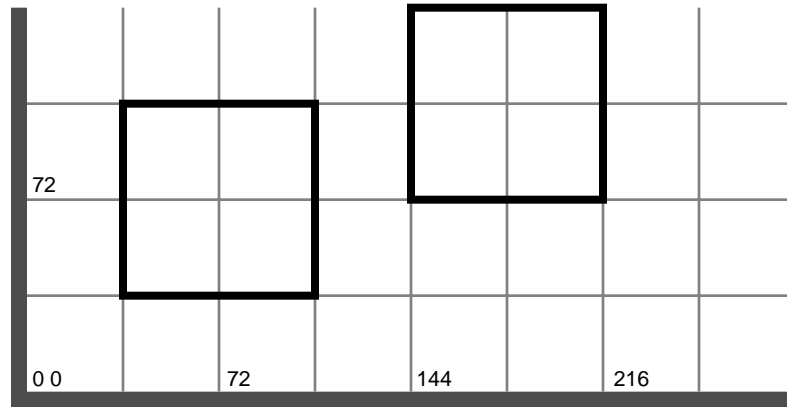
1 setlinejoin      % left, rounded
  36 36 moveto rect

2 setlinejoin      % middle, beveled
 126 36 moveto rect

0 setlinejoin      % right, miter
 216 36 moveto rect

```

Here's an example of defining a procedure. The boxes below are drawn with the `rlineto` operator, or *relative lineto*. Each line is drawn based on the location of the previous current point.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:box def1.eps
%%BoundingBox:34 34 218 146

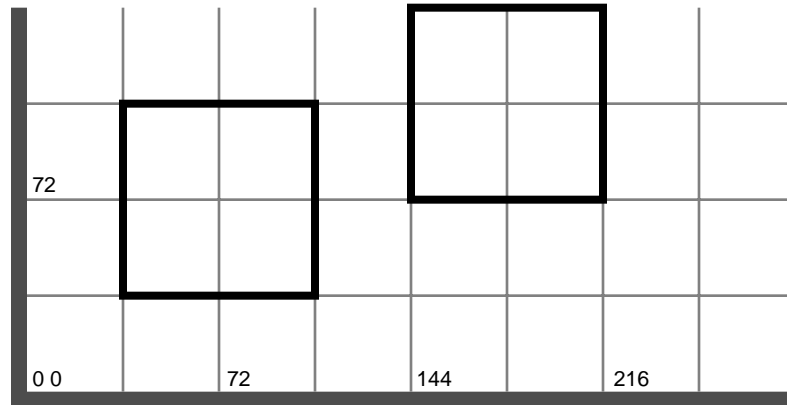
/box { moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
        closepath } def

3 setlinewidth

36 36 box stroke

144 72 box stroke
    
```


This box procedure is defined using the `lineto` operator. Note that the placement for the box is controlled by moving the origin using the `translate` operator. The `gsave` and `grestore` save and restore the graphic state.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:box def2.eps
%%BoundingBox:34 34 218 146

/box { 0 0 moveto 72 0 lineto 72 72 lineto 0 72 lineto
       closepath } def

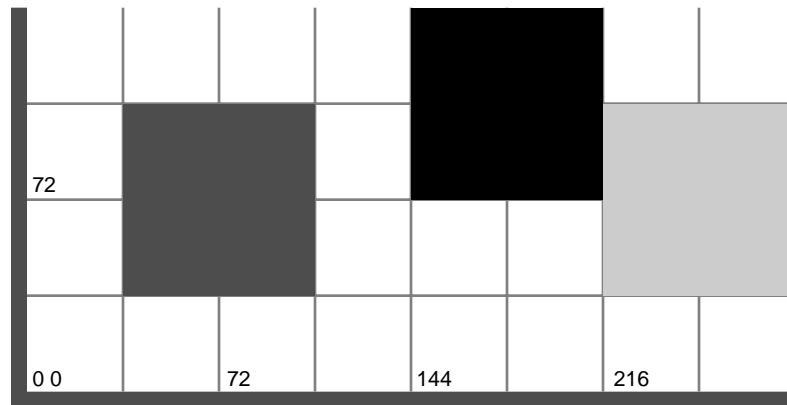
3 setlinewidth

gsave
    36 36 translate
    box stroke
grestore

gsave
    144 72 translate
    box stroke
grestore

```

The box procedure is now filled with a value. 0 equals black (the default) and 1 equals white. Note that the second box is black because the default is in force.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:box def4.eps
%%BoundingBox:36 36 288 144

/box { 0 0 moveto 72 0 lineto 72 72 lineto 0 72 lineto
       closepath } def

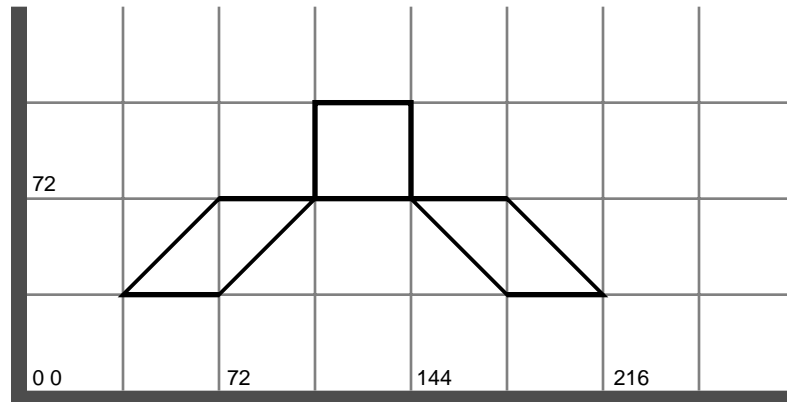
3 setlinewidth

gsave
    .3 setgray
    36 36 translate
    box fill
grestore

gsave
    144 72 translate
    box fill
grestore

gsave
    .8 setgray
    216 36 translate
    box fill
grestore
    
```

The `concat` operator is used to skew the box.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:concat_3.eps
%%BoundingBox:34 34 218 110

/box {0 0 moveto 0 36 rlineto 36 0 rlineto 0 -36 rlineto
      closepath stroke} def

2 setlinewidth

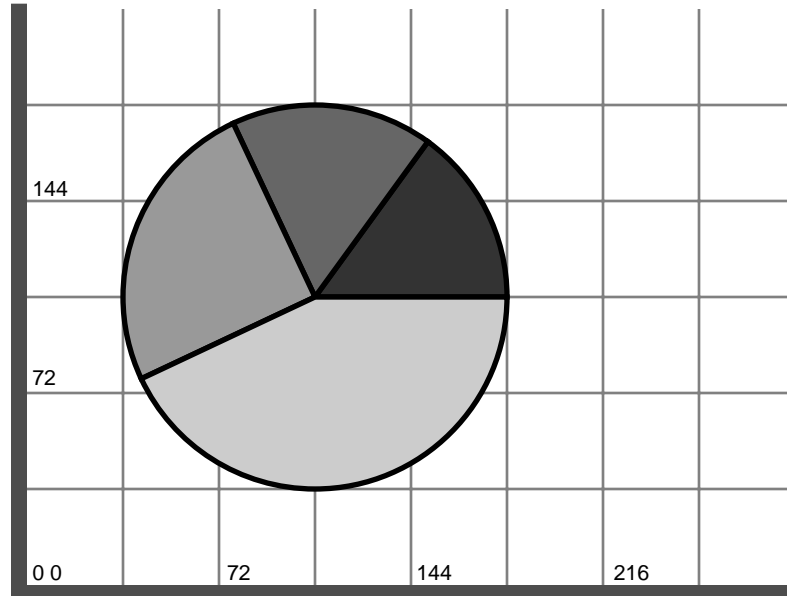
36 36 translate
[1 0 1 1 0 0] concat box

36 36 translate
[1 0 -1 1 0 0] concat box

72 -36 translate
[1 0 -1 1 0 0] concat box

```

The following is a variation of the pie chart example from section 6.2.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:pieChart_2.eps
%%BoundingBox:34 34 182 182

2 setlinewidth
10 10 translate

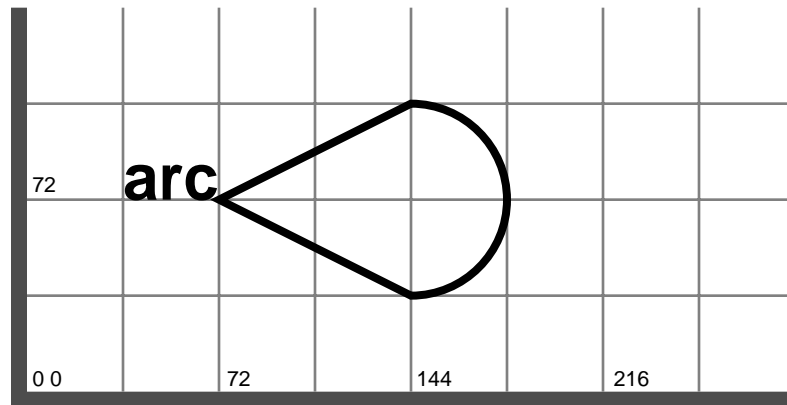
43 25 17 15      % percentages with space in between
/p1 exch 3.6 mul def
/p2 exch 3.6 mul p1 add def
/p3 exch 3.6 mul p2 add def
/p4 exch 3.6 mul p3 add def

/x 72 def
/y 72 def
/r 72 def
/wedge {setgray arc closepath gsave fill grestore
        0 setgray stroke} def

x y moveto
  x y r 0 p1 .2 wedge
x y moveto
  x y r p1 p2 .4 wedge
x y moveto
  x y r p2 p3 .6 wedge
x y moveto
  x y r p3 p4 .8 wedge

```

The `show` operator does not initialize the current point. Therefore, the arc attaches itself by a line to the current point after the word `arc`. `closepath` returns the line.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arc&type_1.eps
%%BoundingBox:36 34 180 110

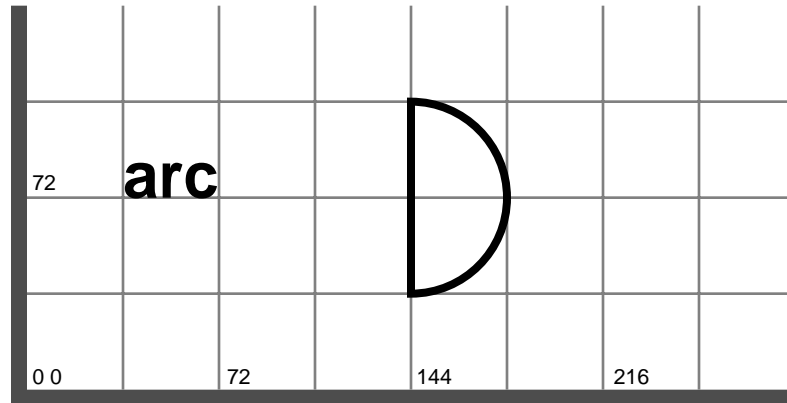
3 setlinewidth

/Helvetica-Bold findfont 24 scalefont setfont
36 72 moveto (arc) show

144 72 36 270 90 arc closepath stroke

```

`newpath` initializes the current point left after the word `arc`. The current point is then the beginning of the arc. This is why `closepath` draws a line from the beginning to the end of the arc.

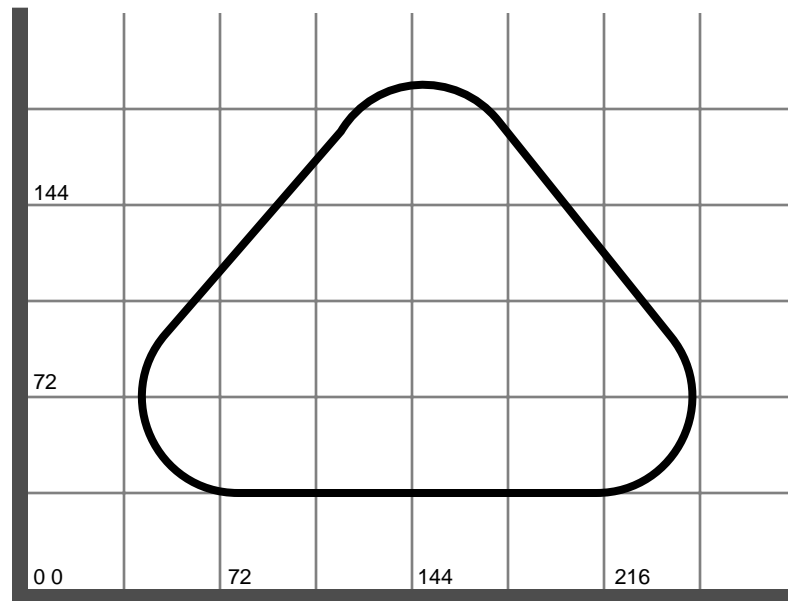


```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:arc&type_2.eps
%%BoundingBox:36 34 180 110

3 setlinewidth

/Helvetica-Bold findfont 24 scalefont setfont
36 72 moveto (arc) show

newpath
144 72 36 270 90 arc closepath stroke
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:arcto_2.eps
%%BoundingBox:40 34 250 190

3 setlinewidth
144 36 moveto

288 36 144 216 36 arcto

144 216 36 36 36 arcto

0 36 108 36 36 arcto

closepath stroke

12 {pop} repeat

```



```
!PS-Adobe-2.0 EPSF-1.2
%%Title:arcto_3.eps
%%BoundingBox:38 36 252 180

3 setlinewidth

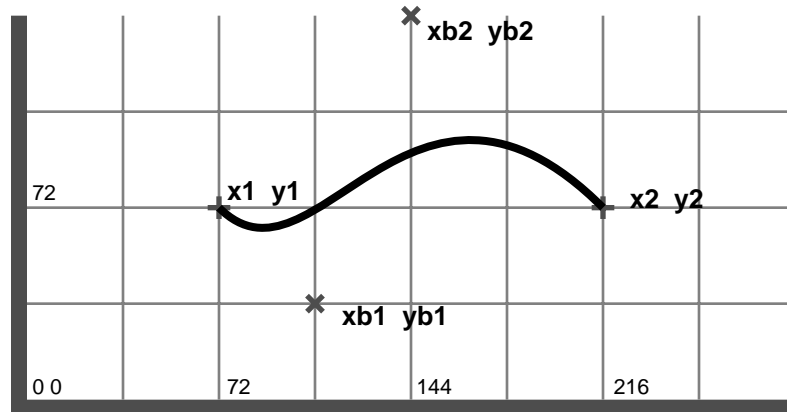
72 180 moveto
0 0 288 216 36 arcto

4 {pop} repeat
252 36 lineto stroke
```


In this example, the Beziér control points are marked to label the `curveto` syntax below:

```
x1 y1 xb2 yb2 x2 y2 curveto
```

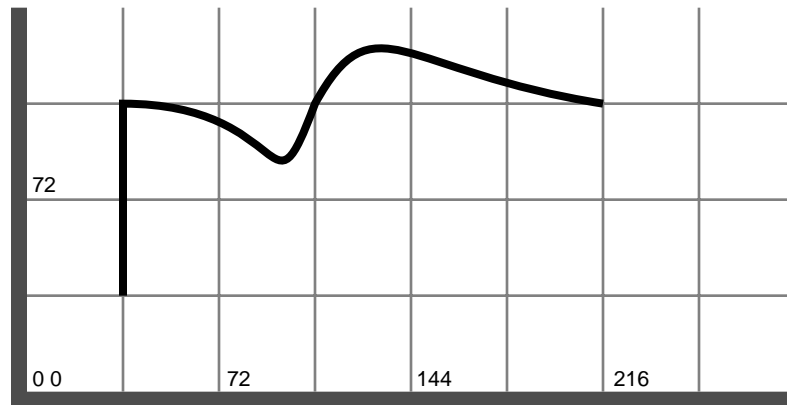
`curveto` requires the existing current point `x1 y1` .



```
%!PS-Adobe-2.0 EPSF-1.2  
%%Title:curveto_2.eps  
%%BoundingBox:72 60 216 96
```

```
3 setlinewidth  
72 72 moveto  
108 36 144 144 216 72 curveto stroke
```

This is another, more involved curve, demonstrating how complex paths can be created.



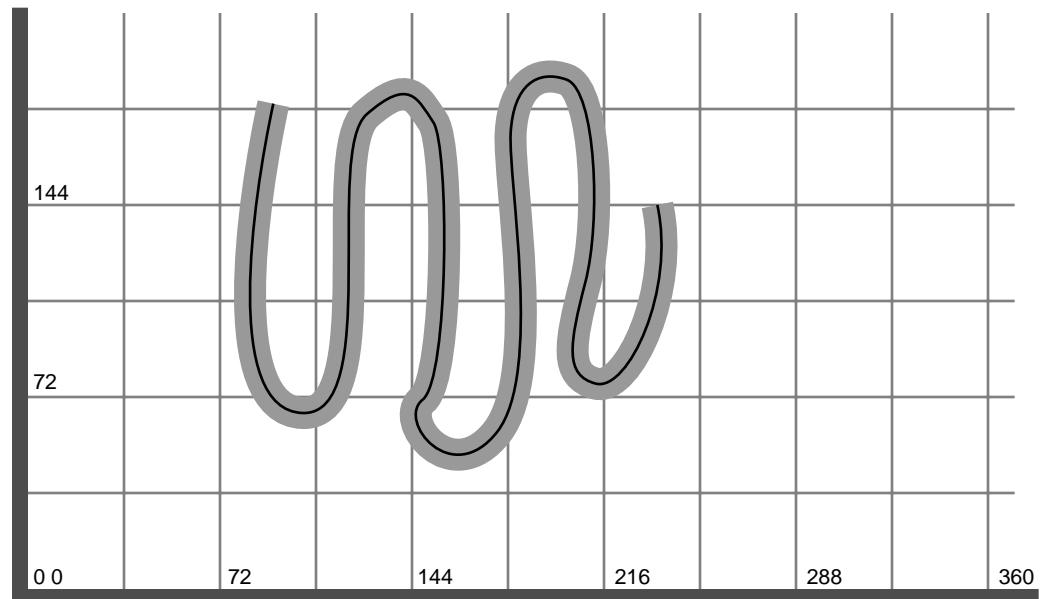
```
!PS-Adobe-2.0 EPSF-1.2
%%Title:curveto_3.eps
%%BoundingBox:34 36 216 132

3 setlinewidth

36 36 moveto
36 108 lineto
100 108 90 60 108 108 curveto
130 148 140 120 216 108 curveto

stroke
```

This program demonstrates how the `setflat` operator can affect the accuracy of a curve's drawing. The final result will depend on the resolution of the printer because this operator is resolution dependent.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setflat.eps
%%BoundingBox:72 40 250 198

/curve {
  56 146 moveto
  48 109 37 31 67 30 curveto
  97 29 75 128 91 142 curveto
  107 156 110 148 116 139 curveto
  122 130 122 43 112 35 curveto
  102 27 123 1 140 23 curveto
  157 45 144 117 145 135 curveto
  146 153 155 159 166 155 curveto
  177 151 179 102 173 79 curveto
  167 56 165 44 177 41 curveto
  189 38 207 76 200 108 curveto
} def

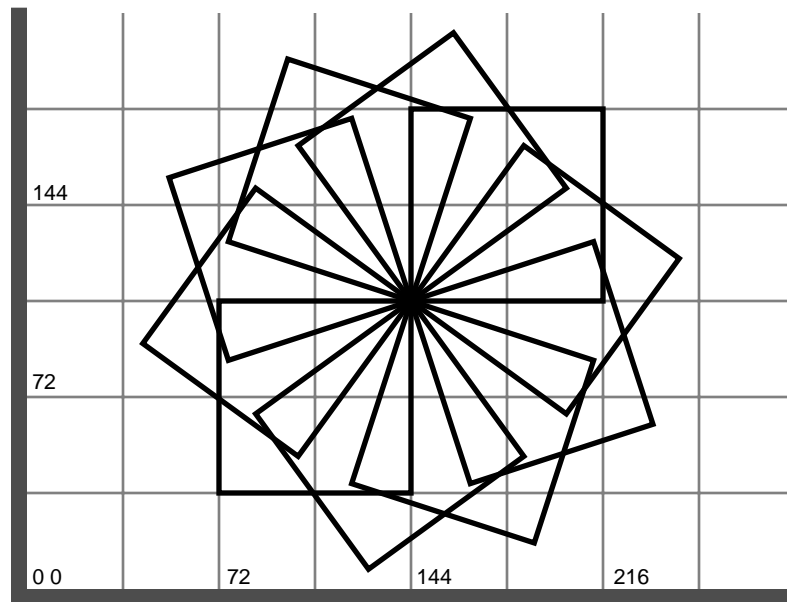
36 36 translate

gsave      % thick gray line
  .6 setgray
  12 setlinewidth
  90 setflat
  curve stroke
grestore

curve stroke      % thin black line

```

The `rotate` operator rotates the current transformation matrix or CTM. By repeating that rotation, this design can be made.



```

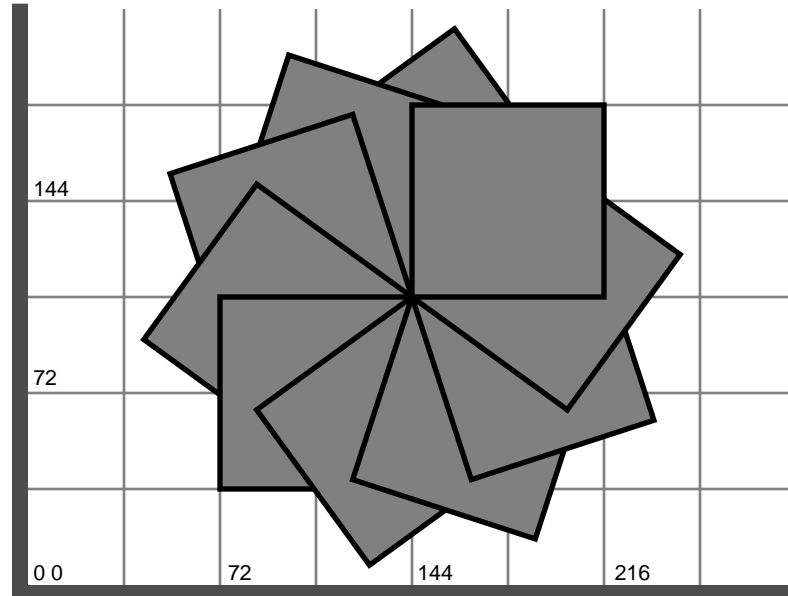
%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_box1.eps
%%BoundingBox:36 0 252 288

/box {
    0 0 moveto
    0 72 rlineto 72 0 rlineto 0 -72 rlineto
    closepath stroke
} def

144 108 translate
2 setlinewidth

10 { 36 rotate box } repeat
    
```

This is a variation of the previous example using a filled and stroked square.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:repeat_box2.eps
%%BoundingBox:36 0 252 288

/box {
  0 0 moveto
  0 72 rlineto 72 0 rlineto 0 -72 rlineto
  closepath
  gsave
    .5 setgray fill
  grestore
  stroke
} def

