More Postscript

Steven R. Bagley

Introduction

- Spend some more time looking at writing PostScript programs
- Emphasis on writing programs that create graphics
- Also see how we can replicate common program language structures in PostScript

some of this will form part of the lab

Resources

- Uploaded some PDF tutorials to the website
- A list of useful PostScript commands
- PostScript Language Reference Manual is on Adobe's website
- Lab exercises…

Chapter 8 is brilliant for seeing how the operators work
Nothing beats writing code for learning how to write code, so do the lab exercises…

PostScript

- Programming language
- Stack-based, like FORTH
- Rich support for graphics
  - Primarily path-based

Drawing

- Drawing operators are:

`newpath`	clear the current path
`x y moveto`	set the current point to (x,y)
`x y lineto`	draw a line to point (x,y)
`x y rmoveto`	move to currentpoint + (x,y)
`x y rlineto`	draw line to currentpoint +(x,y)
`x y r ang1 ang2 arc`	append anticlockwise circular arc

rmoveto/rlineto allow for relative motion
arcs are centred on x y, with raidus r from angle 1 to angle 2

PostScript Paths

- Paths do not need to be continuous
- Can use `moveto` to jump to another position
- Note `closepath` only connects back to the end of the current sub-path

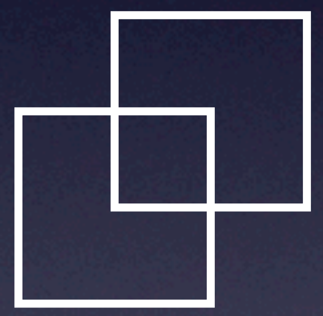
```
 72  72 moveto   72 0 rlineto
  0  72 rlineto -72 0 rlineto
closepath

108 108 moveto   72 0 rlineto
  0  72 rlineto -72 0 rlineto
closepath

stroke
```

PostScript Paths

- Paths do not need to be continuous
- Can use `moveto` to jump to another position
- Note `closepath` only connects back to the end of the current sub-path