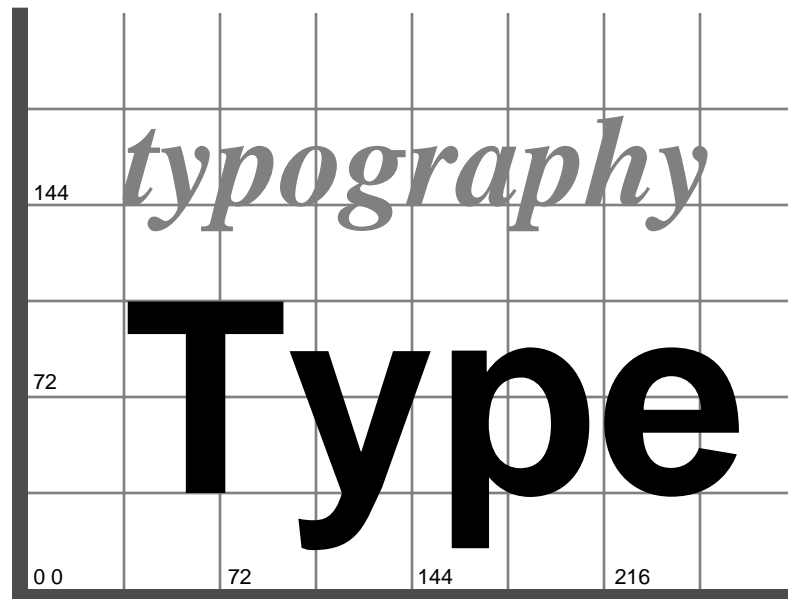
144 108 translate
2 setlinewidth

10 { 36 rotate box} repeat

```

The `show` operator in the example below paints a string of characters starting at a current point, with the current font, in the current color.

`moveto` establishes the current point, `/fontname findfont pointsize scalefont setfont` establishes the current font, and `value setgray` establishes the current color.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_1.eps
%%BoundingBox:36 12 270 180

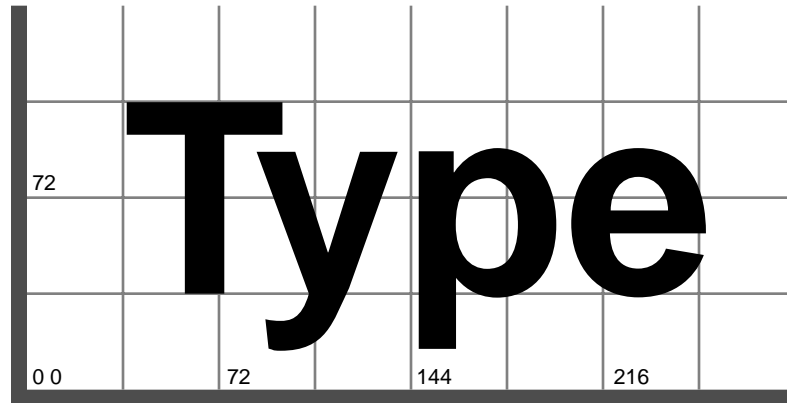
/Helvetica-Bold findfont 100 scalefont setfont

36 36 moveto
(Type) show

/Times-BoldItalic findfont 48 scalefont setfont

36 144 moveto
.5 setgray
(typography) show
    
```

The `show` operator does not initialize the current point. In the example below, `rmoveto` adjusts the character spacing of the word *Type* by moving the current point left after the setting of the *T*.



```

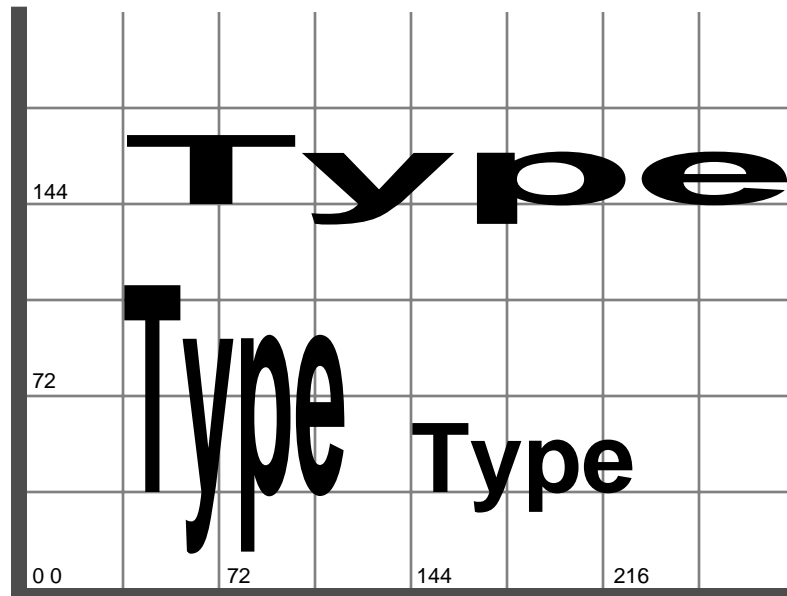
%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_2.eps
%%BoundingBox:36 12 255 110

/Helvetica-Bold findfont 100 scalefont setfont

36 36 moveto
(T) show -12 0 rmoveto (ype) show

```

The program below contains examples of the `makefont` operator.



```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_3.eps
%%BoundingBox:36 12 288 168

% top
/Helvetica-Bold findfont [108 0 0 36 0 0] makefont setfont

36 144 moveto
(Type) show

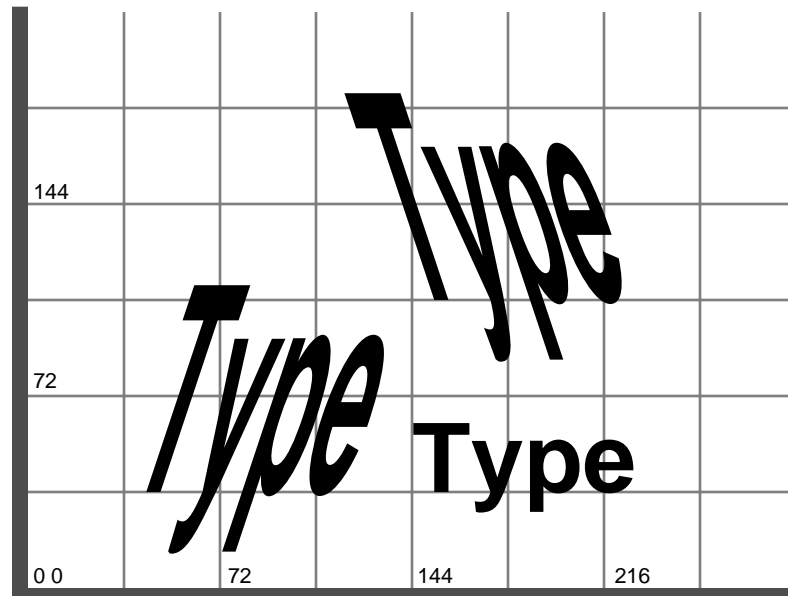
% bottom left
/Helvetica-Bold findfont [36 0 0 108 0 0] makefont setfont

36 36 moveto
(Type) show

% bottom right normal
/Helvetica-Bold findfont [36 0 0 36 0 0] makefont setfont

144 36 moveto
(Type) show
```


Here are more examples of the `makefont` operator.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:type_4.eps
%%BoundingBox:40 12 228 190

% top
/Helvetica-Bold findfont [36 0 -36 108 0 0] makefont setfont

144 108 moveto
(Type) show

% bottom left
/Helvetica-Bold findfont [36 0 36 108 0 0] makefont setfont

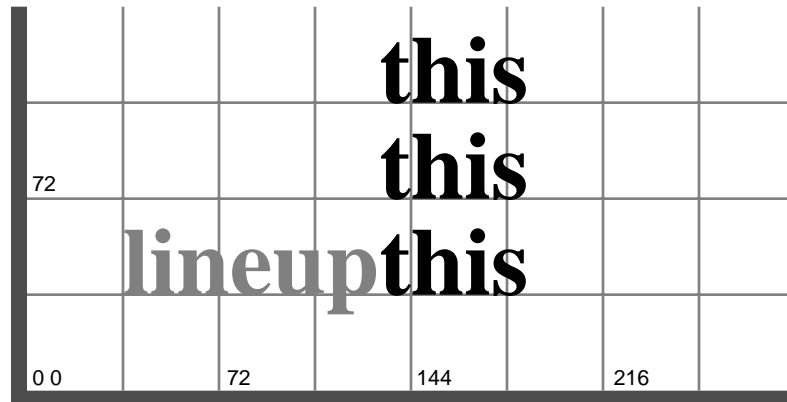
36 36 moveto
(Type) show

% bottom right normal
/Helvetica-Bold findfont [36 0 0 36 0 0] makefont setfont

144 36 moveto
(Type) show

```

The following is an example of the `stringwidth` operator.



```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:stringwidth.eps
%%BoundingBox:36 30 200 144

/Times-Bold findfont 36 scalefont setfont

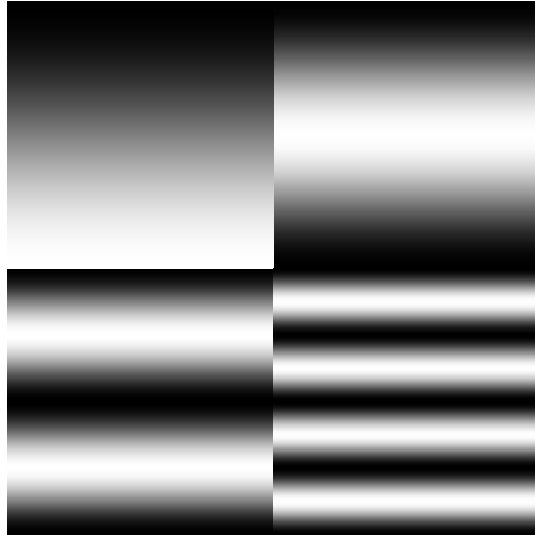
/width 36 (lineup) stringwidth pop add def

.5 setgray

36 36 moveto
(lineup) show 0 setgray (this) show

width 72 moveto
(this) show

width 108 moveto
(this) show
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_3-3.eps
%%BoundingBox:0 0 200 200

/fountain {
  /str 256 string def
  0 1 255 { str exch dup
    255 div change mul cos neg 2 div .5 add 255 mul cvi put
  } for

  /ury exch def /urx exch def
  /lly exch def /llx exch def
  gsave
    llx lly translate
    urx llx sub ury lly sub scale
    1 256 8[1 0 0 -256 0 256] {str} image
  grestore } def

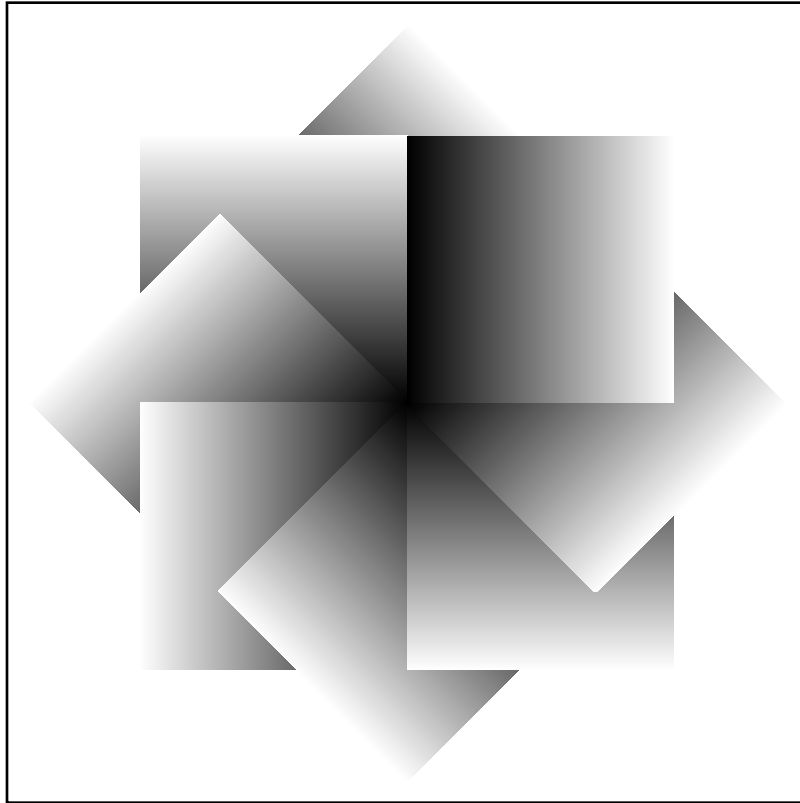
/change 720 def
0 0 100 100 fountain

/change 360 def
100 100 200 200 fountain

/change 180 def
0 100 100 200 fountain

/change 1440 def
100 0 200 100 fountain

```

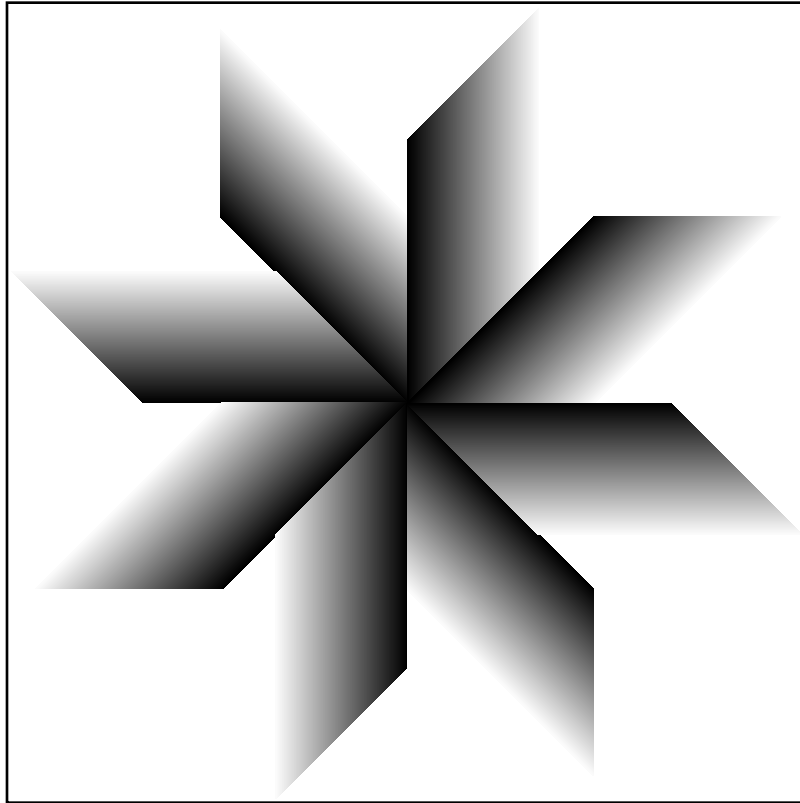


```
!PS-Adobe-2.0 EPSF-1.2
%%Title:rotatedFont.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 300 300

/str 256 string def
0 1 255 { str exch dup put } for

0 0 300 300 rectstroke
150 150 translate

gsave
  8 {45 rotate
    gsave
      100 100 scale
      255 1 8 [255 0 0 1 0 0] {str} image
    grestore} repeat
grestore
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rotatedFont2.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 300 300

/str 256 string def
0 1 255 { str exch dup put } for

0 0 300 300 rectstroke
150 150 translate

gsave
  8 {45 rotate
    gsave
      70 70 scale
      255 1 8 [255 0 -255 1 0 0] {str} image
    grestore} repeat
grestore

```

The following is a logo variation demonstrating clipping with type.



```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:a&M1.eps
%%BoundingBox:20 19 295 295

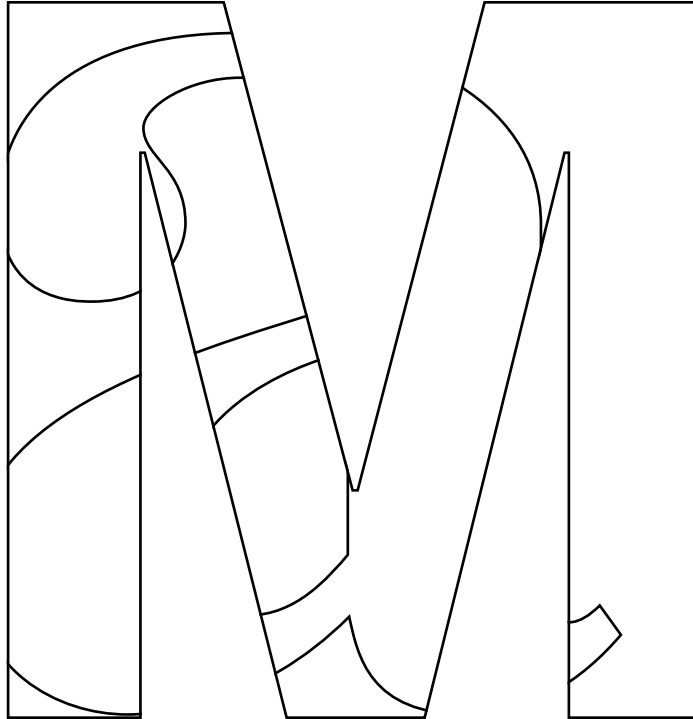
% a is filled
% M is filled

/inside /Times-Bold findfont 525 scalefont def
/outline /Helvetica-Bold findfont 375 scalefont def

0 20 translate

0 0 moveto outline setfont (M) true charpath clip
.2 setgray fill

.5 setgray
0 9 moveto
inside setfont (a) show
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:a&M2.eps
%%BoundingBox:20 19 295 295

% both letterforms are outlined

/inside /Times-Bold findfont 525 scalefont def
/outline /Helvetica-Bold findfont 375 scalefont def

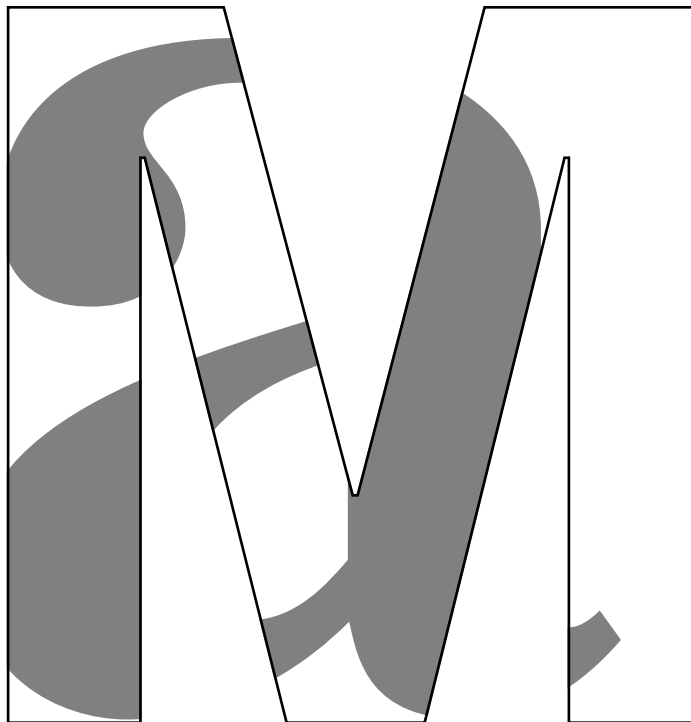
0 20 translate

0 0 moveto outline setfont (M) true charpath clip newpath

0 9 moveto
inside setfont (a) true charpath stroke

2 setlinewidth
0 0 moveto outline setfont (M) true charpath stroke

```



```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:a&M3.eps
%%BoundingBox:20 19 295 295

% a is filled
% M is outlined

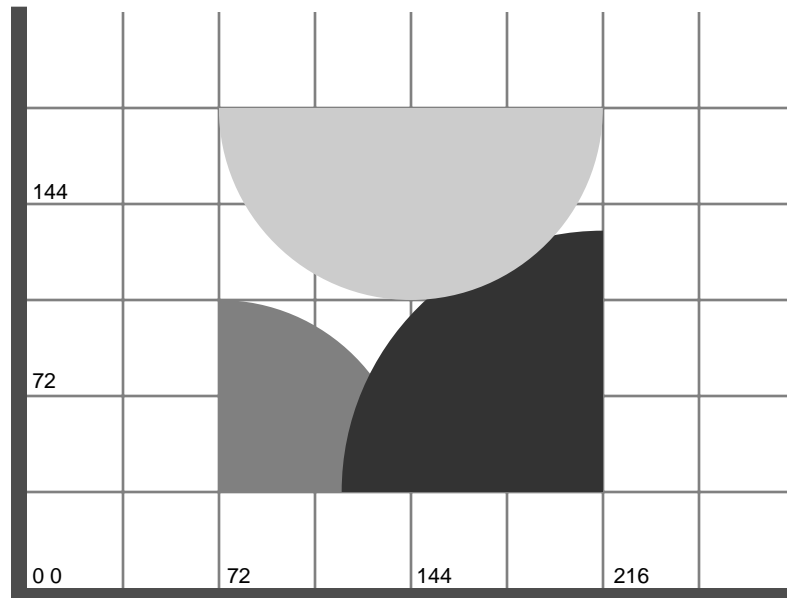
/inside /Times-Bold findfont 525 scalefont def
/outline /Helvetica-Bold findfont 375 scalefont def

0 20 translate

0 0 moveto outline setfont (M) true charpath clip newpath

.5 setgray
0 9 moveto
inside setfont (a) show

0 setgray
2 setlinewidth
0 0 moveto outline setfont (M) true charpath stroke
```

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:clip_1.eps
%%BoundingBox:72 36 216 180

72 36 moveto 0 144 rlineto 144 0 rlineto 0 -144 rlineto

closepath clip

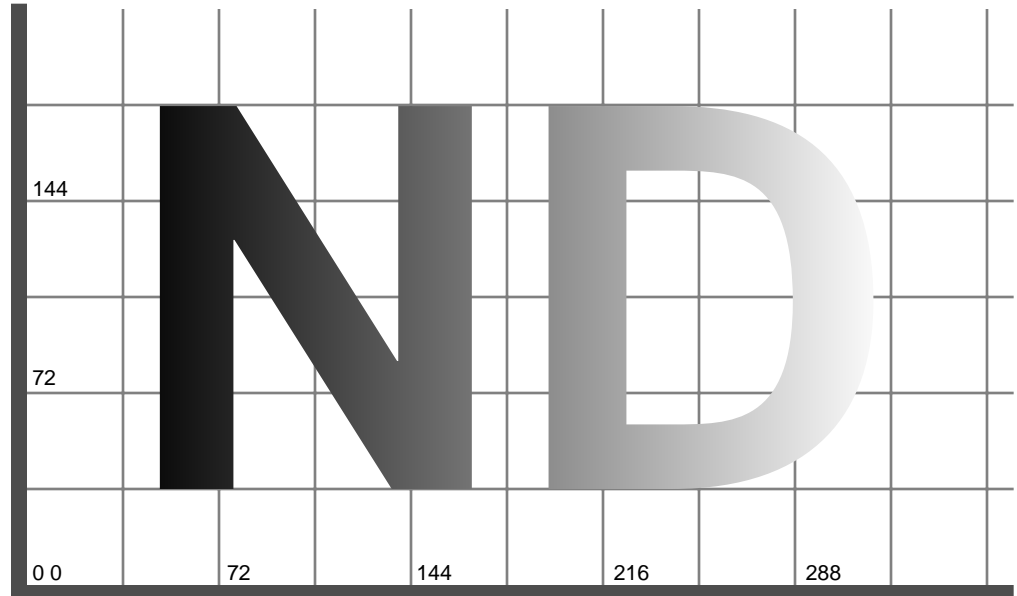
newpath

72 36 72 0 360 arc
.5 setgray fill

216 36 98 0 360 arc
.2 setgray fill

144 180 72 0 360 arc
.8 setgray fill

```



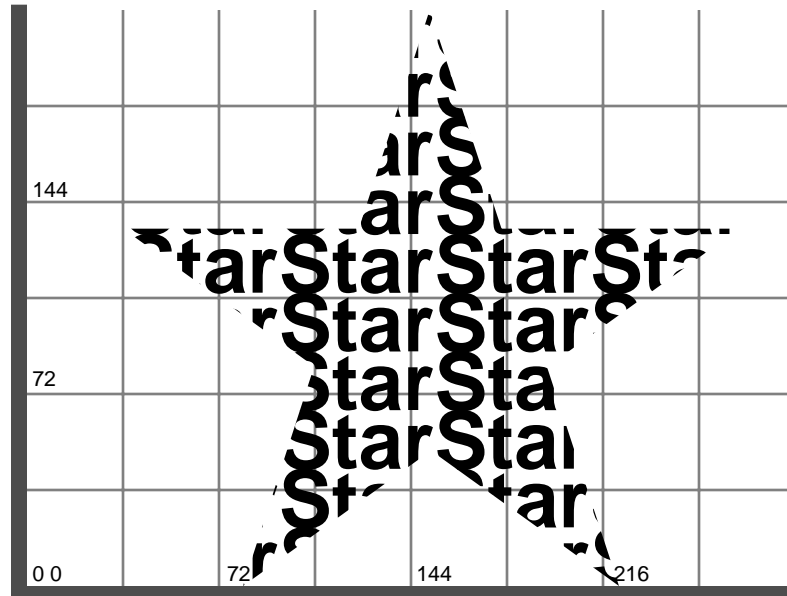
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:clip_2.eps
%%BoundingBox:48 36 324 184

36 36 translate

0 0 moveto
/Helvetica-Bold findfont 200 scalefont setfont
(ND) true charpath clip

/str 256 string def
0 1 255 { str exch dup put } for

288 170 scale
255 1 8 [ 255 0 0 1 0 0 ] {str} image
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:star clip.eps
%%BoundingBox:36 0 270 216

/str (StarStarStarStar) def
/leading { currentpoint 22 add exch pop 0 exch moveto } def
/background { str show leading } def
/Helvetica-Bold findfont 30 scalefont setfont

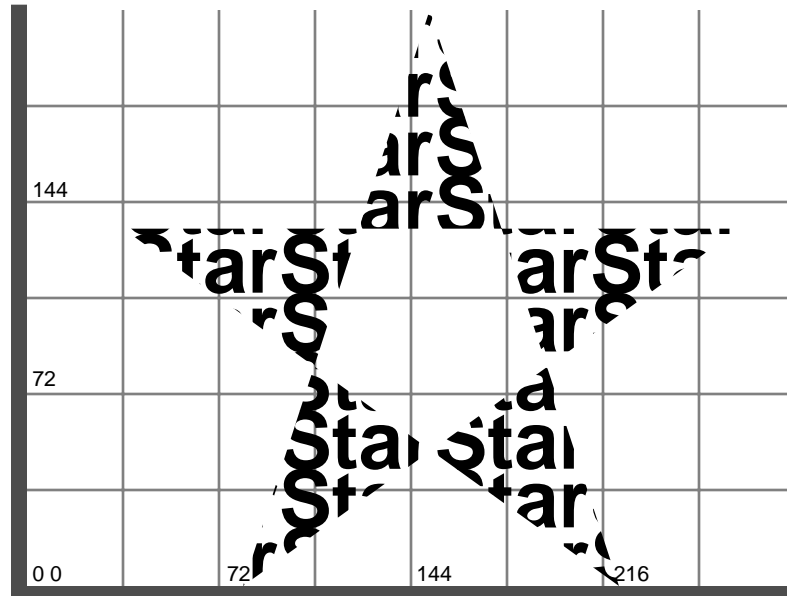
36 0 translate

% star
45 0 moveto 115 216 lineto 186 0 lineto 0 134 lineto
231 134 lineto 45 0 lineto closepath

clip
0 0 moveto 10 {background} repeat

```

The following is an example of `eoclip`.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:star eoclip.eps
%%BoundingBox:36 0 270 216

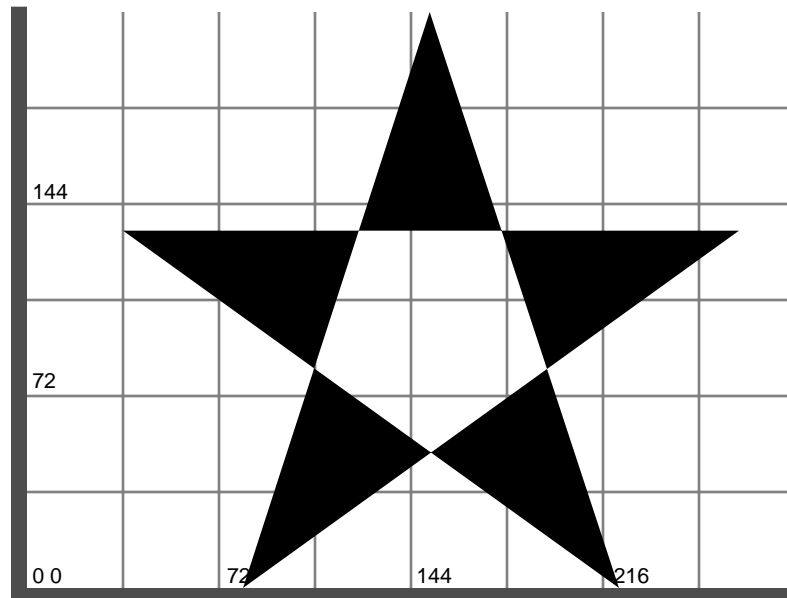
/str (StarStarStarStar) def
/leading { currentpoint 22 add exch pop 0 exch moveto } def
/background { str show leading } def
/Helvetica-Bold findfont 30 scalefont setfont

36 0 translate

% star
45 0 moveto 115 216 lineto 186 0 lineto 0 134 lineto
231 134 lineto 45 0 lineto closepath

eoclip
0 0 moveto 10 {background} repeat
    
```

The following is an example of `eofill`.



```

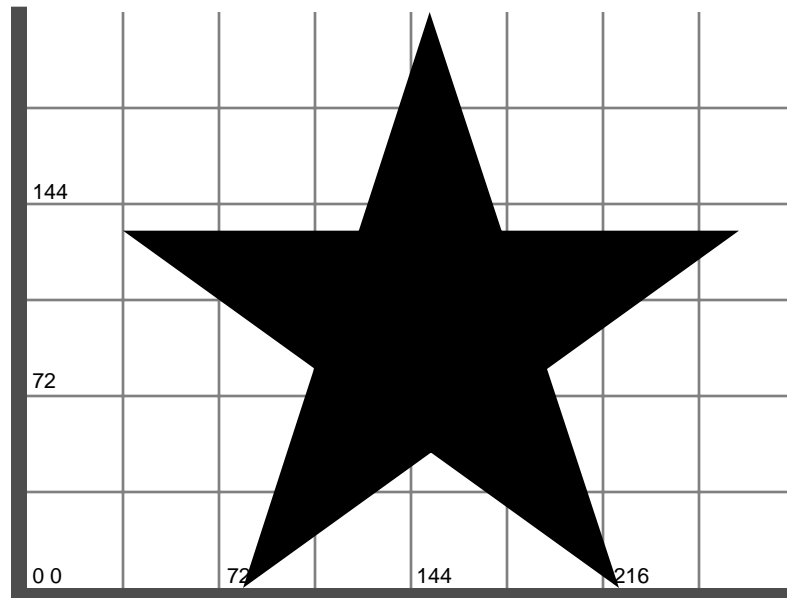
%!PS-Adobe-2.0 EPSF-1.2
%%Title:star eofill.eps
%%BoundingBox:36 0 270 216

36 0 translate

% star
45 0 moveto 115 216 lineto 186 0 lineto 0 134 lineto
231 134 lineto 45 0 lineto closepath

eofill

```



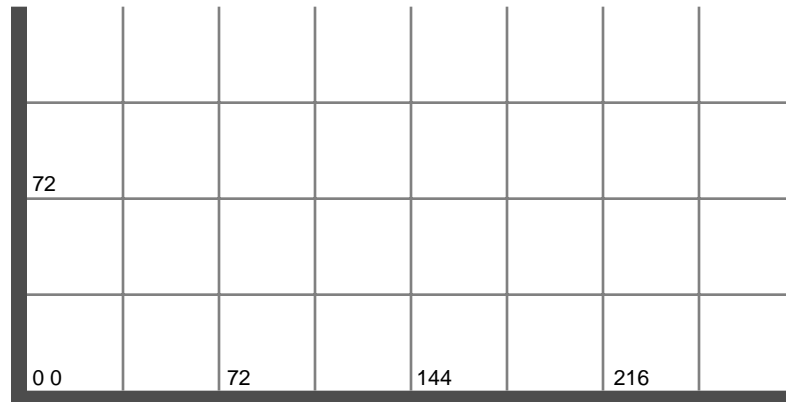
```
!PS-Adobe-2.0 EPSF-1.2
%%Title:star fill.eps
%%BoundingBox:36 0 270 216

36 0 translate

% star
45 0 moveto 115 216 lineto 186 0 lineto 0 134 lineto
231 134 lineto 45 0 lineto closepath

fill
```

This is the background grid for many of the examples.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:bkGrid.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 294 150

/vline{0 0 moveto 0 144 lineto stroke} def
/hline{0 0 moveto 288 0 lineto stroke} def

.3 setgray
6 setlinewidth
3 150 moveto 3 3 lineto 294 3 lineto stroke

.5 setgray
1 setlinewidth
6 6 translate

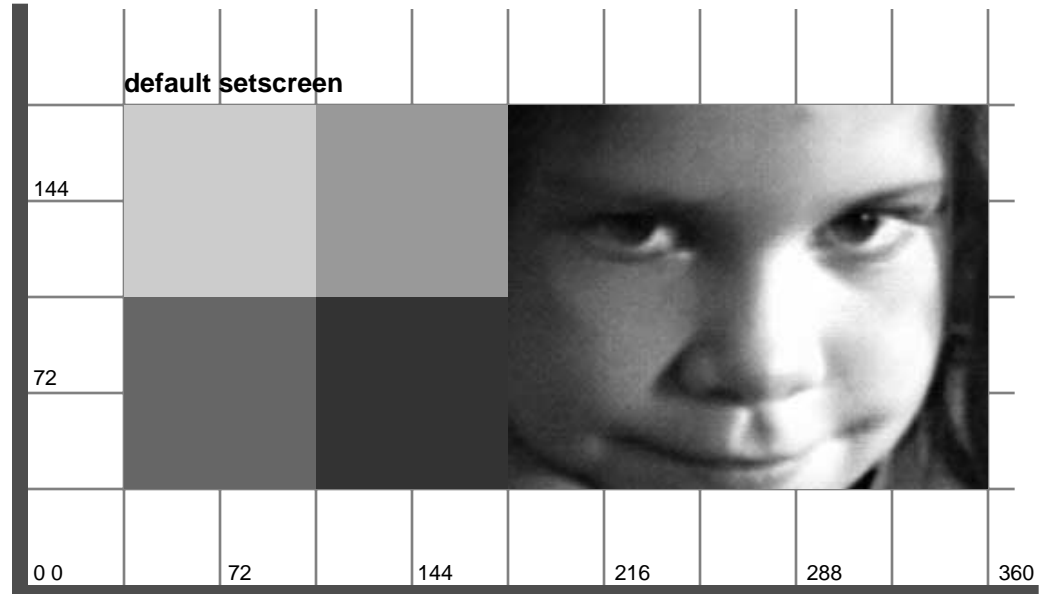
gsave
  7 {36 0 translate vline} repeat
grestore

gsave
  3 {0 36 translate hline} repeat
grestore

0 setgray
/Helvetica findfont 8 scalefont setfont
2 2 moveto (0 0) show
75 2 moveto (72) show
146 2 moveto (144) show
220 2 moveto (216) show
2 75 moveto (72) show

```

In this example, the `setscreen` is commented out for comparison with the other `setscreen` examples. Most of the scanned image is deleted to save space.



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:theresa3.eps
%%BoundingBox:36 36 360 198

/box {moveto 72 0 rlineto 0 72 rlineto -72 0 rlineto
      closepath fill} def

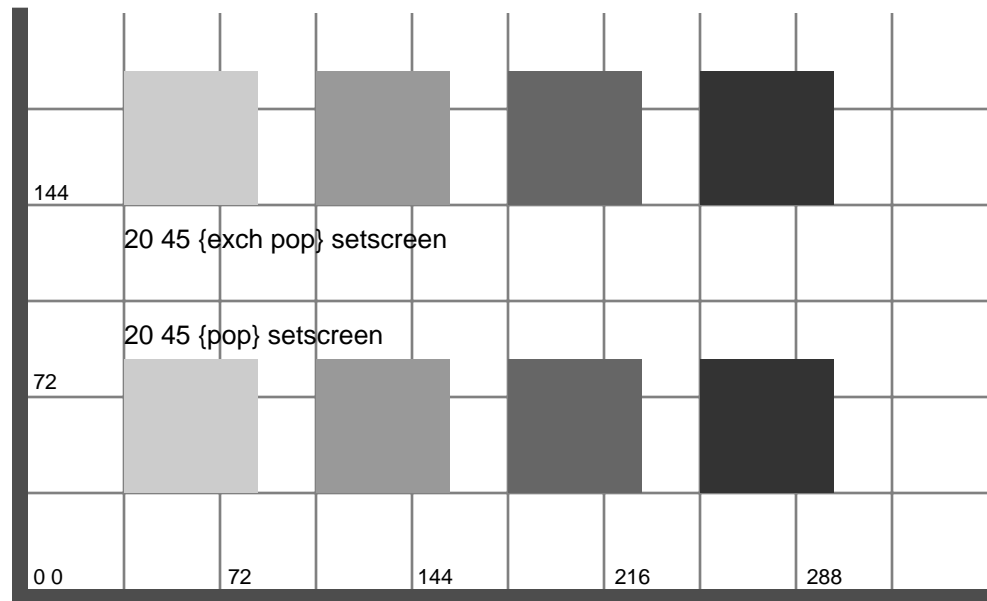
/Helvetica-Bold findfont 10 scalefont setfont
36 185 moveto (default setscreen) show

.8 setgray 36 108 box
.6 setgray 108 108 box
.4 setgray 36 36 box
.2 setgray 108 36 box

% scanned picture

%%Title: Theresa.eps
. . . header info . . .
72 65536 mul 4718592 div dup cols mul exch rows mul scale
cols rows 8 [cols 0 0 rows neg 0 rows]
beginimage
0B0A0D0A070E0F0B090D0D110F0E13161515151B17181A141D1A1916181
B1616
. . . . data . . .
32130A0707050A31607E797B6D5B4D4A514E4C4B4D4E51564C525246434
74F51
565D5D616D6457514F3F352C2118121612161716
grestore end

```

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setscreen1-2.eps
%%BoundingBox:36 36 306 198

/Helvetica findfont 10 scalefont setfont
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
closepath fill} def

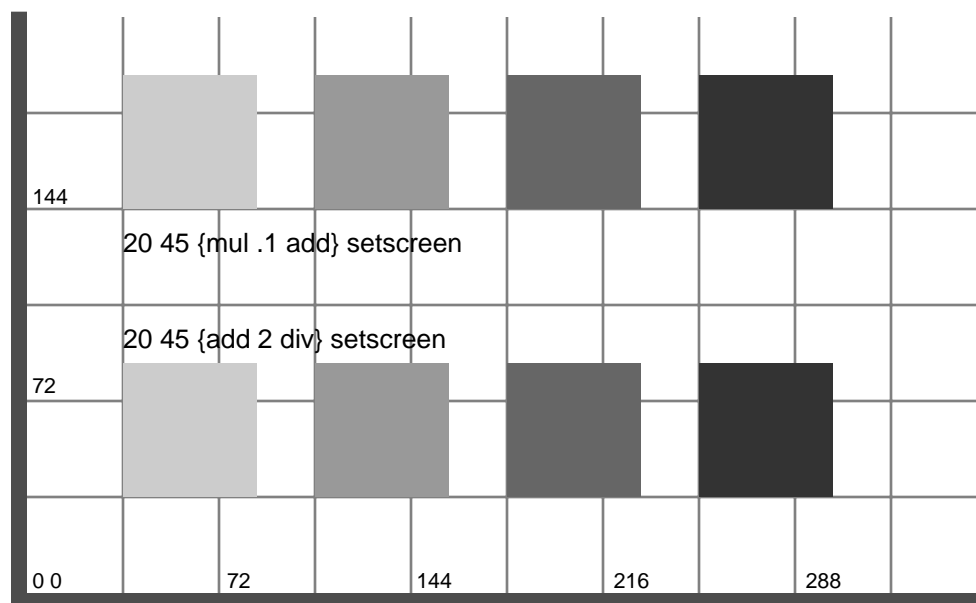
20 45 {exch pop} setscreen

36 128 moveto (20 45 {exch pop} setscreen) show
.8 setgray 36 144 box
.6 setgray 108 144 box
.4 setgray 180 144 box
.2 setgray 252 144 box

0 setgray
20 45 {pop} setscreen

36 92 moveto (20 45 {pop} setscreen) show
.8 setgray 36 36 box
.6 setgray 108 36 box
.4 setgray 180 36 box
.2 setgray 252 36 box

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setscreen3-4.eps
%%BoundingBox:36 36 306 198

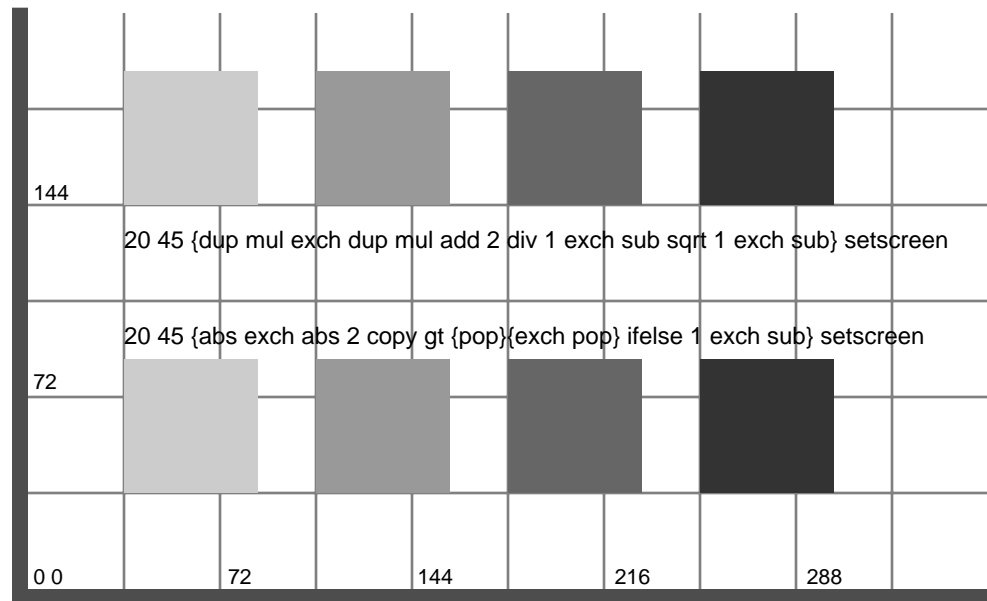
/Helvetica findfont 10 scalefont setfont
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
      closepath fill} def

20 45 {mul .1 add} setscreen

36 128 moveto (20 45 {mul .1 add} setscreen) show
.8 setgray 36 144 box
.6 setgray 108 144 box
.4 setgray 180 144 box
.2 setgray 252 144 box

0 setgray
20 45 {add 2 div} setscreen

36 92 moveto (20 45 {add 2 div} setscreen) show
.8 setgray 36 36 box
.6 setgray 108 36 box
.4 setgray 180 36 box
.2 setgray 252 36 box
    
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setscreen5-6.eps
%%BoundingBox:36 36 342 198

/Helvetica findfont 9 scalefont setfont
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
closepath fill} def

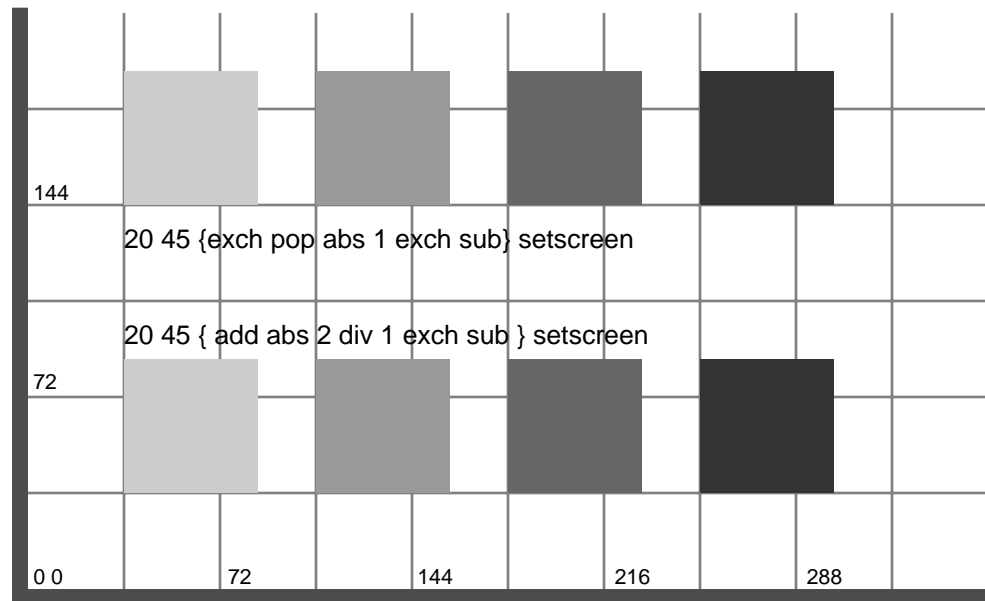
20 45 {dup mul exch dup mul add 2 div 1 exch sub sqrt
1 exch sub} setscreen

36 128 moveto (20 45 {dup mul exch dup mul add 2 div 1 exch
sub sqrt 1 exch sub} setscreen) show
.8 setgray 36 144 box
.6 setgray 108 144 box
.4 setgray 180 144 box
.2 setgray 252 144 box

0 setgray
20 45 {abs exch abs 2 copy gt {pop}{exch pop} ifelse
1 exch sub} setscreen

36 92 moveto (20 45 {abs exch abs 2 copy gt {pop}{exch pop}
ifelse 1 exch sub} setscreen) show
.8 setgray 36 36 box
.6 setgray 108 36 box
.4 setgray 180 36 box
.2 setgray 252 36 box

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:setscreen7-8.eps
%%BoundingBox:36 36 306 198

/Helvetica findfont 10 scalefont setfont
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
closepath fill} def

20 45 {exch pop abs 1 exch sub} setscreen

36 128 moveto (20 45 {exch pop abs 1 exch sub} setscreen) show
.8 setgray 36 144 box
.6 setgray 108 144 box
.4 setgray 180 144 box
.2 setgray 252 144 box

0 setgray
20 45 {add abs 2 div 1 exch sub} setscreen

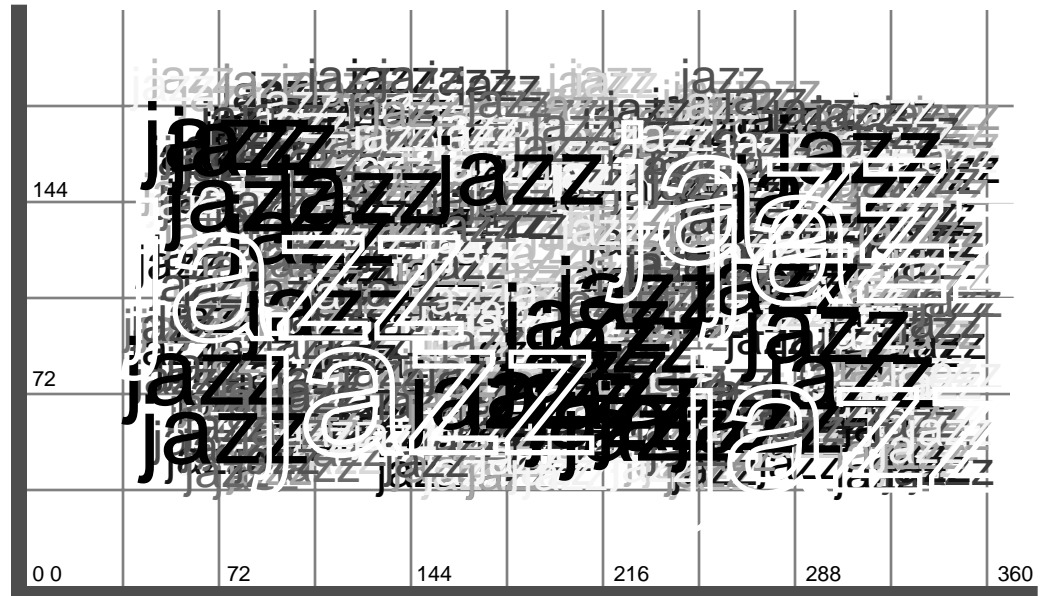
36 92 moveto (20 45 {add abs 2 div 1 exch sub} setscreen) show
.8 setgray 36 36 box
.6 setgray 108 36 box
.4 setgray 180 36 box
.2 setgray 252 36 box
    