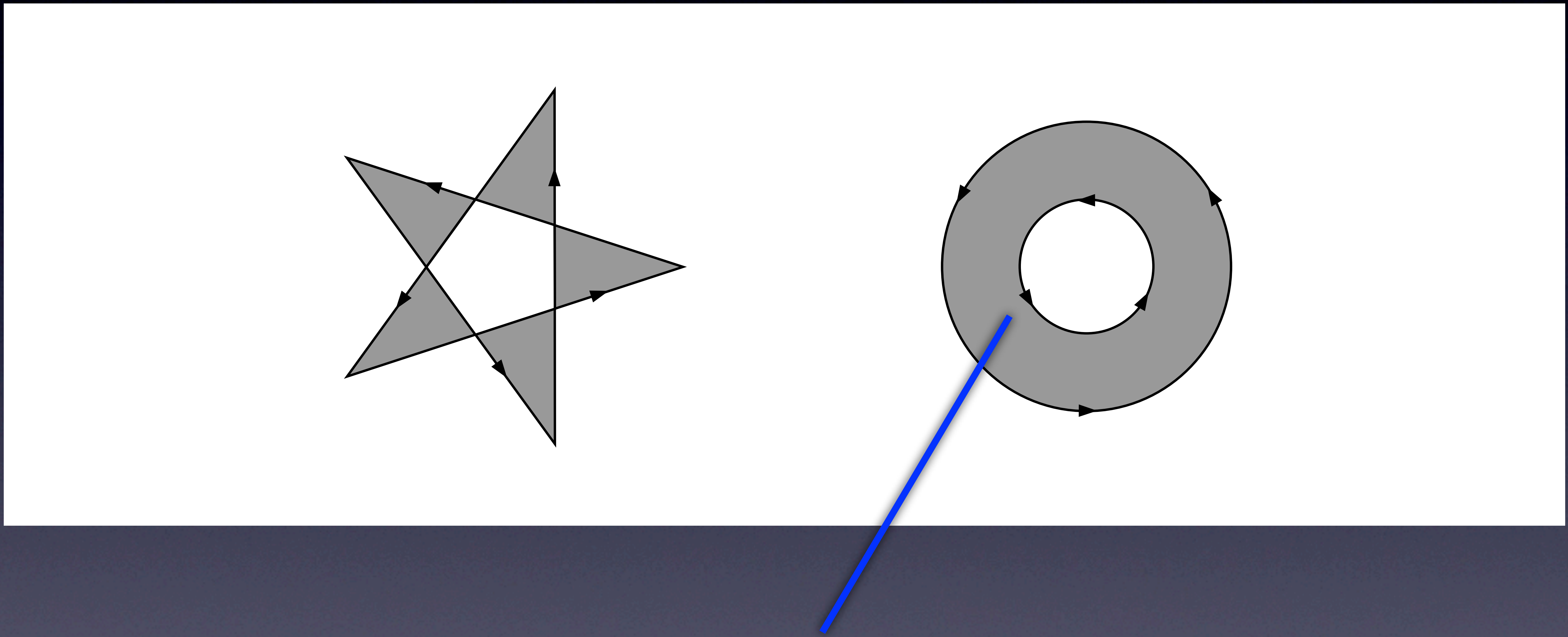
Filled Paths

- PostScript provides two fill models
  - Non-zero winding rule (`fill`)
  - Even-Odd fill rule (`eofill`)
- Both decided whether a pixel is set by tracing a ray out from the pixel to infinite
- And seeing how the ray crosses the path

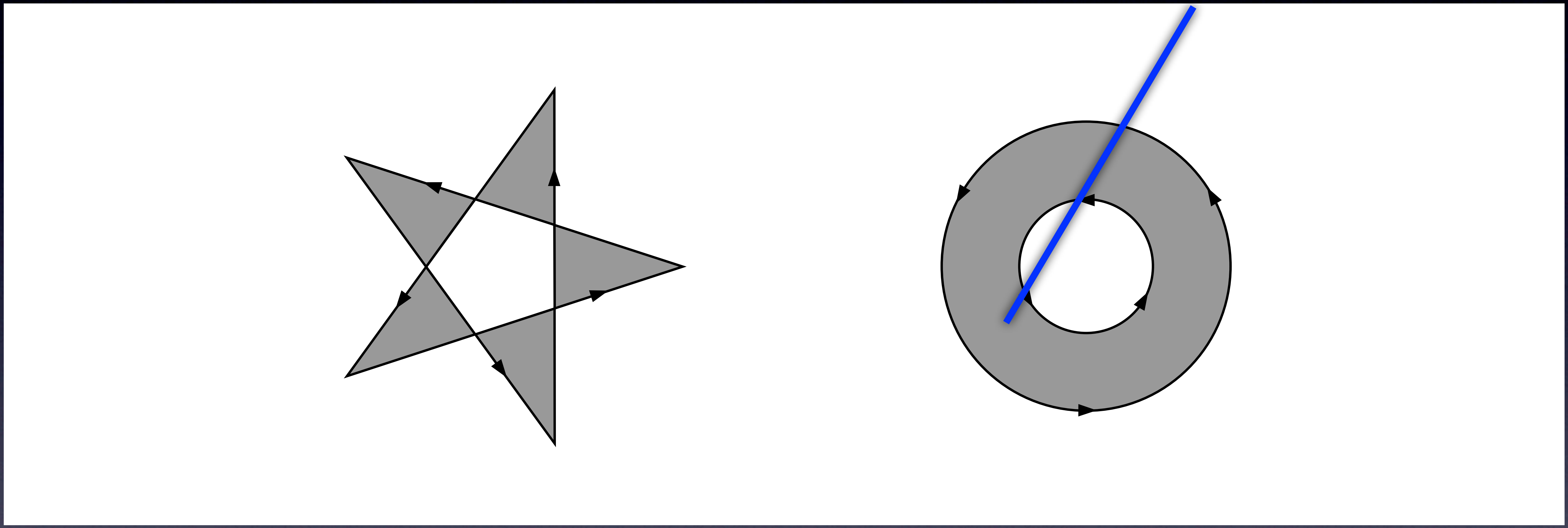
Even-Odd Fill Rule

- The simpler of the two…
- Counts how many times the ray crosses the path
- If it’s an odd number, the pixel is inside the shape (and so set)
- If even, the pixel is outside and so not set