```



```
!PS-Adobe-2.0 EPSF-1.2
%%Title:fountain_LScreen.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 170 170

/str 256 string def
0 1 255 { str exch dup put } for
20 90 {pop} setscreen
170 170 scale

255 1 8 [ 255 0 0 1 0 0 ] {str} image
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:randJazz.eps
%%BoundingBox:34 34 366 198

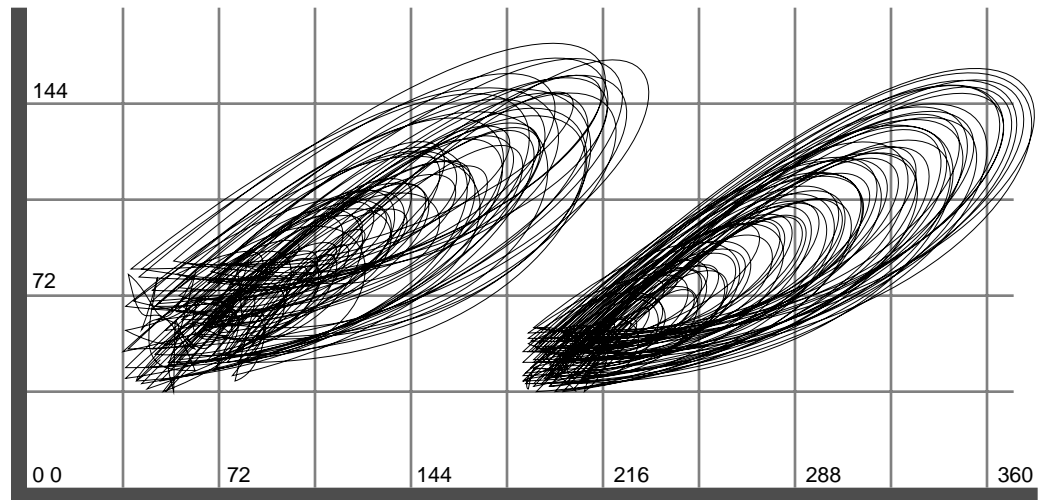
/n {rand exch mod} def
/gray {100 n .01 mul} def

171717 srand
36 36 translate

/Helvetica findfont 18 scalefont setfont
  25 {gray setgray
    25 {300 n 150 n moveto (jazz) show} repeat} repeat

/Helvetica findfont 36 scalefont setfont
  0 setgray
  25 {250 n 125 n moveto (jazz) show} repeat

/Helvetica findfont 72 scalefont setfont
  2 setlinewidth 1 setgray
  5 {250 n 100 n moveto
    (jazz) true charpath stroke} repeat
    
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:gesture.eps
%%BoundingBox:36 36 378 170

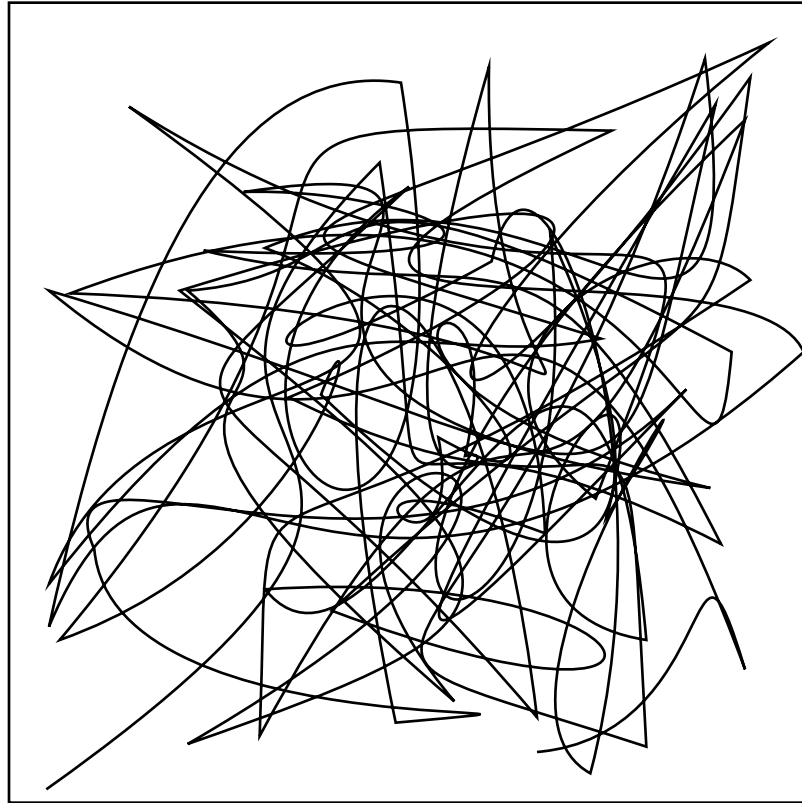
/n      85 def
/nnn   n 3 mul 1 sub def
/x1    {rand 50 mod} bind def
/x2    {rand 25 mod} bind def

36 36 translate

.25 setlinewidth
x1 x1 moveto
0 1 nnn {} for
n {x1 x1 x1 curveto} repeat stroke

150 0 translate
.25 setlinewidth
x2 x2 moveto
0 1 nnn {} for
n {x2 x2 x2 curveto} repeat stroke

```

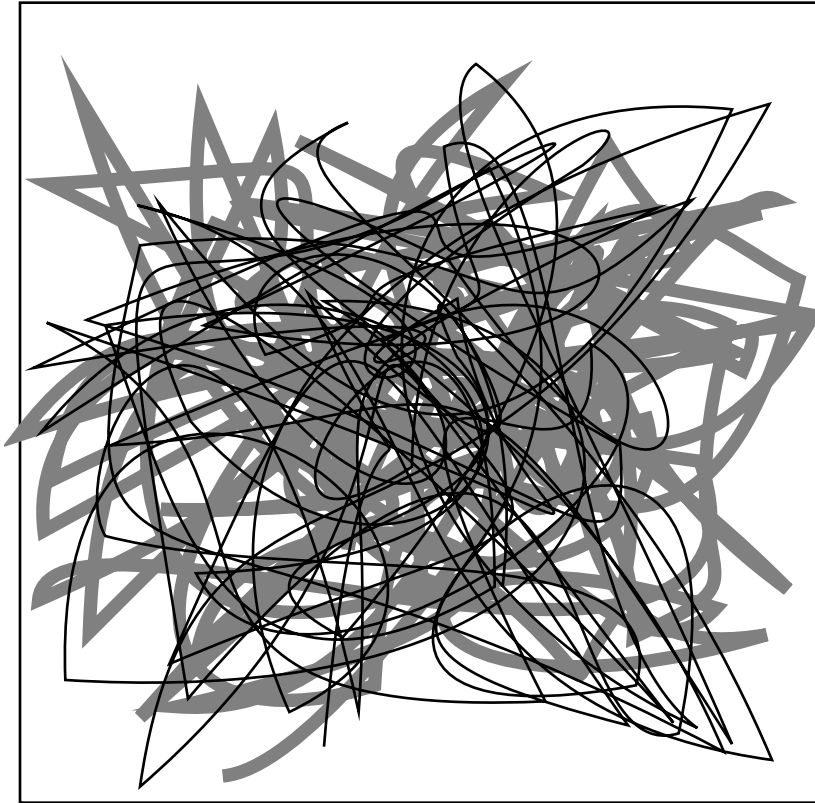


```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:excited1.eps
%%BoundingBox:0 0 300 300

0 0 300 300 rectstroke

/x {rand 300 mod} bind def

x x moveto
50 {x x x x x x curveto} repeat
stroke
```

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:excited2.eps
%%BoundingBox:0 0 300 300

0 0 300 300 rectstroke

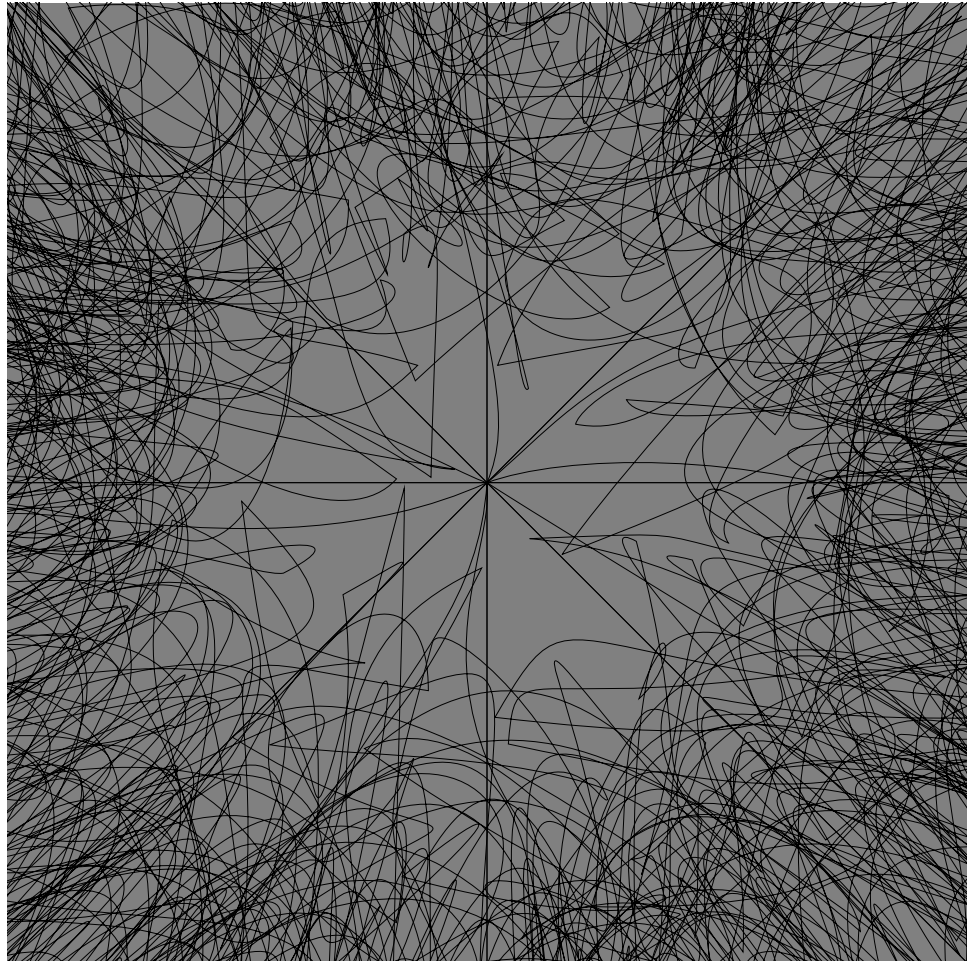
/x    {rand 300 mod} bind def

/excit{
    x x moveto
    50 {x x x x x x curveto} repeat
    stroke} def

5 setlinewidth
.5 setgray
excit

1 setlinewidth
0 setgray
excit

```



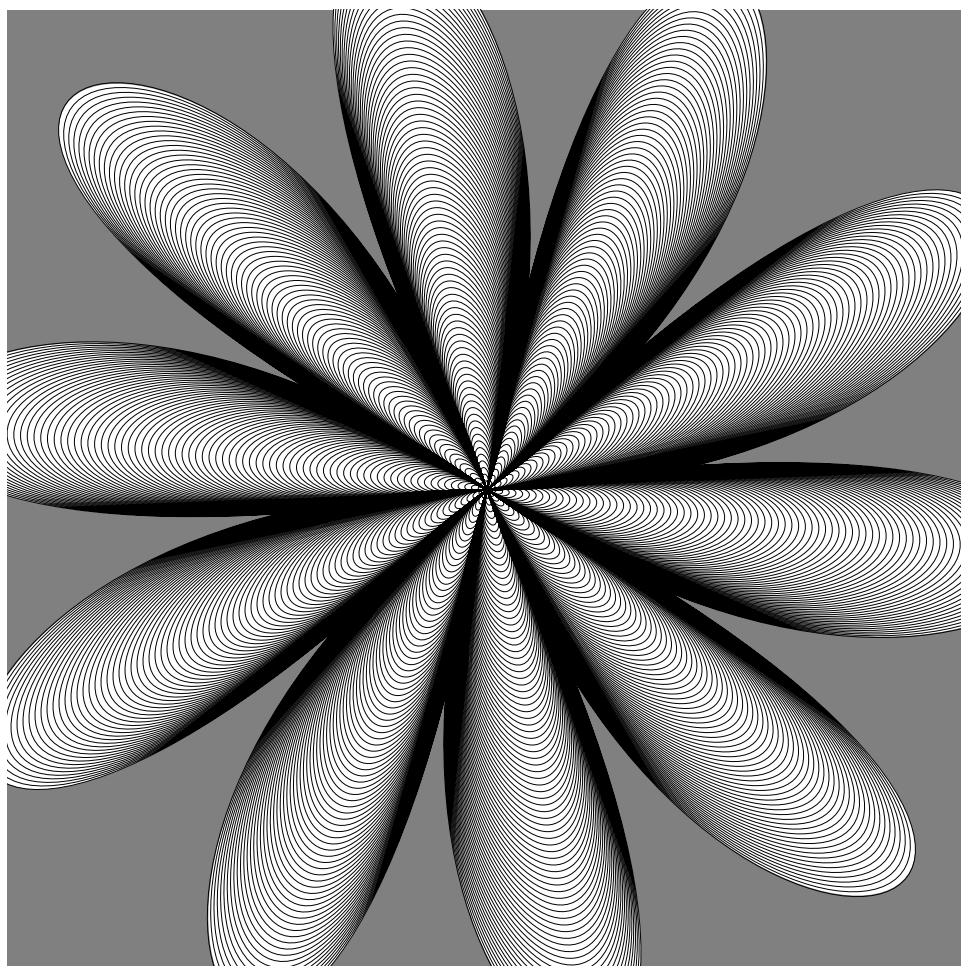
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:exploded.eps
%%BoundingBox:0 0 360 360

/a {50 rand exch mod} bind def
/b {100 rand exch mod} bind def
/c {200 rand exch mod} bind def
/d {300 rand exch mod} bind def
/e {400 rand exch mod} bind def
/f {500 rand exch mod} bind def

0 0 360 360 rectclip
.5 setgray 0 0 360 360 rectfill

.25 setlinewidth 0 setgray
180 180 translate

8 {45 rotate 0 0 moveto
  100 {a b c d e f curveto} repeat stroke
  0 0 600 600 rectstroke
} repeat
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:flower.eps
%%BoundingBox:0 0 360 360

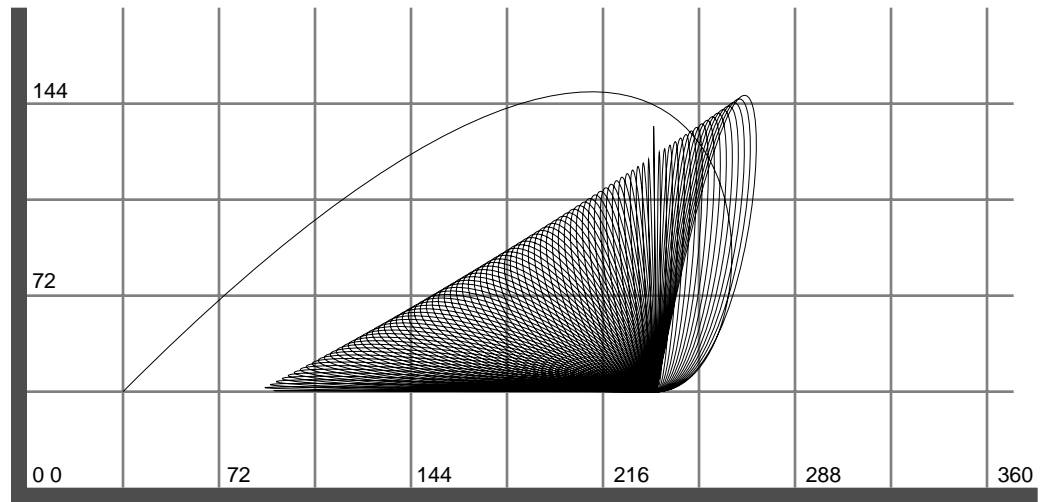
0 0 360 360 rectclip
.5 setgray
0 0 360 360 rectfill

180 180 translate

1 setgray
10 { 36 rotate
    0 0 moveto 252 253 253 0 0 0 curveto fill} repeat

0 setgray
.25 setlinewidth
10 { 36 rotate
    0 1 254 {} for
    0 0 moveto
    85 { 0 0 0 curveto} repeat stroke
    } repeat

```

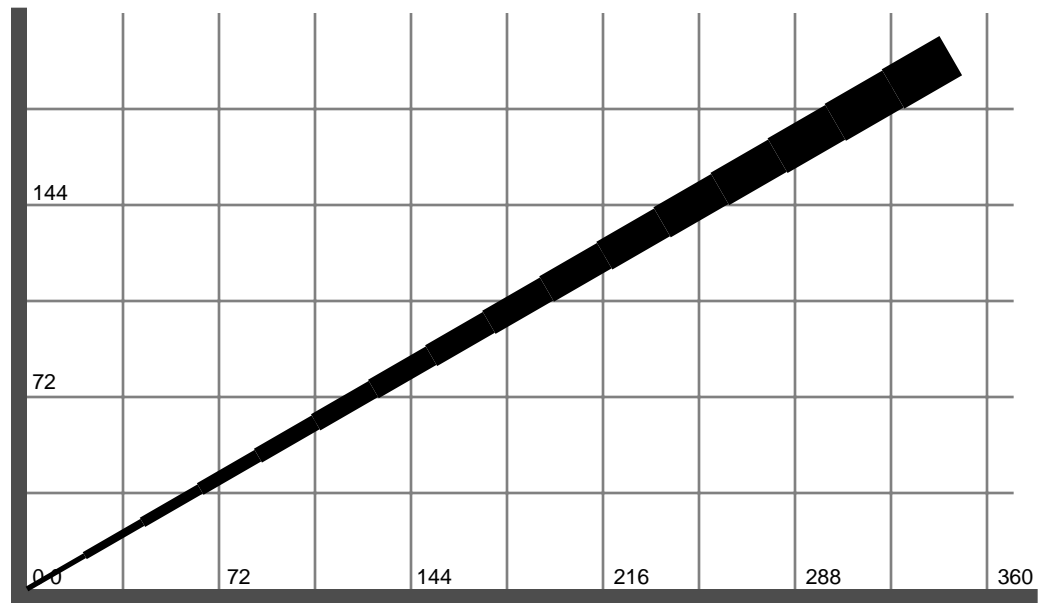


```
!PS-Adobe-2.0 EPSF-1.2
%%Title:petal2.eps
%%BoundingBox:0 0 385 186

36 36 translate

.25 setlinewidth

0 1 254 {} for
  0 0 moveto
  85 { 0 200 0 curveto} repeat stroke
```



```

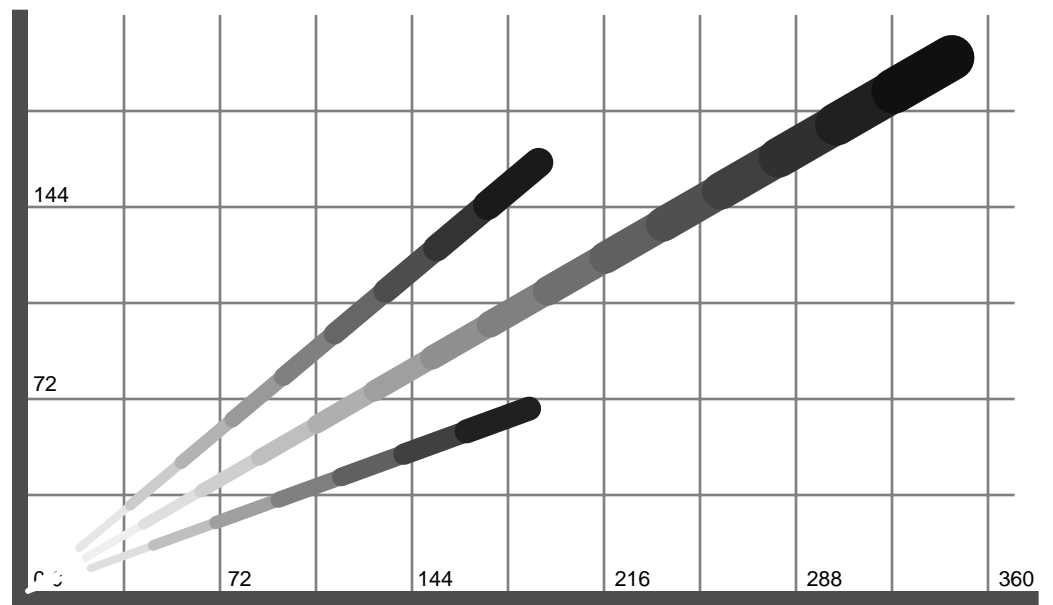
%!PS-Adobe-2.0 EPSF-1.2
%%Title:3D-Line1.eps
%%BoundingBox:0 0 385 224

/line {25 0 rlineto} def

30 rotate
1 setlinewidth

0 0 moveto
16 {currentlinewidth 1 add setlinewidth line
    currentpoint stroke moveto} repeat

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:3D-Line2.eps
%%BoundingBox:0 0 385 224

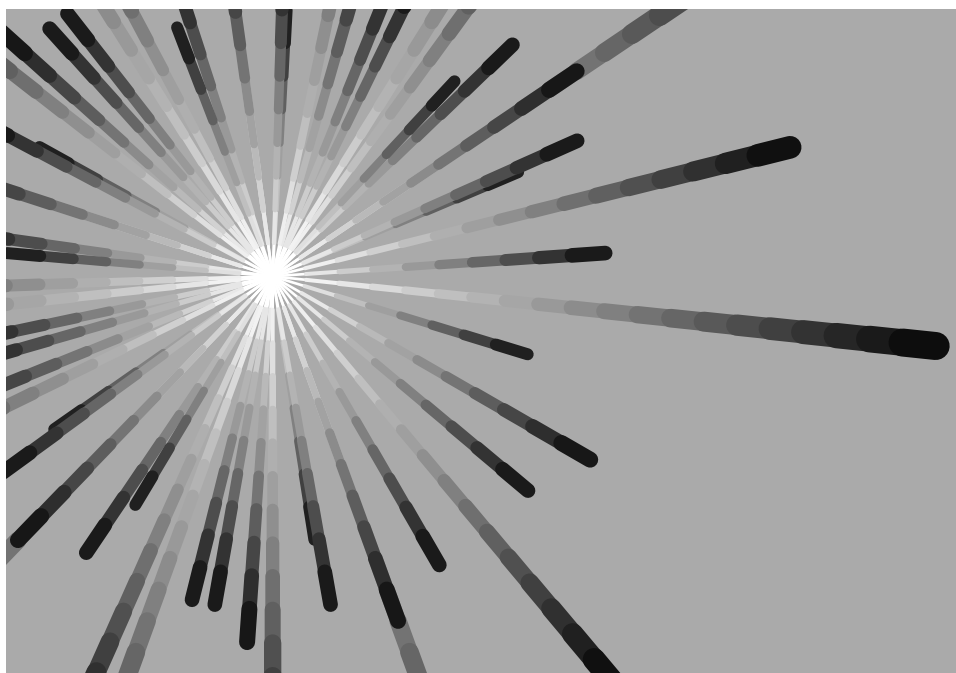
/line {25 0 rlineto} def

1 setlinewidth
1 setlinecap
1 setgray

0 0 moveto
/L {/length exch def
  gsave
  /sections {length 25 div cvi} def
  /v 1 sections div def
  sections {
    currentlinewidth 1 add setlinewidth line
    currentpoint stroke moveto
    currentgray v sub setgray} repeat
  grestore} def

20 rotate 200 L
10 rotate 400 L
10 rotate 250 L

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:3D-LineStar.eps
%%BoundingBox:0 0 360 250

.5 .5 scale
.666 setgray
0 0 720 500 rectclip
0 0 720 500 rectfill

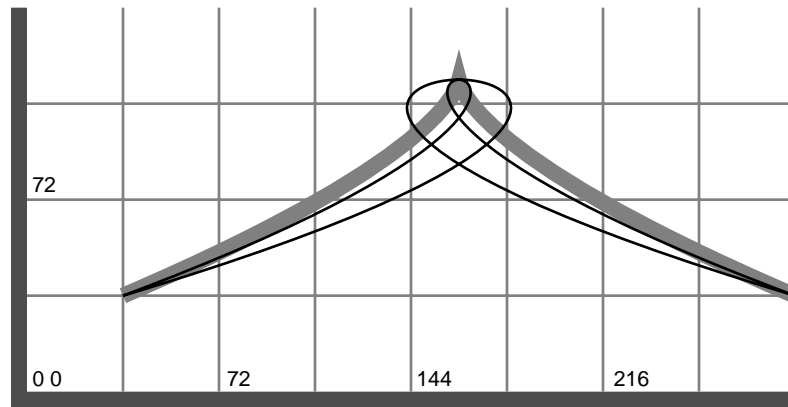
/line {25 0 rlineto} def

200 300 translate 10 rotate
1 setlinewidth 1 setlinecap 1 setgray

0 0 moveto
/L { /length exch def
      gsave
      /sections {length 25 div cvi} def
      /v 1 sections div def
      sections {currentlinewidth 1 add setlinewidth line
                currentpoint stroke moveto
                currentgray v sub setgray} repeat
      grestore} def

10 { 13 rotate 200 L
     11 rotate 500 L
     10 rotate 250 L
     10 rotate 400 L
     10 rotate 250 L
     10 rotate 275 L} repeat

```

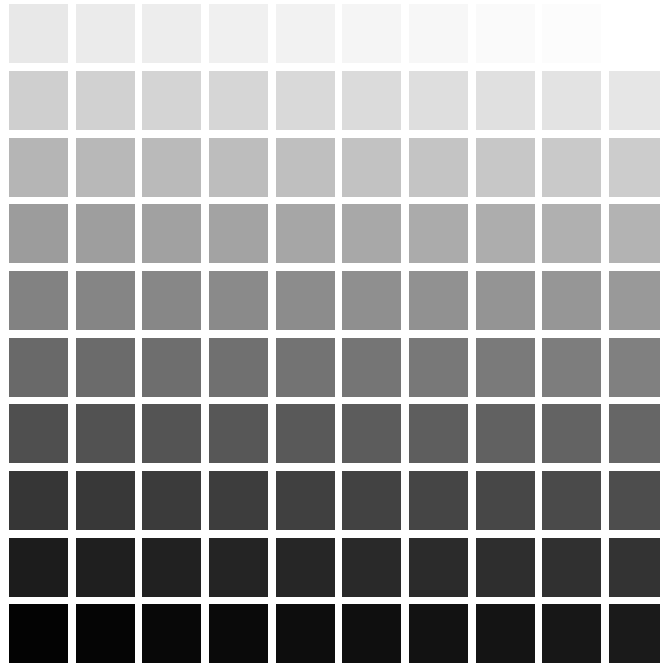


```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:curvetoLoops.eps
%%BoundingBox:34 34 290 120

.5 setgray
6 setlinewidth
36 36 moveto
288 144 36 144 288 36 curveto
stroke

0 setgray
1 setlinewidth
36 36 moveto
324 144 0 144 288 36 curveto
stroke

36 36 moveto
396 144 -72 144 288 36 curveto
stroke
```

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:grayChart.eps
%%BoundingBox:25 25 275 275

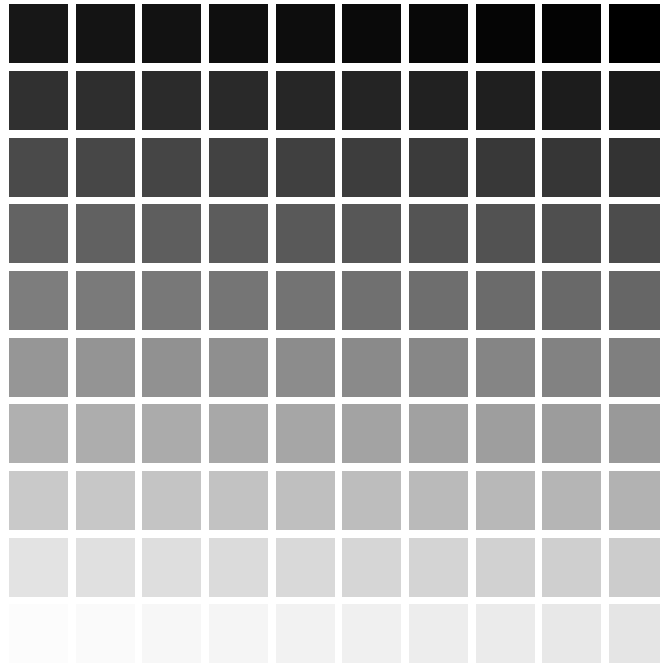
/box {0 0 moveto 22 0 rlineto 0 22 rlineto -22 0 rlineto
closepath fill} def

/boxx
  {10 {25 0 translate currentgray .01 add setgray box}
repeat} def

0 25 translate
boxx

9 {-250 25 translate boxx} repeat

```



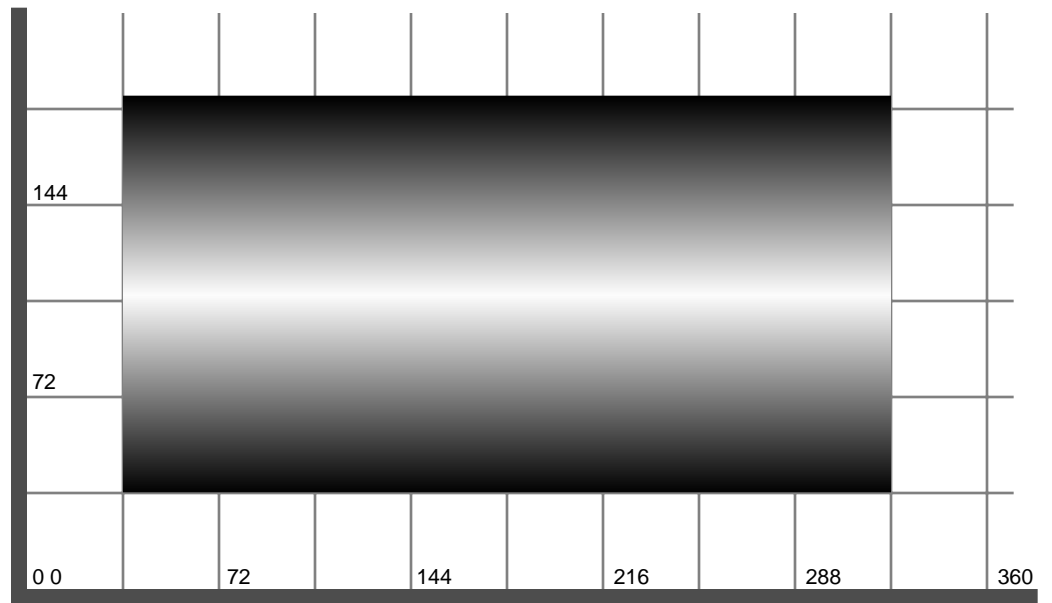
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:grayChartRev.eps
%%BoundingBox:25 25 275 275

/box {0 0 moveto 22 0 rlineto 0 22 rlineto -22 0 rlineto
      closepath fill} def

/boxx
  {10 {25 0 translate currentgray .01 add setgray box}
  repeat} def

{1 exch sub} settransfer
0 25 translate
boxx

9 {-250 25 translate boxx} repeat
```



```

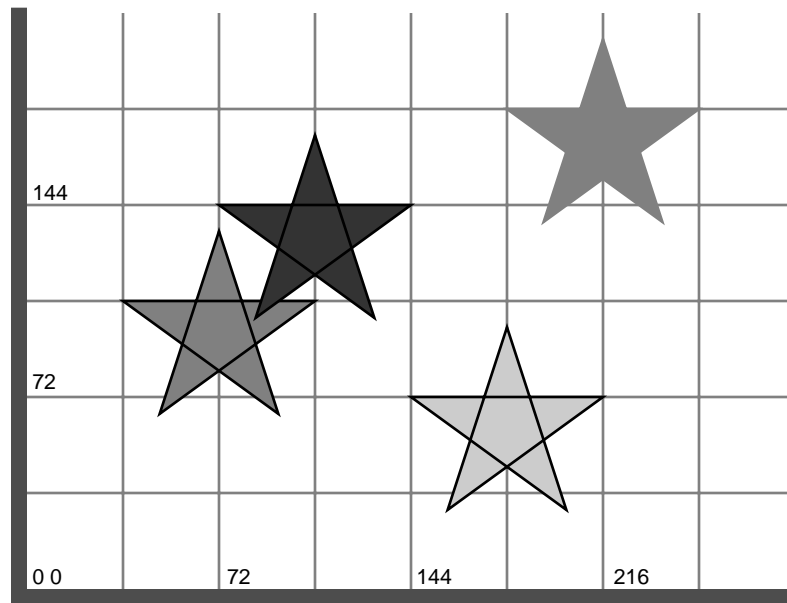
%!PS-Adobe-2.0 EPSF-1.2
%%Title:fountainLine.eps
%%BoundingBox:36 36 324 185

/line {0 0 moveto 288 0 lineto stroke} def

36 36 translate

99 {0 .75 translate currentgray .01 add setgray line} repeat
99 {0 .75 translate currentgray .01 sub setgray line} repeat

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:star2.eps
%%BoundingBox:36 30 252 210

/starside {72 0 lineto currentpoint translate
-144 rotate} def

/star
{moveto
currentpoint translate
4 {starside} repeat closepath
gsave
    setgray fill
grestore
stroke} def

gsave
    .5 36 108 star
grestore

gsave
    .2 72 144 star
grestore

gsave
    .8 144 72 star
grestore

gsave
    .5 setgray
    .5 180 180 star
grestore
    
```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:bas-relief.eps
%%BoundingBox:0 0 360 130

0 0 360 130 rectstroke
/name { 0 0 moveto (bas-relief) show} def

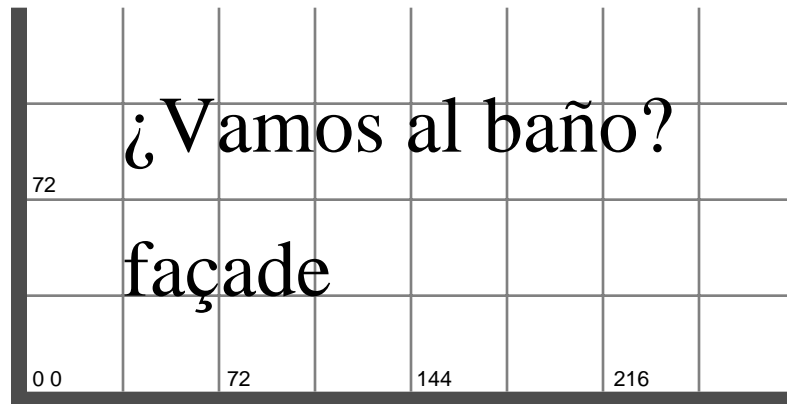
/Helvetica-Bold findfont 72 scalefont setfont

36 36 translate

0 .05 .95 {setgray name -1 .5 translate} for

0 setgray name

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:unencoding1.eps
%%BoundingBox:36 30 240 110

/BaseFont /Times-Roman findfont def

/newfont BaseFont length dict def
newfont begin
BaseFont
    {1 index dup /FID ne exch
    /FID ne exch
    /Encoding ne and
    {def}{pop pop} ifelse
    }bind forall

/Encoding 256 array def
StandardEncoding Encoding copy pop

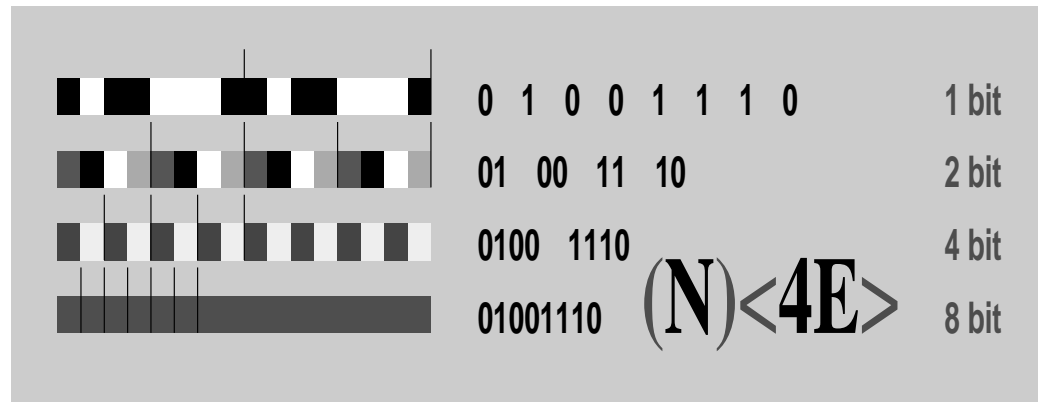
Encoding 1 /ntilde put
Encoding 2 /ccedilla put

end

/Times-RomanAccent newfont definefont pop
/Times-RomanAccent findfont 30 scalefont setfont

36 90 moveto
(\277Vamos al ba\1o?) show
36 36 moveto
(fa\2ade) show

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:unencoding2.eps
%%BoundingBox:36 36 360 110

/BaseFont /Palatino-Roman findfont def

/newfont BaseFont length dict def
newfont begin
BaseFont
  {1 index dup /FID ne exch
  /FID ne exch
  /Encoding ne and
  {def}{pop pop} ifelse
  }bind forall

/Encoding 256 array def
StandardEncoding Encoding copy pop

Encoding 1 /copyright put
Encoding 2 /registered put
Encoding 3 /trademark put

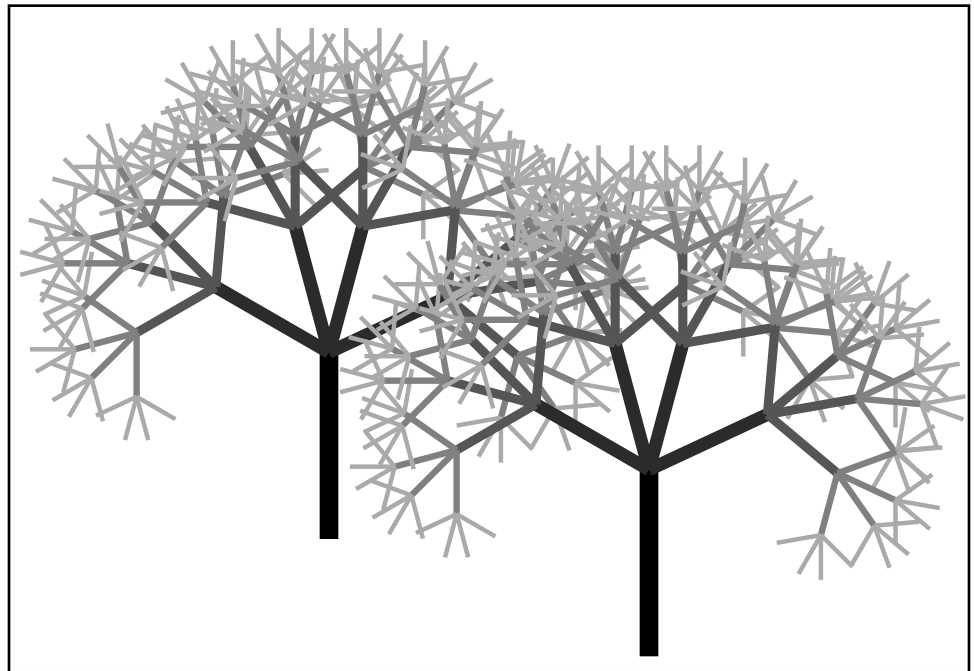
end

/Times-RomanAccent newfont definefont pop

/Times-RomanAccent findfont 30 scalefont setfont

36 36 moveto
(\0011989 Adobe) show
36 90 moveto
(PostScript\2 Language\3) show

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:branch2.eps
%%BoundingBox: 0 0 360 250

/depth 0 def
/maxd 4 def
/down {/depth depth 1 add def} bind def
/up {/depth depth 1 sub def} bind def

/line % vertical line
{0 100 rlineto currentpoint
stroke translate 0 0 moveto} bind def

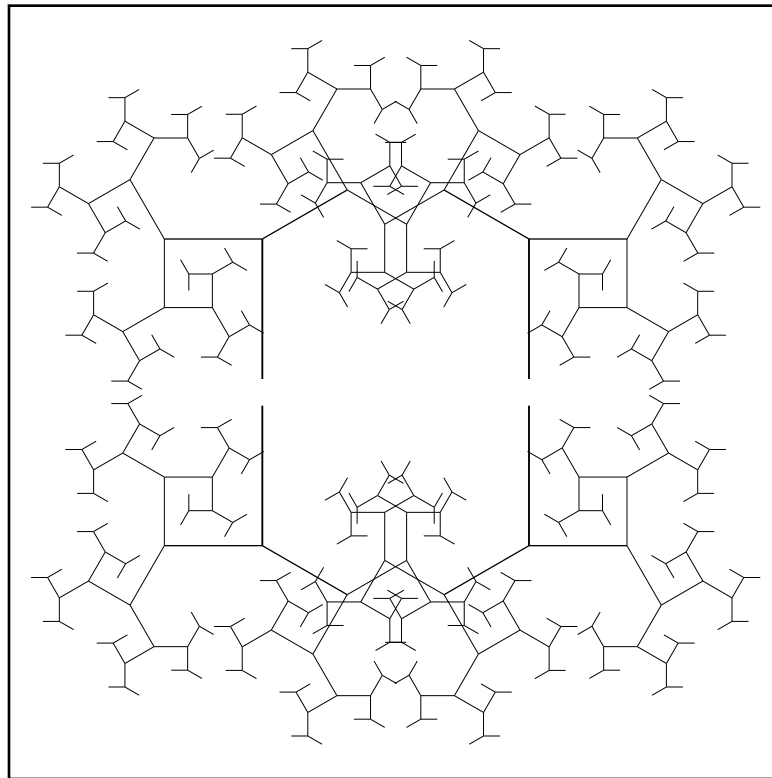
/branch {gsave .7 .7 scale
down line
depth maxd 2 add div setgray
depth maxd le
{60 rotate branch
-45 rotate branch
-30 rotate branch
-20 rotate
-30 rotate branch} if
up grestore} bind def

0 0 360 250 rectstroke
10 setlinewidth

120 100 moveto branch stroke

240 6 moveto branch stroke

```

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:branch3.eps
%%BoundingBox: 0 0 290 290

/depth 0 def
/maxd 6 def
/down {/depth depth 1 add def} bind def
/up {/depth depth 1 sub def} bind def

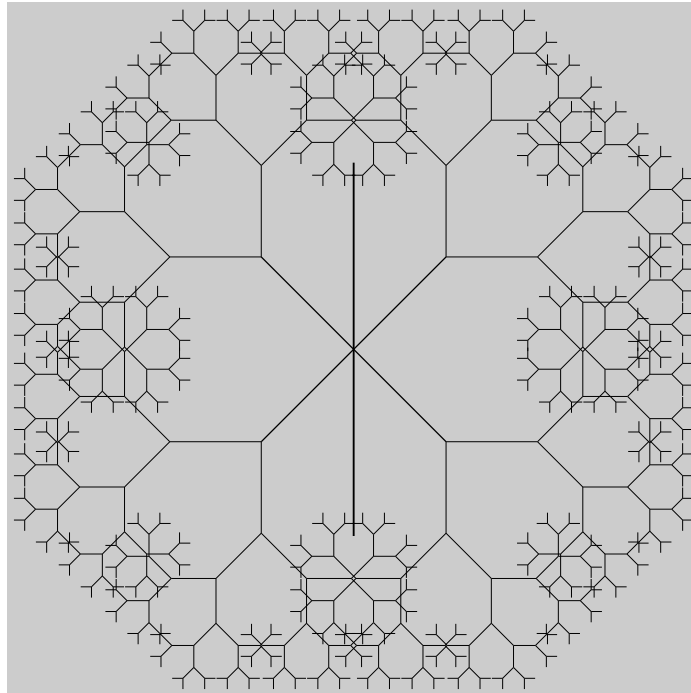
/line {0 75 rlineto currentpoint
stroke translate 0 0 moveto} bind def

/branch {gsave .7 .7 scale
down line depth maxd le
{60 rotate branch -150 rotate branch} if
up grestore} bind def

0 0 290 290 rectstroke
-25 145 translate

220 5 moveto branch stroke
[-1 0 0 1 0 0] concat
-120 5 moveto branch stroke
[1 0 0 -1 0 0] concat
-120 5 moveto branch stroke
[-1 0 0 1 0 0] concat
220 5 moveto branch stroke

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:branch4.eps
%%BoundingBox: 0 0 260 260

/depth 0 def
/maxd 8 def
/down {/depth depth 1 add def} bind def
/up    {/depth depth 1 sub def} bind def

/line % vertical line
      {0 100 rlineto currentpoint
       stroke translate 0 0 moveto} bind def

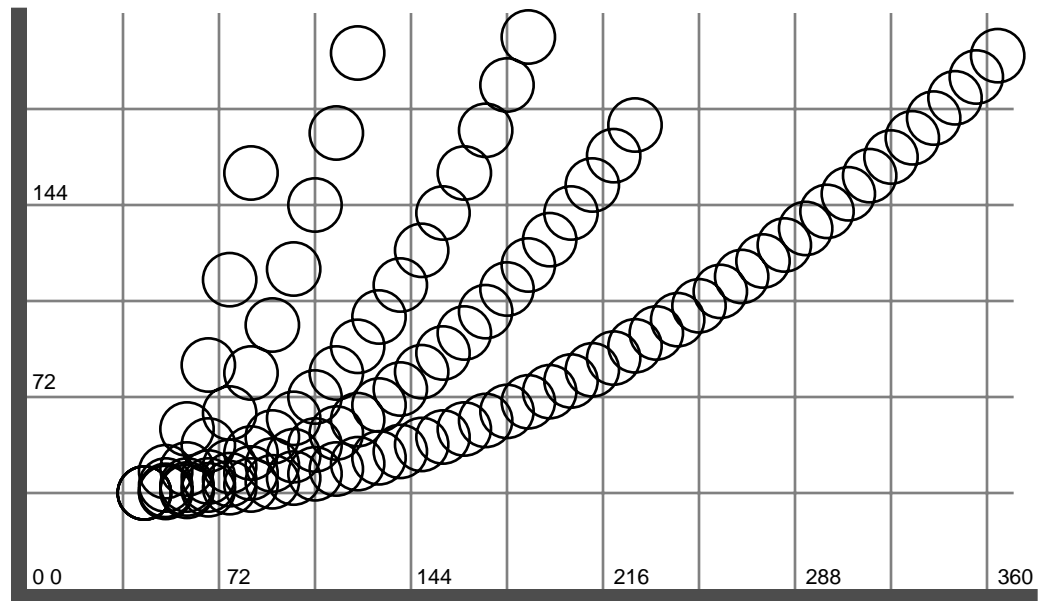
/branch{gsave .7 .7 scale
         down line
         depth maxd le
           {45 rotate branch
            -90 rotate branch} if
         up grestore} bind def