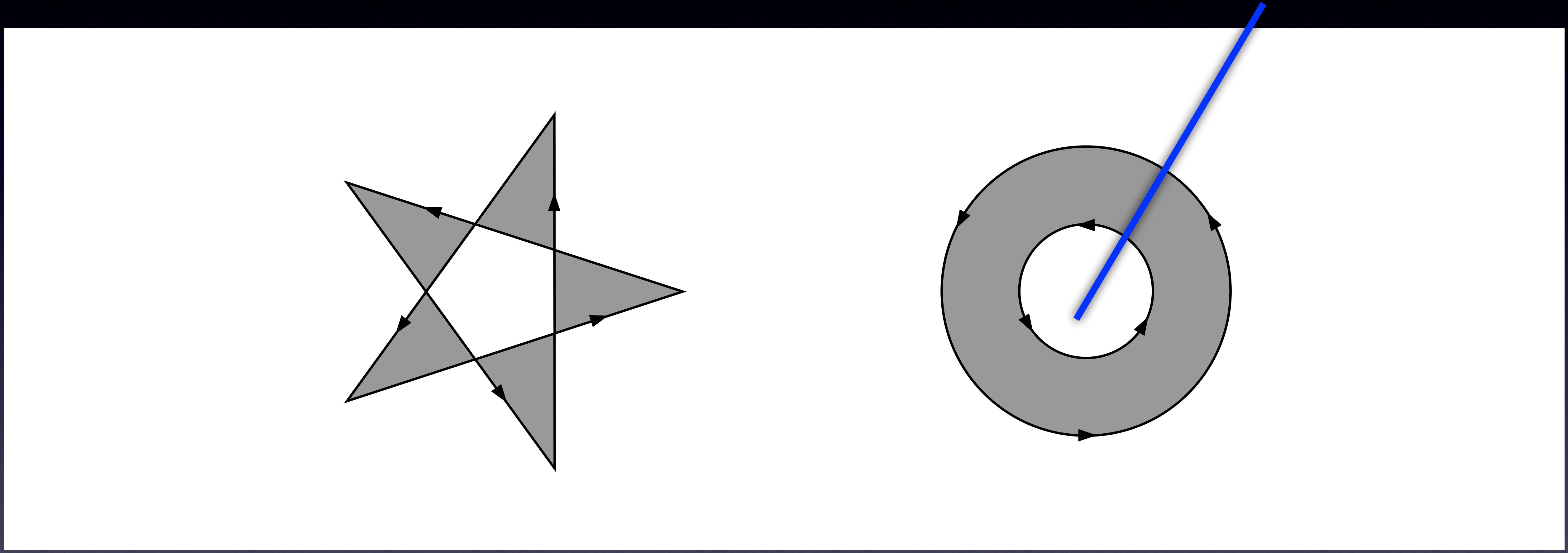
Even-odd Rule



Even-odd Rule



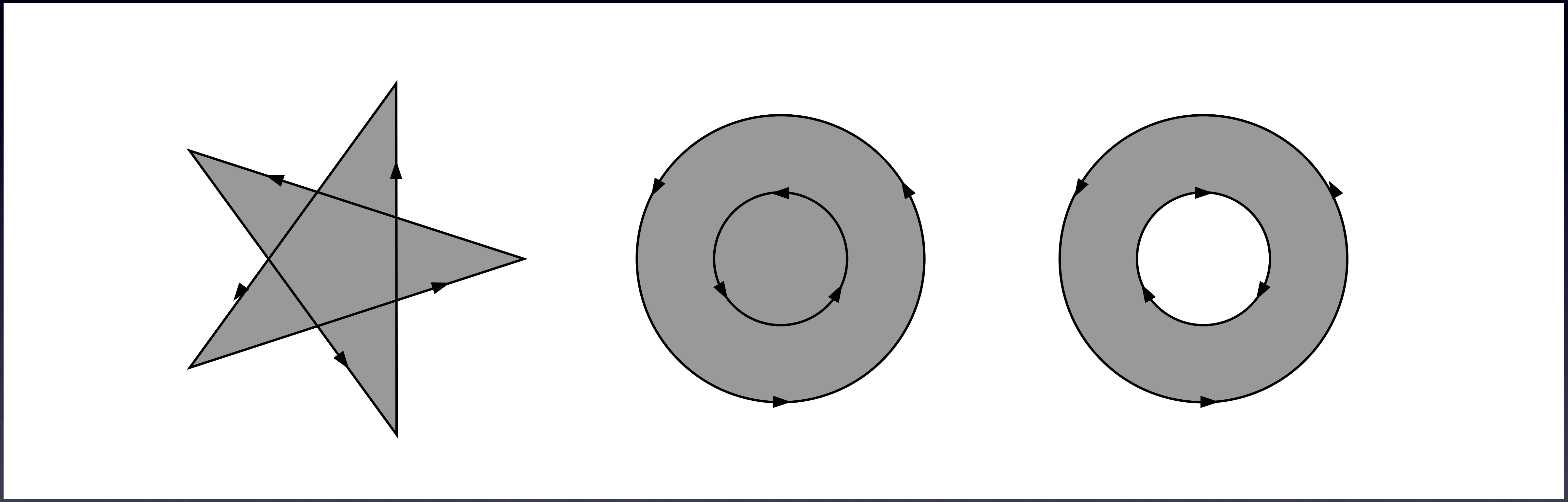
Even-odd Rule



Non-zero winding rule

- Starts with a count of 0
- Adds one each time the ray crosses from left-to-right
- Subtracts one each time ray crosses from right-to-left
- If value is non-zero, it is inside

Non-zero winding rule



Graphics State

- PostScript maintains a ‘graphics state’
- Modified by operators
- Sometimes helpful to be able to undo operations
- PostScript lets us save and restore the graphics state

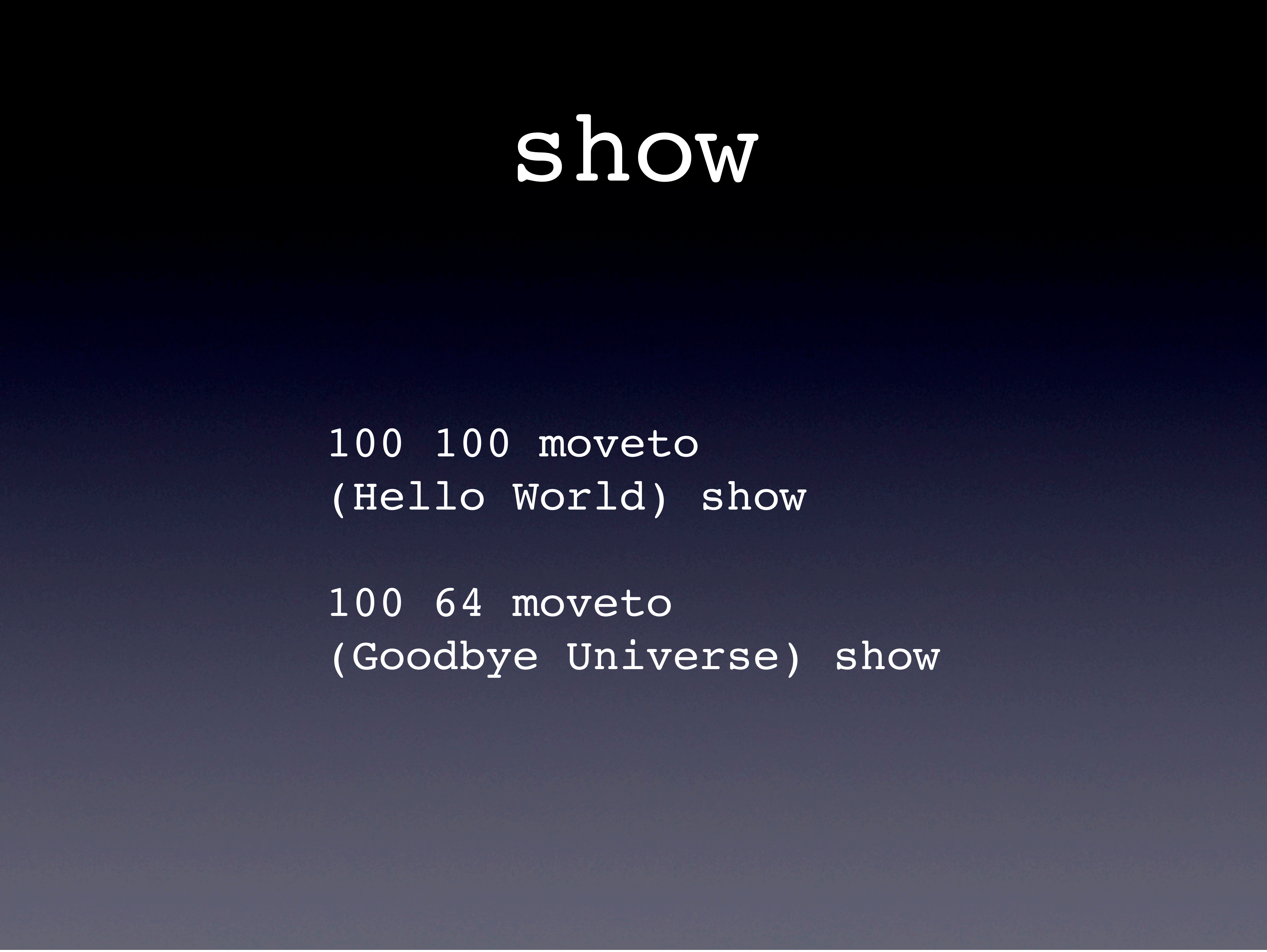
Graphics State Stack

- Stored on the graphics state stack using `gsave`
- Pushed onto the top of the stack so can have multiple levels
- Popped and restored by calling `grestore`
- Preserves the state only, has no effect on any imaged marks