.8 setgray
0 0 260 260 rectfill

0 setgray
130 60 moveto branch

130 60 translate
180 rotate
0 -130 moveto branch stroke

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:slinky.eps
%%BoundingBox:34 24 378 220

36 36 translate

/slope .2 def
/circle {10 0 moveto 0 0 10 0 360 arc stroke} def

gsave
  0 1 40 {8 exch slope mul translate circle} for
grestore

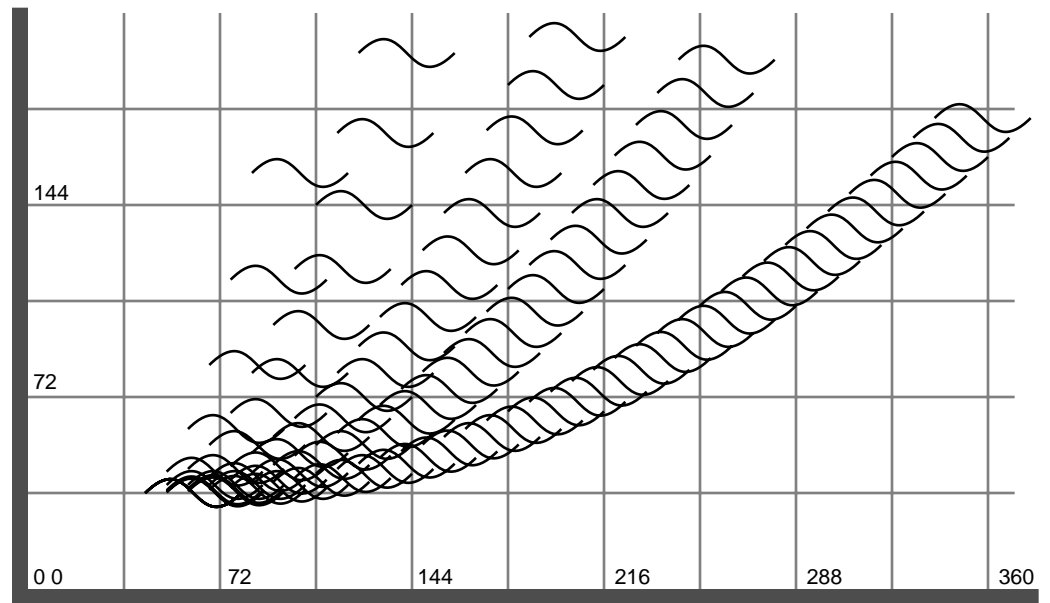
/slope .5 def
gsave
  0 1 23 {8 exch slope mul translate circle} for
grestore

/slope 1 def
gsave
  0 1 18 {8 exch slope mul translate circle} for
grestore

/slope 3 def
gsave
  0 1 10 {8 exch slope mul translate circle} for
grestore

/slope 8 def
gsave
  0 1 5 {8 exch slope mul translate circle} for
grestore

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:slinky2.eps
%%BoundingBox:38 30 378 216

36 36 translate

/slope .2 def
/circle {0 0 moveto 18 18 18 -18 36 0 curveto stroke} def

gsave
  0 1 37 {8 exch slope mul translate circle} for
grestore

/slope .5 def
gsave
  0 1 25 {8 exch slope mul translate circle} for
grestore

/slope 1 def
gsave
  0 1 18 {8 exch slope mul translate circle} for
grestore

/slope 3 def
gsave
  0 1 10 {8 exch slope mul translate circle} for
grestore

/slope 8 def
gsave
  0 1 5 {8 exch slope mul translate circle} for
grestore

```



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:TCpos&rev.eps
%%BoundingBox: 0 0 288 144

gsave
0 setgray
/rows 72 def
/cols 72 def
/picstr1 72 string def
/readdata {currentfile exch readhexstring pop} def
/beginimage {{picstr1 readdata} image} def
72 65536 mul 2359296 div dup cols mul exch rows mul scale
cols rows 8 [cols 0 0 rows neg 0 rows]
beginimage
191F1F1E1F202429

... picture data ...

8147519C9A836D55
grestore

gsave
144 0 translate
{1 exch sub} settransfer
beginimage
191F1F1E1F202429

... picture data ...

8147519C9A836D55
grestore end

```

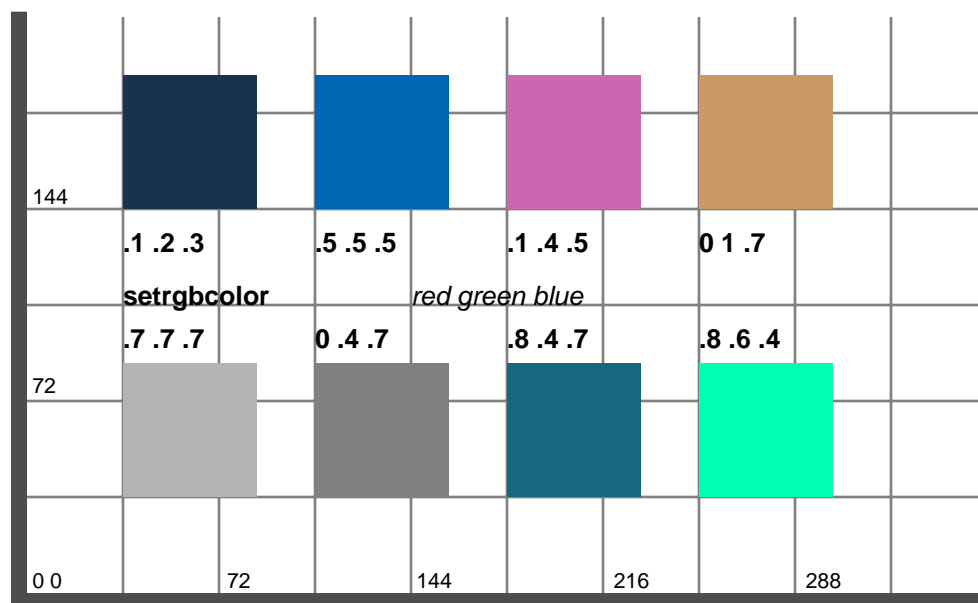




library of color examples

Some of these programs will work on a black and white printer. Some of the operators used are part of PostScript Level 2, which means you'll have to try them on a color laser printer. The files `rgbBoxes.eps` and `hsbBoxes.eps` will work on an Apple LaserWriter, the file `cmYkBoxes.eps` will not.

`cPattern1.eps` 291
`cfountain_1.eps` 295
`c-imageWord1.eps` 289
`c-imageWord2.eps` 290
`cmYkBoxes.eps` 288
`colorRNotes.eps` 292
`cRLines1.eps` 293
`cRLines2.eps` 294
`hsbBoxes.eps` 287
`rgbBoxes.eps` 286



20-1

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:rgbBoxes.eps
%%BoundingBox:36 36 306 198

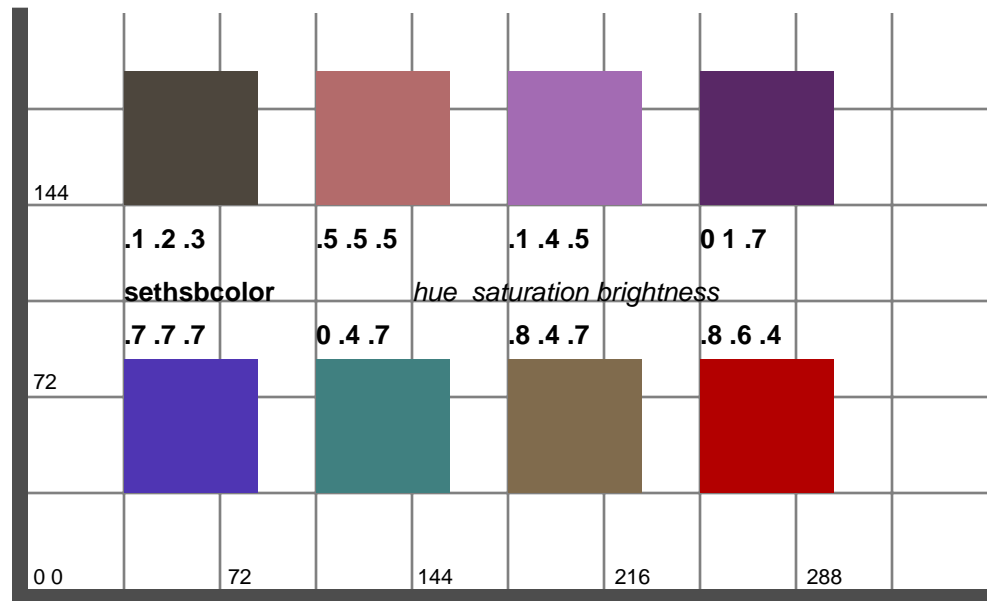
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
closepath fill} def
/black {0 setgray} def

/Helvetica-Oblique findfont 10 scalefont setfont
144 108 moveto (red green blue) show

/Helvetica-Bold findfont 10 scalefont setfont
36 108 moveto (setrgbcolor) show

36 128 moveto (.1 .2 .3 ) show
    .1 .2 .3 setrgbcolor 36 144 box
black 108 92 moveto (0 .4 .7) show
    0 .4 .7 setrgbcolor 108 144 box
black 180 92 moveto (.8 .4 .7) show
    .8 .4 .7 setrgbcolor 180 144 box
black 252 92 moveto (.8 .6 .4) show
    .8 .6 .4 setrgbcolor 252 144 box

black 36 92 moveto (.7 .7 .7) show
    .7 .7 .7 setrgbcolor 36 36 box
black 108 128 moveto (.5 .5 .5) show
    .5 .5 .5 setrgbcolor 108 36 box
black 180 128 moveto (.1 .4 .5) show
    .1 .4 .5 setrgbcolor 180 36 box
black 252 128 moveto (0 1 .7) show
    0 1 .7 setrgbcolor 252 36 box
    
```

20-2

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:hsbBoxes.eps
%%BoundingBox:36 36 306 198

/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
closepath fill} def
/black {0 setgray} def

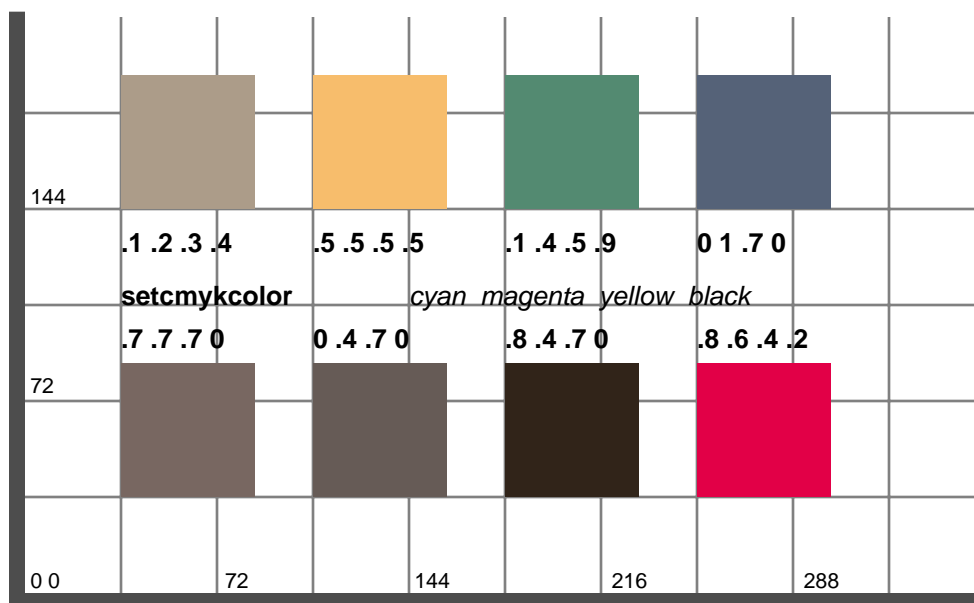
/Helvetica-Oblique findfont 10 scalefont setfont
144 108 moveto (hue saturation brightness) show

/Helvetica-Bold findfont 10 scalefont setfont
36 108 moveto (sethsbcolor) show

36 128 moveto (.1 .2 .3 ) show
    .1 .2 .3 sethsbcolor 36 144 box
black 108 92 moveto (0 .4 .7) show
    0 .4 .7 sethsbcolor 108 144 box
black 180 92 moveto (.8 .4 .7) show
    .8 .4 .7 sethsbcolor 180 144 box
black 252 92 moveto (.8 .6 .4) show
    .8 .6 .4 sethsbcolor 252 144 box

black 36 92 moveto (.7 .7 .7) show
    .7 .7 .7 sethsbcolor 36 36 box
black 108 128 moveto (.5 .5 .5) show
    .5 .5 .5 sethsbcolor 108 36 box
black 180 128 moveto (.1 .4 .5) show
    .1 .4 .5 sethsbcolor 180 36 box
black 252 128 moveto (0 1 .7) show
    0 1 .7 sethsbcolor 252 36 box

```



20-3

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:cmykBoxes.eps
%%BoundingBox:36 36 306 198

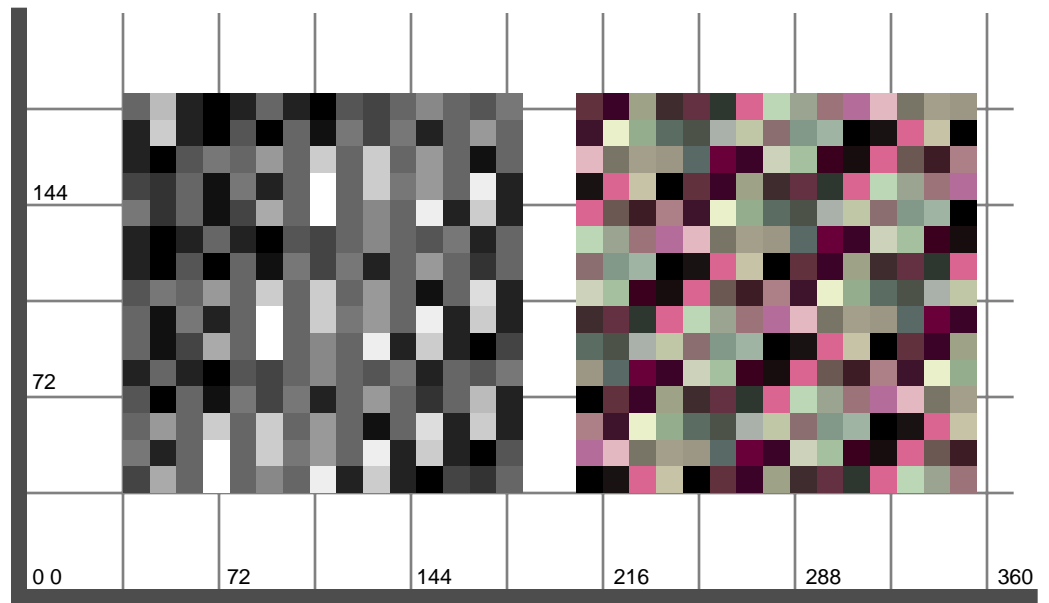
/box {moveto 50 0 rlineto 0 50 rlineto -50 0 rlineto
      closepath fill} def
/black {0 setgray} def

/Helvetica-Oblique findfont 10 scalefont setfont
144 108 moveto (cyan magenta yellow black) show

/Helvetica-Bold findfont 10 scalefont setfont
36 108 moveto (setcmykcolor) show

36 128 moveto (.1 .2 .3 .4) show
      .1 .2 .3 .4 setcmykcolor 36 144 box
black 108 92 moveto (0 .4 .7 0) show
      0 .4 .7 0 setcmykcolor 108 144 box
black 180 92 moveto (.8 .4 .7 0) show
      .8 .4 .7 0 setcmykcolor 180 144 box
black 252 92 moveto (.8 .6 .4 .2) show
      .8 .6 .4 .2 setcmykcolor 252 144 box

black 36 92 moveto (.7 .7 .7 0) show
      .7 .7 .7 0 setcmykcolor 36 36 box
black 108 128 moveto (.5 .5 .5 .5) show
      .5 .5 .5 .5 setcmykcolor 108 36 box
black 180 128 moveto (.1 .4 .5 .9) show
      .1 .4 .5 .9 setcmykcolor 180 36 box
black 252 128 moveto (0 1 .7 0) show
      0 1 .7 0 setcmykcolor 252 36 box
    
```



learn

20-4

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:c-imageWord1.eps
%%Creator:John F Sherman
%%CreationDate:Dec 1990
%%BoundingBox:36 36 356 186

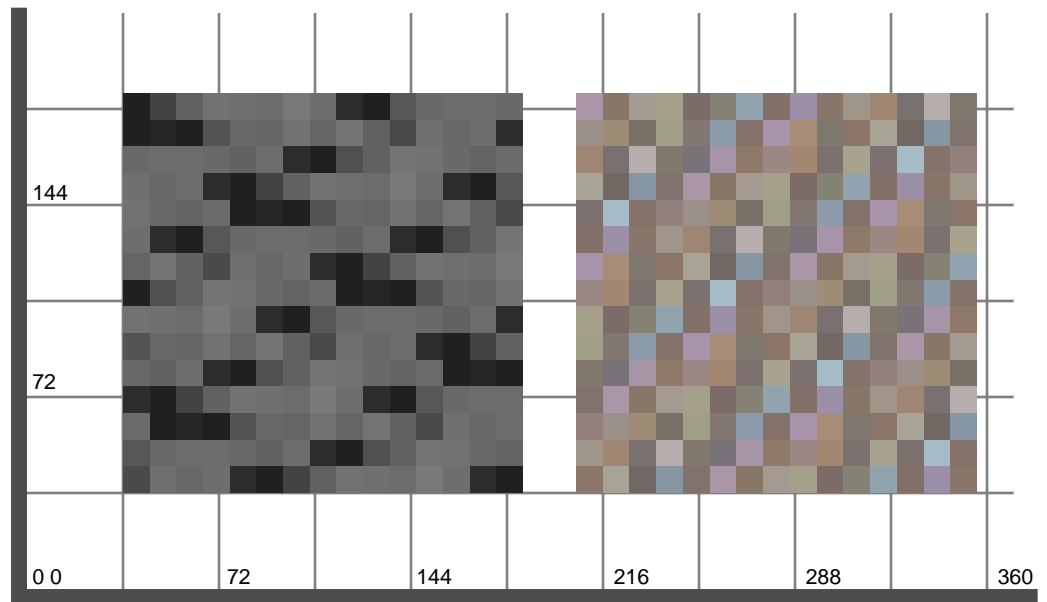
/word (John, Carolyn, William, Patrick & Theresa) def

36 36 translate
15 15 4 [.1 0 0 .1 0 0] {word} true 1 colorimage

170 0 translate
15 15 4 [.1 0 0 .1 0 0] {word} false 4 colorimage

```

The difference between this PostScript example and the example on the previous page is that these two squares are 8-bit patterns and the previous squares are 4-bit.



20-5

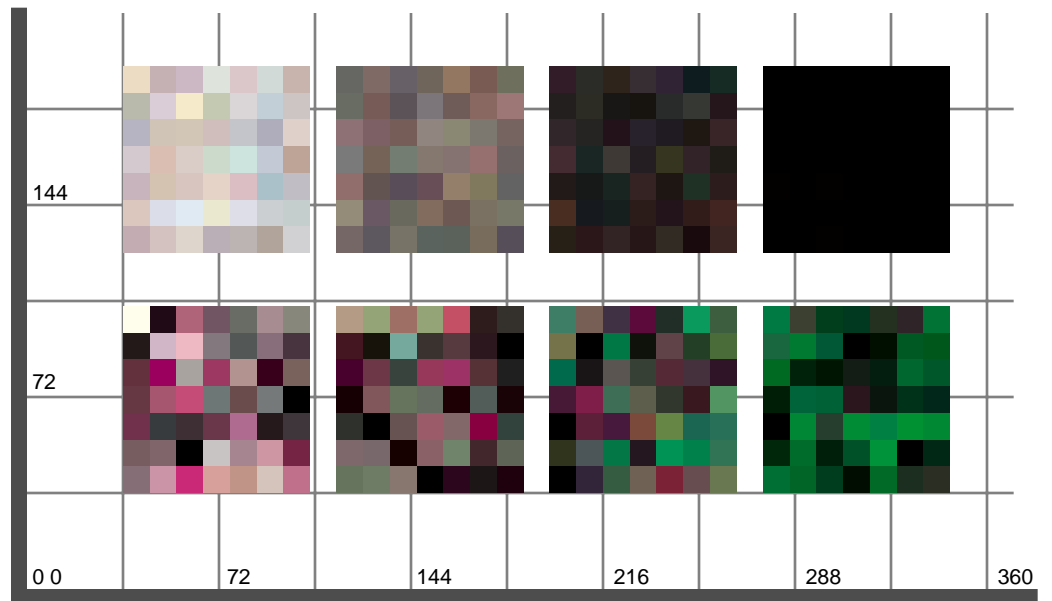
```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:c-imageWord2.eps
%%Creator:John F Sherman
%%CreationDate:Dec 1990
%%BoundingBox:36 36 356 186

/word (John, Carolyn, William, Patrick & Theresa) def

36 36 translate
15 15 8 [.1 0 0 .1 0 0] {word} true 1 colorimage

170 0 translate
15 15 8 [.1 0 0 .1 0 0] {word} false 4 colorimage
    
```



20-6

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:cPattern1.eps
%%BoundingBox:36 36 350 196

/str 512 string def
/bit 4 def
/cPattern {
  /a exch def /b exch def
  /diff a b sub def
  7 7 bit [.1 0 0 .1 0 0]
  {0 1 511 {str exch rand diff mod b add put}
  for str} false 4 colorimage} def

222 srand
36 36 translate 0 75 cPattern

80 0 translate 75 150 cPattern

80 0 translate 150 225 cPattern

80 0 translate 225 250 cPattern

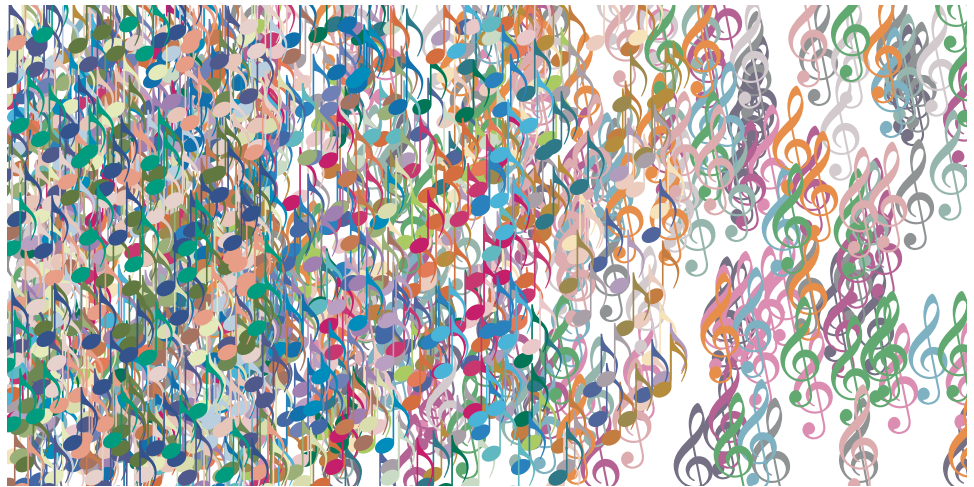
/bit 8 def
-240 90 translate 0 75 cPattern

80 0 translate 75 150 cPattern

80 0 translate 150 225 cPattern

80 0 translate 225 250 cPattern

```



20-7

```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:colorRNotes.eps
%%BoundingBox:0 0 360 180

/Sonata findfont 24 scalefont setfont

/n {rand exch mod 6 sub} def
/g {rand 100 mod .01 mul} def
/clipBox
  {/ury exch def /urx exch def
   /lly exch def /llx exch def

   llx lly moveto urx lly lineto urx ury lineto
   llx ury lineto closepath clip} def

173417 srand

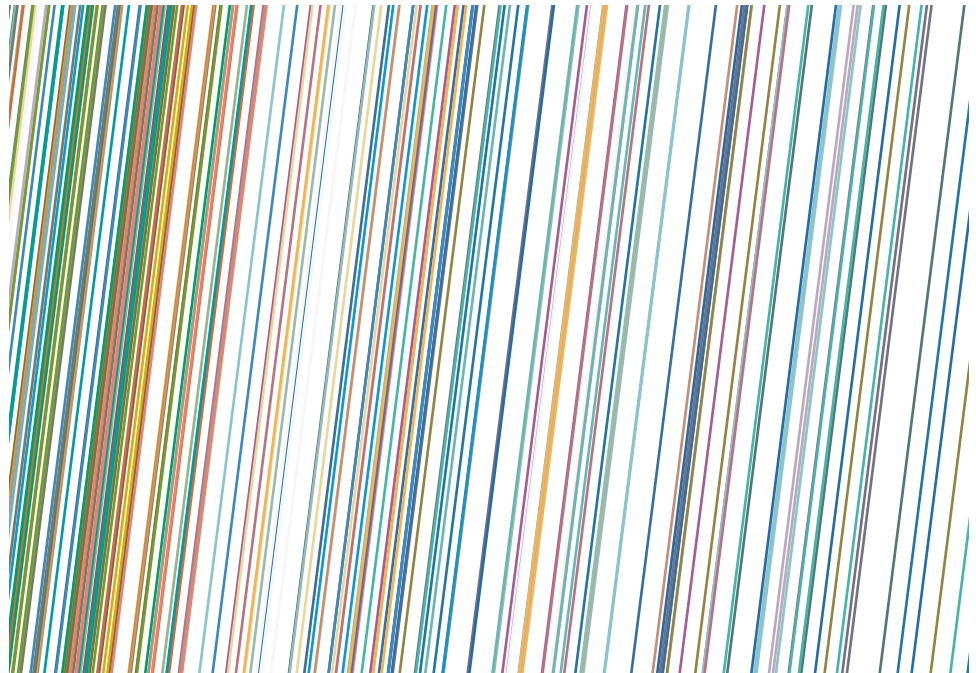
gsave
  0 0 360 180 clipBox

  /a {360 n 180 n moveto (&) show} def
  /b {250 n 180 n moveto (e) show} def
  /c {200 n 180 n moveto (e) show} def
  /d {150 n 180 n moveto (e) show} def
  /e {100 n 180 n moveto (e) show} def

  10 {g g g 0 setcmykcolor 25 {a} repeat} repeat
  10 {g g g 0 setcmykcolor 25 {b} repeat} repeat
  10 {g g g 0 setcmykcolor 25 {c} repeat} repeat
  10 {g g g 0 setcmykcolor 25 {d} repeat} repeat
  10 {g g g 0 setcmykcolor 25 {e} repeat} repeat
grestore
  
```



20-8



```

%!PS-Adobe-2.0 EPSF-1.2
%%Title:cRLines1.eps
%%Creator:John F Sherman
%%CreationDate:Dec 1990
%%BoundingBox:36 36 396 286

/n {rand exch mod} def
/g {rand 100 mod .01 mul} bind def

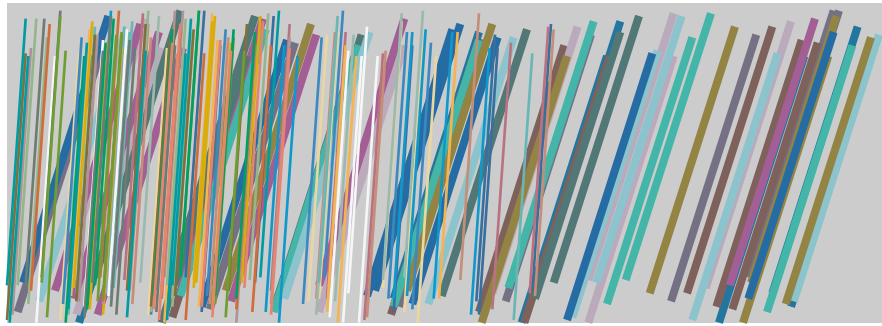
173417 srand
36 36 360 250 rectclip
gsave
  /a {400 n 36 n moveto 38 300 rlineto stroke} def
  /b {300 n 36 n moveto 38 300 rlineto stroke} def
  /c {100 n 36 n moveto 38 300 rlineto stroke} def

  10 {g g g 0 setcmykcolor 7 {a} repeat} repeat
  10 {g g g 0 setcmykcolor 7 {b} repeat} repeat
  10 {g g g 0 setcmykcolor 7 {c} repeat} repeat
grestore

```



20-9



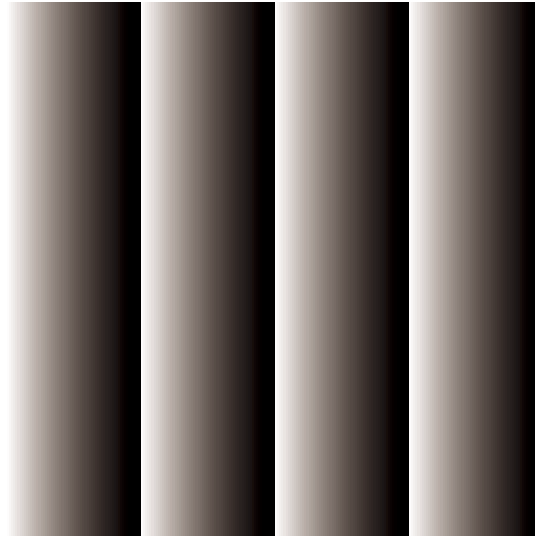
```
%!PS-Adobe-2.0 EPSF-1.2
%%Title:cRLines2.eps
%%BoundingBox:0 0 330 120

/n {rand exch mod} bind def
/g {rand 100 mod .01 mul} bind def

173417 srand
.8 setgray
0 0 330 120 rectfill

gsave
/a {300 n 18 n moveto 32 100 rlineto stroke} def
/b {200 n 18 n moveto 7 100 rlineto stroke} def
/c {100 n 18 n moveto 7 100 rlineto stroke} def

3 setlinewidth
10 {g g g 0 setcmykcolor 7 {a} repeat} repeat
1 setlinewidth
10 {g g g 0 setcmykcolor 7 {b} repeat} repeat
10 {g g g 0 setcmykcolor 7 {c} repeat} repeat
grestore
```

learn

20-10

```

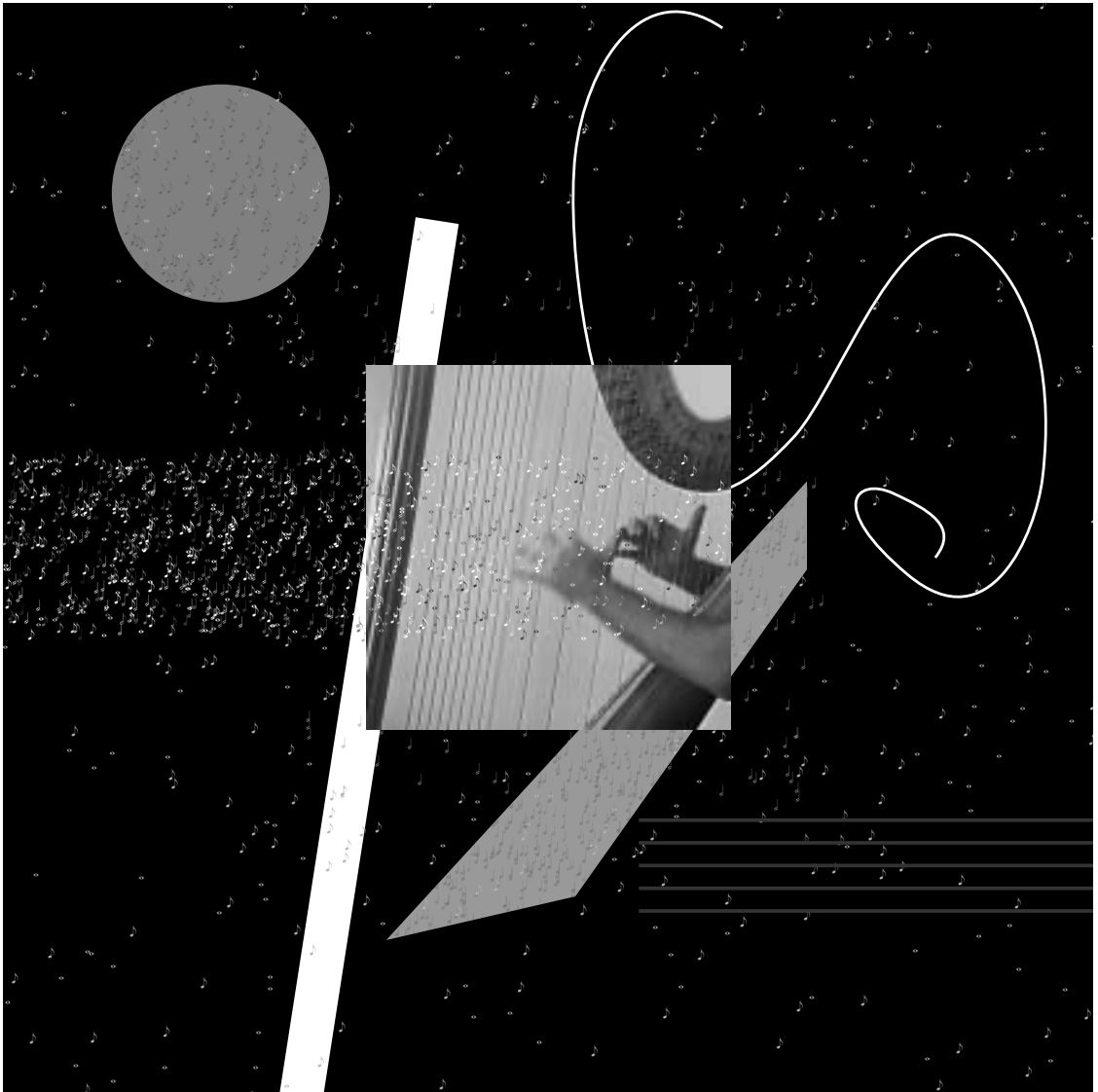
%!PS-Adobe-2.0 EPSF-1.2
%%Title:cfountain_1.eps
%%Creator:John F Sherman
%%BoundingBox:0 0 200 200

/fountstring 256 string def
0 1 255 { fountstring exch dup put } for

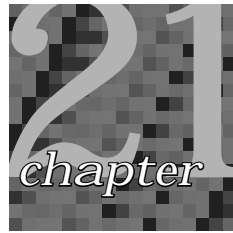
200 200 scale

255 1 8 [ 255 0 0 1 0 0 ] { fountstring } false 4 colorimage

```

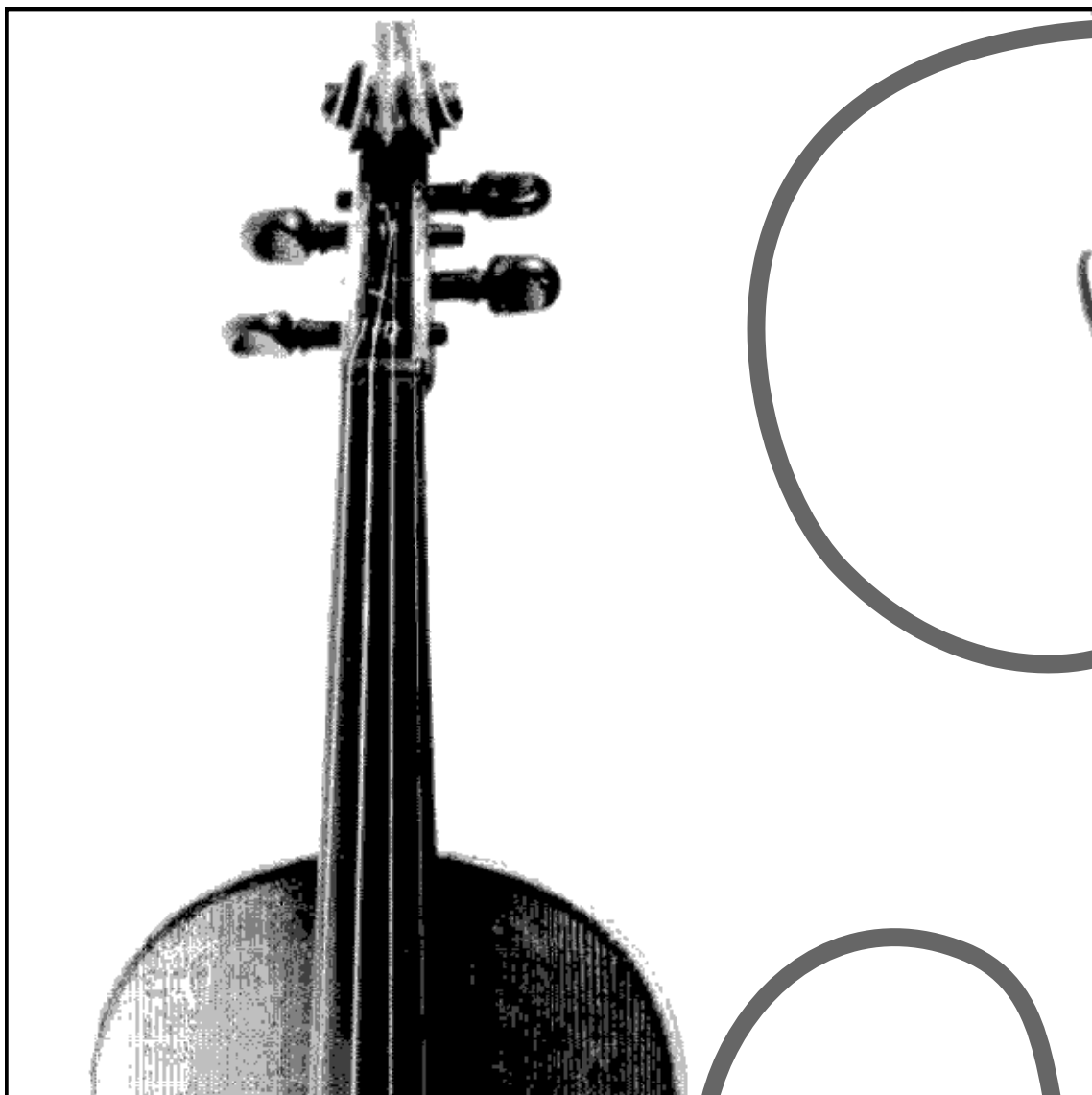


Mike's Romance

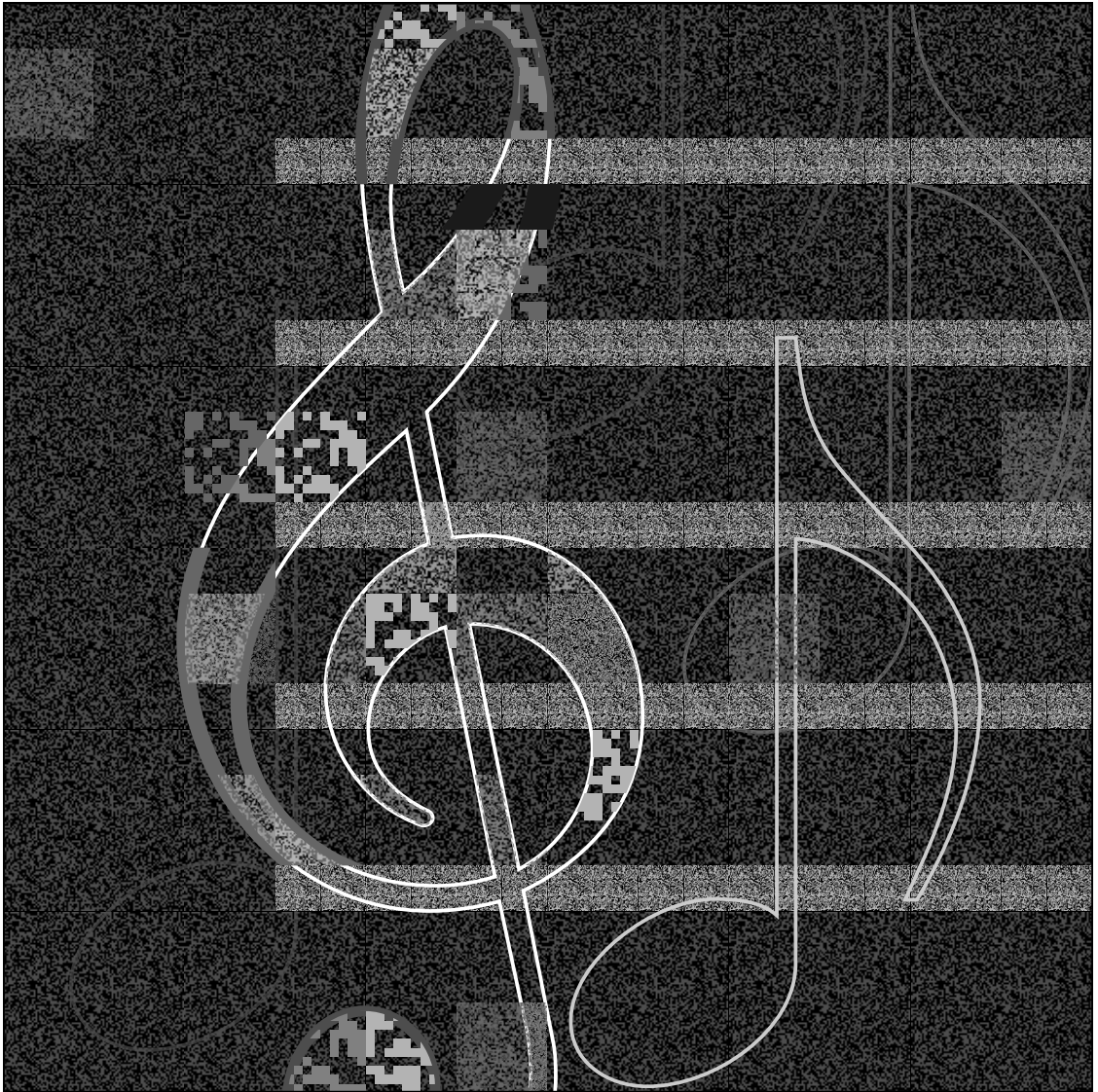


gallery of designs

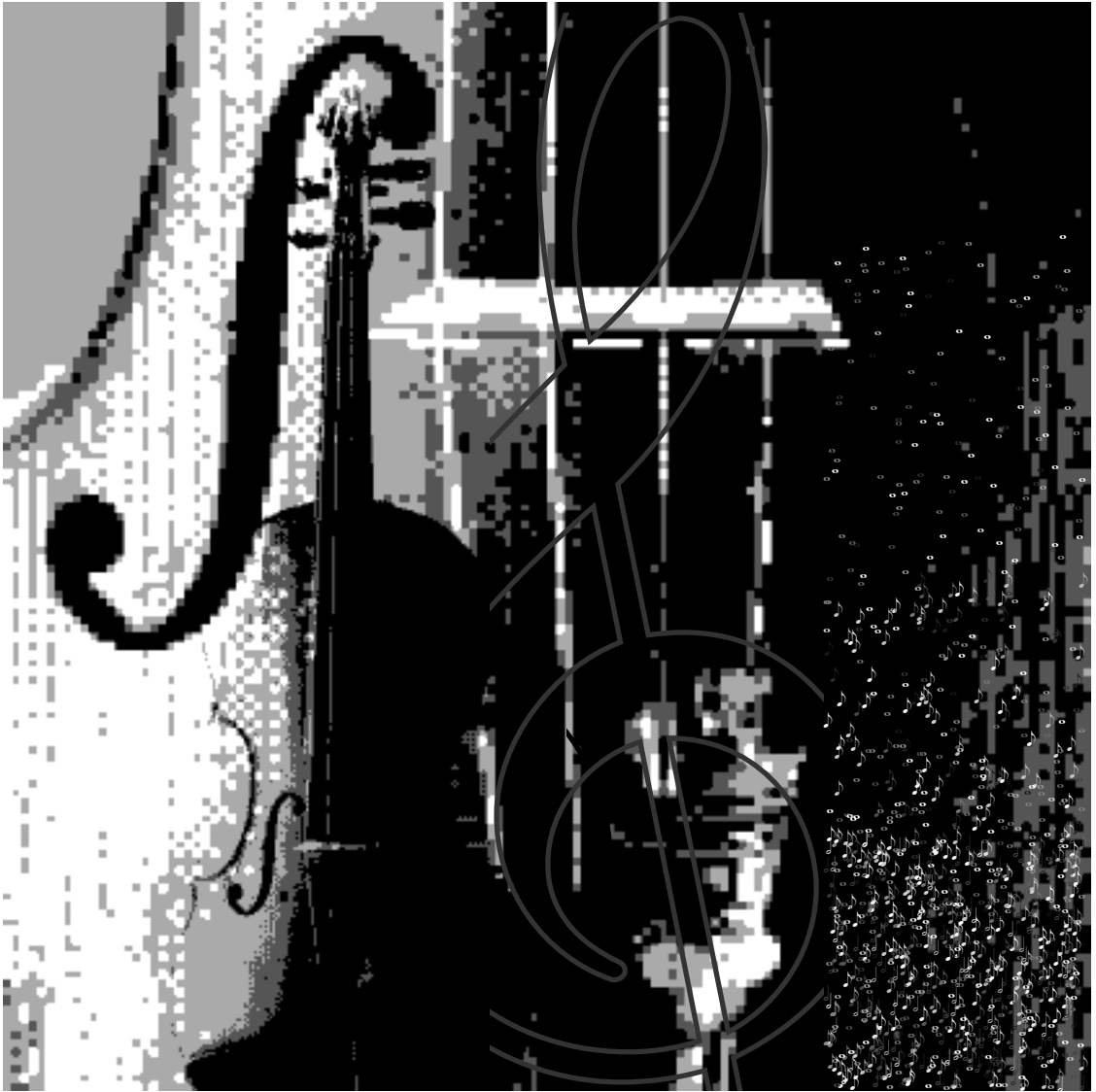
This chapter contains a number of examples of designs that demonstrate the potential of PostScript image making.



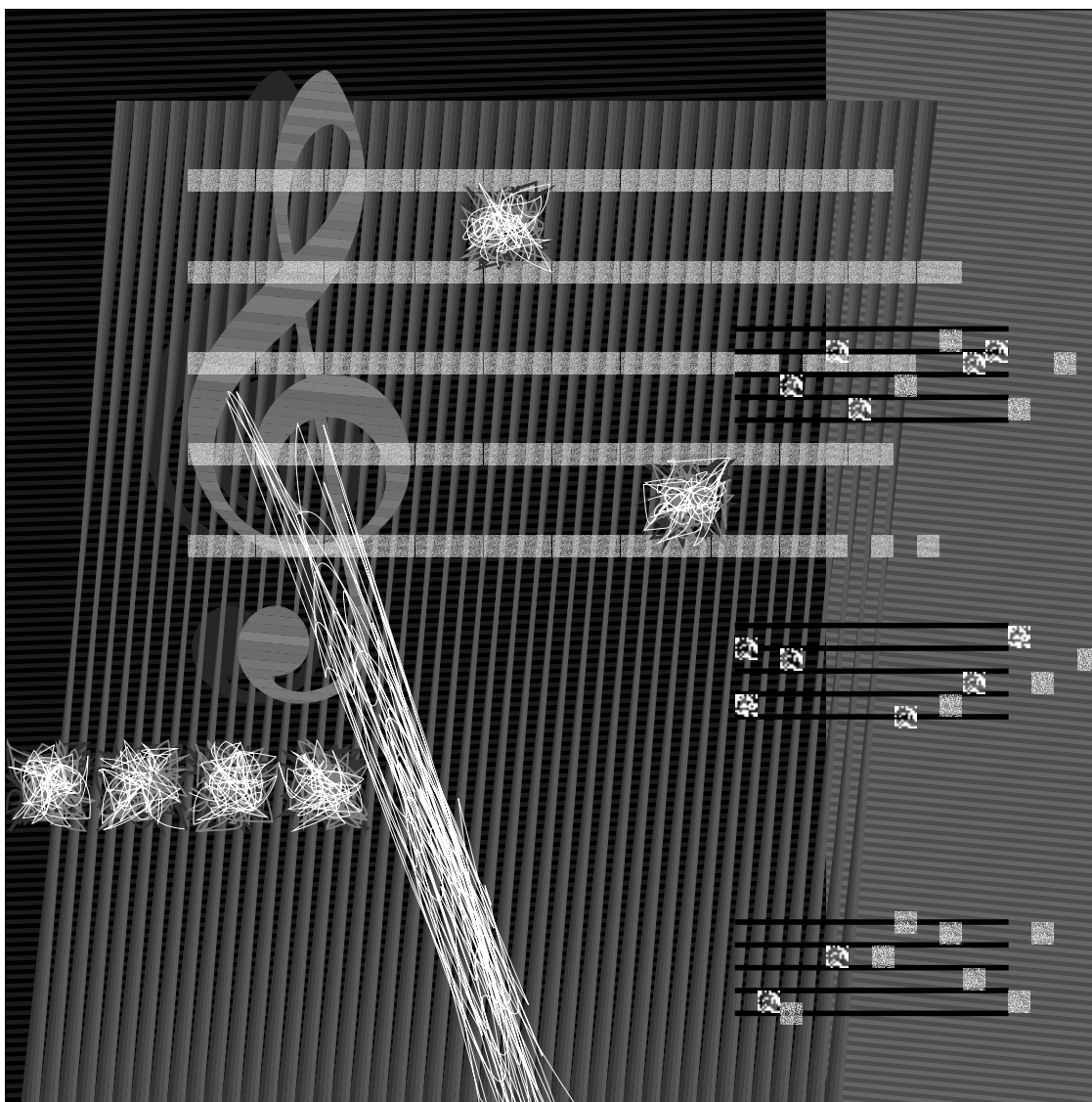
after the Romance



bit o Romance



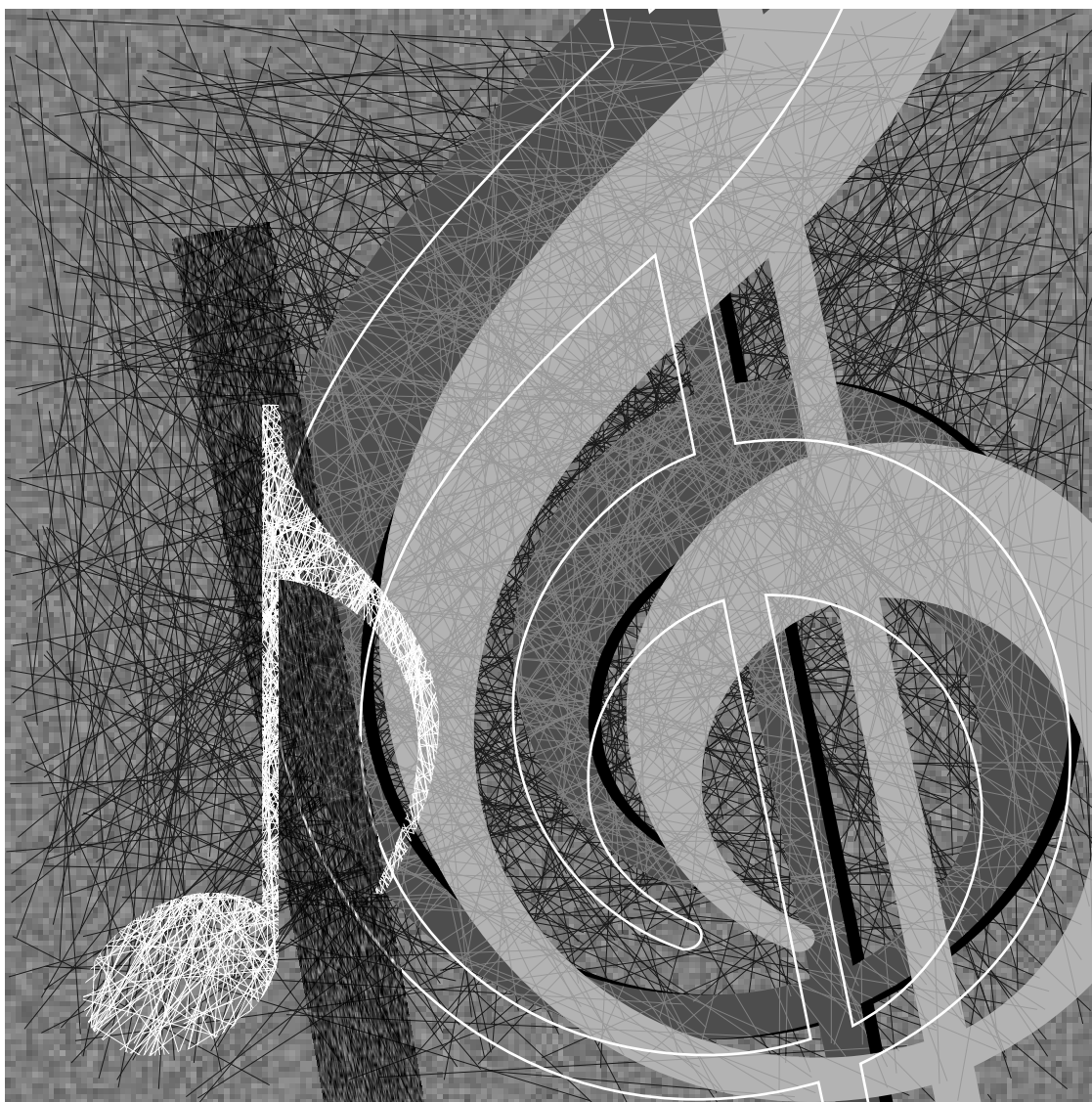
vioLynn



excited



violin Switch



radClef



tangent, sine, & cosine charts

Consult this chart for matrix transformations. See section 10.4.

degree	tangent	degree	tangent
0°	0.0	46°	1.036
1°	0.0175	47°	1.072
2°	0.0349	48°	1.111
3°	0.0524	49°	1.150
4°	0.0699	50°	1.192
5°	0.0875	51°	1.235
6°	0.1051	52°	1.280
7°	0.1228	53°	1.327
8°	0.1405	54°	1.376
9°	0.1584	55°	1.428
10°	0.1763	56°	1.483
11°	0.1944	57°	1.540
12°	0.2126	58°	1.600
13°	0.2309	59°	1.664
14°	0.2493	60°	1.732
15°	0.2679	61°	1.804
16°	0.2867	62°	1.881
17°	0.3057	63°	1.963
18°	0.3249	64°	2.050
19°	0.3443	65°	2.145
20°	0.3640	66°	2.246
21°	0.3839	67°	2.356
22°	0.4040	68°	2.475
23°	0.4245	69°	2.605
24°	0.4452	70°	2.747
25°	0.4663	71°	2.904
26°	0.4877	72°	3.078
27°	0.5095	73°	3.271
28°	0.5317	74°	3.487
29°	0.5543	75°	3.732
30°	0.5774	76°	4.011
31°	0.6009	77°	4.331
32°	0.6249	78°	4.705
33°	0.6494	79°	5.145
34°	0.6745	80°	5.671
35°	0.7002	81°	6.314
36°	0.7265	82°	7.115
37°	0.7536	83°	8.144
38°	0.7813	84°	9.514
39°	0.8098	85°	11.43
40°	0.8391	86°	14.30
41°	0.8693	87°	19.08
42°	0.9004	88°	28.64
43°	0.9325	89°	57.29
44°	0.9657	90°	360.0
45°	1.0		

Consult this chart for matrix transformations. See section 10.4.

degree	sine	cosine	degree	sine	cosine
0°	0.0	1.0	46°	0.7193	0.6947
1°	0.0175	0.9998	47°	0.7214	0.6820
2°	0.0349	0.9994	48°	0.7431	0.6691
3°	0.0523	0.9986	49°	0.7547	0.6561
4°	0.0698	0.9976	50°	0.7660	0.6428
5°	0.0872	0.9962	51°	0.7771	0.6293
6°	0.1045	0.9945	52°	0.7880	0.6157
7°	0.1219	0.9925	53°	0.7986	0.6018
8°	0.1392	0.9903	54°	0.8090	0.5878
9°	0.1564	0.9877	55°	0.8192	0.5736
10°	0.1736	0.9848	56°	0.8290	0.5592
11°	0.1908	0.9816	57°	0.8387	0.5446
12°	0.2079	0.9781	58°	0.8480	0.5299
13°	0.2250	0.9744	59°	0.8572	0.5150
14°	0.2419	0.9703	60°	0.8660	0.5000
15°	0.2588	0.9659	61°	0.8746	0.4848
16°	0.2756	0.9613	62°	0.8829	0.4695
17°	0.2924	0.9563	63°	0.8910	0.4540
18°	0.3090	0.9511	64°	0.8988	0.4384
19°	0.3256	0.9455	65°	0.9063	0.4226
20°	0.3420	0.9397	66°	0.9135	0.4067
21°	0.3584	0.9336	67°	0.9205	0.3907
22°	0.3746	0.9272	68°	0.9272	0.3746
23°	0.3907	0.9205	69°	0.9336	0.3584
24°	0.4067	0.9135	70°	0.9397	0.3420
25°	0.4226	0.9063	71°	0.9455	0.3256
26°	0.4384	0.8988	72°	0.9511	0.3090
27°	0.4540	0.8910	73°	0.9563	0.2924
28°	0.4695	0.8829	74°	0.9613	0.2756
29°	0.4848	0.8746	75°	0.9659	0.2588
30°	0.5000	0.8660	76°	0.9703	0.2419
31°	0.5150	0.8572	77°	0.9744	0.2250
32°	0.5299	0.8480	78°	0.9781	0.2079
33°	0.5446	0.8387	79°	0.9816	0.1908
34°	0.5592	0.8290	80°	0.9848	0.1736
35°	0.5736	0.8192	81°	0.9877	0.1564
36°	0.5878	0.8090	82°	0.9903	0.1392
37°	0.6018	0.7986	83°	0.9925	0.1219
38°	0.6157	0.7880	84°	0.9945	0.1045
39°	0.6293	0.7771	85°	0.9962	0.0872
40°	0.6428	0.7660	86°	0.9976	0.0698
41°	0.6561	0.7547	87°	0.9986	0.0523
42°	0.6691	0.7431	88°	0.9994	0.0349
43°	0.6820	0.7314	89°	0.9998	0.0175
44°	0.6947	0.7193	90°	1.0	0.0
45°	0.7071	0.7071			



PS utilities

D.1

PostScript programs do not necessarily print a graphic. They are used for a wide range of utilities, from getting feedback from the printer to setting printer defaults.

start-up page

This PostScript program stops the printer from printing the start-up page when it is first turned on.

```
%!PS-Adobe-2.0
%%Title:start-upPage1.ps

% stops start-up page on LaserWriter

serverdict begin 0 exitserver

statusdict begin
false setdostartpage
```

This PostScript program restores the printing of the start-up page.

```
%!PS-Adobe-2.0
%%Title:start-upPage2.ps

% restores start-up page on LaserWriter

serverdict begin 0 exitserver

statusdict begin
true setdostartpage
```

D.2

page count

This PostScript program prints a test page which contains the number of pages printed on the laser printer.

```
%!PS-Adobe-2.0
%%Title:pagecount.ps

% pages printed on a LaserWriter

/Helvetica findfont 14 scalefont setfont

30 500 moveto
(The number of pages that have been printed are: ) show
```

```
statusdict begin pagecount
10 string cvs show

showpage
```

D.3 *printing multiple copies*

There is a convenient way to print multiple copies of a page. When `showpage` is used, a variable named `#copies` is referenced. It can be given a new value for the number of copies required. The default is set at 1.

```
%!PS-Adobe-2.0
%%Title:copies.ps

72 144 moveto
/AvantGarde-Demi findfont 200 scalefont setfont

(ND) show

/#copies 3 def

showpage
```

D.4 *listing of available fonts*

There are a number of ways to determine what fonts are available in your printer. A benefit of doing this is that it's a convenient way to learn the required spelling for a particular font. Included here are four programs for listing the font directory. The first pair is for the Apple LaserWriter and the second pair is for the NeXT computer. The first program of the set will print the font list twice, the second font listing appearing in its font style. The other program of the set sends the list to the standard output file. The primary difference between the two sets of programs is that the font directory in the LaserWriter is named `FontDirectory` and on the NeXT is named `SharedFontDirectory`.

```
%!PS-Adobe-2.0
%%Title:prFontDir.ps
%%Creator:John F Sherman
%%CreationDate:29 Nov 1990

% prints LaserWriter FontDirectory.ps

/Times-Bold findfont 10 scalefont setfont

/left 72 def
/top 720 def
/newline {show currentpoint exch pop 14 sub left exch moveto}
def
/str 30 string def
/printDir {FontDirectory
```

```

        {pop str cvs newline} forall } def

left top moveto
printDir

( ) newline
(Total memory: ) show
vmstatus str cvs newline

(Memory used: ) show
str cvs newline
pop

( Free memory: ) show
vmstatus exch sub str cvs newline
pop

/printDir {FontDirectory
           {12 scalefont setfont str cvs newline} forall } def

/left 240 def
left top moveto
printDir

showpage

%!PS-Adobe-2.0
%%Title:sendFontDir.ps
%%Creator:John F Sherman
%%CreationDate:29 Nov 1990

% Sends font list to the standard output file.
% The file will either be a text file on your drive
% or it will flash by in a window on your monitor.

FontDirectory {pop == flush} forall

The following programs are for the NeXT SharedFontDirectory listing.

%!PS-Adobe-2.0 EPSF-1.2
%%Title:SharedFontDir1.eps
%%Creator:John F Sherman
%%CreationDate:29 Nov 1990
%%BoundingBox:0 0 360 525

% NeXT SharedFontDirectory listing

/Times-Bold findfont 10 scalefont setfont

/left 36 def
/newline {show currentpoint exch pop 14 sub left exch moveto}
def

```

```

/str 30 string def
/printDir {SharedFontDirectory
           {pop str cvs newline} forall } def

0 0 360 525 rectstroke
left 500 moveto
printDir

/printDir {SharedFontDirectory
           {12 scalefont setfont str cvs newline} forall } def

/left 200 def
left 500 moveto
printDir

%!PS-Adobe-2.0
%%Title:SharedFontDir2.ps
%%Creator:John F Sherman
%%CreationDate:29 Nov 1990

% NeXT SharedFontDirectory listing
% sends font list to the standard output file,
% usually the Console window.

SharedFontDirectory {pop =} forall

```

D.5 *getting the bounding box*

Often it may be useful to receive information from the PostScript interpreter. The following programs are variations on obtaining a path's bounding box and having it sent to the standard output file.

```

%!PS-Adobe-2.0
%%Title:seeBBox1.ps

/Times-Roman findfont
300 scalefont setfont

150 150 moveto (g) true charpath

pathbbox
= = = =

%!PS-Adobe-2.0
%%Title:seeBBox2.ps

200 200 72 0 360 arc

pathbbox
= = = =

```

In this program, `flattenpath` is used before `pathbbox`. `flattenpath` replaces all uses of `curveto` with an equivalent series of `lineto` line segments. Otherwise, `pathbbox` returns values that do not reflect the actual bounding box.

```

%!PS-Adobe-2.0
%%Title:seeBBox3.ps

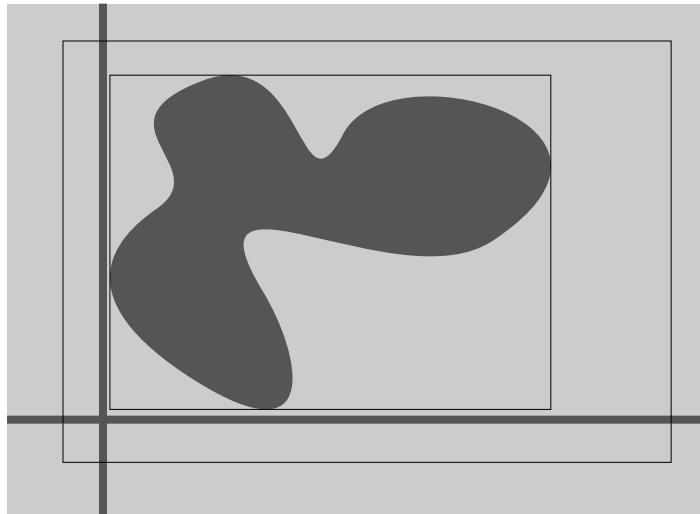
/blob {20 79 moveto
       43.0878 94.8731 -5.0006 110.846 37 127 curveto
       76 142 73.0001 74 90 107 curveto
       107 140 213.0496 111.0607 146 67 curveto
       111 44 27.7468 100.988 60 48 curveto
       74 25 83 -16 34 15 curveto
       -15 46 4 68 20 79 curveto closepath} def

blob
pathbbox
= = = =

blob
flattenpath
pathbbox
= = = =

```

This graphic shows the two different bounding boxes.



D.6 *sending yourself messages*

This procedure can be useful when trying to debug a lengthy PostScript program and it's difficult to discover where the problem is located. Another use for this procedure is to let you know during a long printing job how much of the program has been completed. This is done by leaving messages to yourself within the program that are sent to the standard output file when they are encountered during processing. If you are debugging, the message might be "made it this far." If you are monitoring a print job, there could be a series of messages throughout

the file saying “completed section 1,” “completed section 2,” “completed section 3,” and so on. Since this example is pretty short, the messages will flash by rather quickly.

In messages sent by `print (\n) print flush`, `print` writes a string (your message between parentheses) to the standard output file. `flush` means send it right away. `(\n) print` is like hitting return on a typewriter. Without it, all the messages would line up in one line with no spaces between. `\n` means newline.

```

%!PS-Adobe-2.0
%%Title:message1.ps

% the message is sent to the standard output file
/message {print (\n) print flush} def
/Helvetica findfont 100 scalefont setfont

(10) message
(9) message
(8) message
(7) message
(6) message
(5) message
(4) message
(3) message
(2) message
(1) message
100 100 moveto (Blast off !) show

showpage

```

D.7 *getting what's on the stack*

Often a PostScript program appears to work correctly, but unknown to you, unused items are left on the stack. For example, `kshow` pushes twelve integers that are the string's character codes onto the stack. These are integers that are not needed by this particular `kshow` procedure, and are usually removed by `12 {pop} repeat`.

```

%!PS-Adobe-2.0
%%Title:seeStack.ps

/AvantGarde-Demi findfont 72 scalefont setfont
.9 setgray
36 36 moveto
{-3 0 rmoveto currentgray .12 sub setgray} (RAINBOW) kshow
pstack

```

Another example is the following:

```

%!PS-Adobe-2.0
%%Title:seeMatrix.ps

matrix defaultmatrix ==
matrix defaultmatrix pstack

```



glossary

argument

An argument is used when explaining a PostScript operator that is expecting information supplied by the programmer. For example, `moveto` expects two arguments, an *x* and *y* location, such as `72 72 moveto`. `72 72` are the two arguments for `moveto` in this example.

array

An array is a collection of values or items for some needed purpose. For example, `[1 0 0 1 0 0]` would be an example of a matrix array. In PostScript, the “[” and “]” (bracket-left and bracket-right) are the left and right boundaries of the collection of items or array. An array may be an argument for a PostScript operator, such as `[1 0 0 1 0 0] concat`.

ASCII

Pronounced “ass key,” ASCII are the initials for the American Standard Code for Information Interchange. The ASCII chart is a standard table of characters that all computer systems have in common. Each character has several means of identification depending on the need. There are decimal, binary, octal, and hexadecimal identifications for each of the 256 entries. Some uses of the different identifications are to print special characters outside of the standard keyboard character set, kern characters, or to image scanned data. See Appendix A.

boolean

A boolean is something that is either true or false.

decimal character code

There are 256 possible characters in the ASCII character set, numbered 0 – 255. Each number is the decimal identification for that character. For example, the decimal number for the letter *A* is 65. Appendix A is a table of all these identifications.

hexadecimal character code

Hexadecimal, sometimes referred to as HEX, is an alternate form of writing characters. In hexadecimal, each character is identified by a pair of characters. The pair is made using the digits 0 through 9 and the letters *A* through *F*. For example, the character *A* is written as `41` in hex and *Z* is `5A`. See decimal character code above and appendix A.

key

When a procedure is defined, the name of the procedure is considered a key to the procedure action. PostScript operators are also considered to be keys to the value of the operator.

octal character code

The octal code is one of several ways of identifying a character. See decimal character code above and special characters below. The octal number for the letter *A* is 101 and would be written in PostScript as `8#101` when used with the `awidthshow` kerning operator or `(\101BC)` in a string. See appendix A.

operand

An operand is the argument for an operator. In `72 72 moveto`, `72 72` are the operands for the operator `moveto`.

operator

An operator is one of the commands of PostScript. It may require operands or it may stand alone like `showpage`, which is used for printing.

procedure

PostScript is an extensible language, meaning a name can be given to some operation such as drawing a square or adding several values together. This is called a procedure. Once the procedure is created, its name is used in the program as if it is a command. A procedure is typically used in cases where the action will be needed several times and the program would be more efficient and understandable having one. See section 3.2.

pushed

When discussing stacks, items are *pushed* onto the stack and *popped* off. For example, a by-product of the `kshow` operator is that it pushes character codes onto the operand stack. The `pop` operator discards them.

special characters

There are a number of special characters to be aware of.

“<” and “>” identify hexadecimal, such as `<BC 23 8F>`.

“(” and “)” identify text strings, such as `(type and letters)`.

“/” identifies a key or name for a procedure, for example, `/square`.

“\” identifies a character outside of the standard ASCII character set by its octal number. For example, `(\341SOP)` in a PostScript program will print the ligature of A and E, to get `ÆSOP`. 341 is the octal code for `Æ`. If you need a “\” in a string, use two (this is a backslash: `\\`). See appendix A.

stack

A stack is like a plate dispenser at a cafeteria. The first plate in is the last plate out and the last plate in is first plate used. In PostScript, the stack is an area of memory that values, operators, and other items are *pushed* onto. The first item in a line of PostScript code will be the first item on the stack and the last to be used.

There are four stacks; the operand, dictionary, execution, and graphic state. The operand stack is covered in section 2.8.

string

A string is a group of characters or words within a program that the program prints or displays on the computer screen. For example, in the program fragment `72 72 moveto (type) show`, the word `type` is a text string four characters long identified and contained within the special characters “(” and “).”

variable

A variable is a value that may change depending on the situation. In the example `x -12 moveto`, `x` is a variable that may have been defined earlier in the program to equal 0 or 72. `x` may be used twenty times in a program and it would be easier to change `x` than to search for the twenty numbers.

For example:

```
/x 20 def           % define x to be 20
/word (word) def   % define word to be string "word"
```