Text

- PostScript provides rich support for text
- Including outline fonts
- Simplest approach is the `show` operator
- Pops a string and displays it on screen
- Character by character starting at the current point

Outline fonts use vector descriptions of the text rather than bitmaps
Or rather glyph by glyph



show

```
100 100 moveto
(Hello World) show

100 64 moveto
(Goodbye Universe) show
```

But…
We need to set the font…

Fonts

- Need to specify the font and point size
- Again stored in the graphics state
- PostScript interpreter have mechanisms for storing and retrieving fonts
- Or you can include the definition in your PS file
- In either case, we find them using a name

Postscript name

`findfont`

- Need to get the font definition
- Use `findfont` based on the fonts name

  ```
  /Times-Roman findfont
  /Helvetica findfont
  ```

- Returns the dictionary representing the font
- You'll need to know the exact font name…

Font names are not always what you'd expect either

`setfont`

- The `setfont` operator takes a font dictionary and makes it the current font
- So:

  ```
  /Times-Roman findfont setfont
  ```

- Would set Times-Roman as the current font
- However, it would set it as a 1pt high font…

Point Size

- PostScript fonts are designed to be 1 point high
- Which isn't that useful
- So if we want it to be a different size we need to scale it first

Scaling Fonts

- To get the font to a usable point size, we can use the `scalefont` operator
- Takes a font and a point size and scales it to that size returning a new scaled font dict…
- So:

```
/Helvetica findfont 36 scalefont setfont
```

- Would set the font to 36pt Helvetica

Easier to scale the font than to have to scale the whole CTM

Selecting Fonts

- In PostScript Level 1, that was the only method to set the font
- However, its slow (and long)
- PostScript Level 2 introduced `selectfont` which combines all the above steps

  ```
  /Helvetica findfont 36 scalefont setfont
  /Helvetica 36 selectfont
  ```

- Use this instead…

And is much faster

Showing Text

- `show` is the simplest text showing operator
- Advances by the glyph width after showing each character
- But there are alternatives that allow you to modify the advancement
- Good for getting nice textual effects (kerning, tracking etc.)

Alternatively, you can just break the string up and move to a new point…

Showing Text

- Text is also affected by the graphics state
- So can be rotated, scaled and translated like paths

String Width

- Postscript also lets you find out how big a piece of text is
- Using the `stringwidth` operator
  `(Hello World) stringwidth`
- Pushes `x` and `y` displacement on the stack
- Often need to pop the `y` value
- Can be used to centre text…

Since it'll be zero for horizontal text
Note even rotated text has a zero displacement

Centred Text

```
/Helvetica 72 selectfont
100 100 moveto

(Hello World) stringwidth pop

2 div neg 0 rmoveto
(Hello World) show
```

Text Bounding Box

- Also possible to obtain the bounding box of a piece of text
- Slightly convoluted
- Firstly, we need to convert the text into a path using `charpath`
- This is a path like any other (can be stroked, filled, used as a clipping path etc.)

charpath allows for some great text effects…

Text Bounding Box

- Once we have the path we can flatten it to remove the curves
- Using `flattenpath`
- And then use `pathbbox` to get the bounding box
- As the coordinates of the lower-left and upper-right corners

Text Bounding Box

```
/Helvetica 72 selectfont
100 100 moveto

(Hello World) true charpath
flattenpath pathbbox
```

Not the bool passed to charpath -- specifies whether you want a path for stroking or filling
true means suitable for filling
false means only suitable for stroking

Variables

- Aren’t strictly necessary — could just keep everything on the stack
- But we’d spend a lot of time `dup`, `exch` and `roll`ing the stack about
- Increasingly more complex to follow the code
- So nice to be able to use variables

A good example would be to store the bbox of the text

Variables

- We can simulate variables by using dictionaries (including `userdict`)
- Associate value with a name in some dictionary
- Use the `def` operator

```
/foo 42 def
/x 100 def
```

name as in a postscript name

Variables

- Can then use the name as an operator to get the value
- So given

  ```
  /foo 42 def
  /x 100 def
  ```

- Then using `x` would put `100` on the stack, `foo` would put `42` on the stack

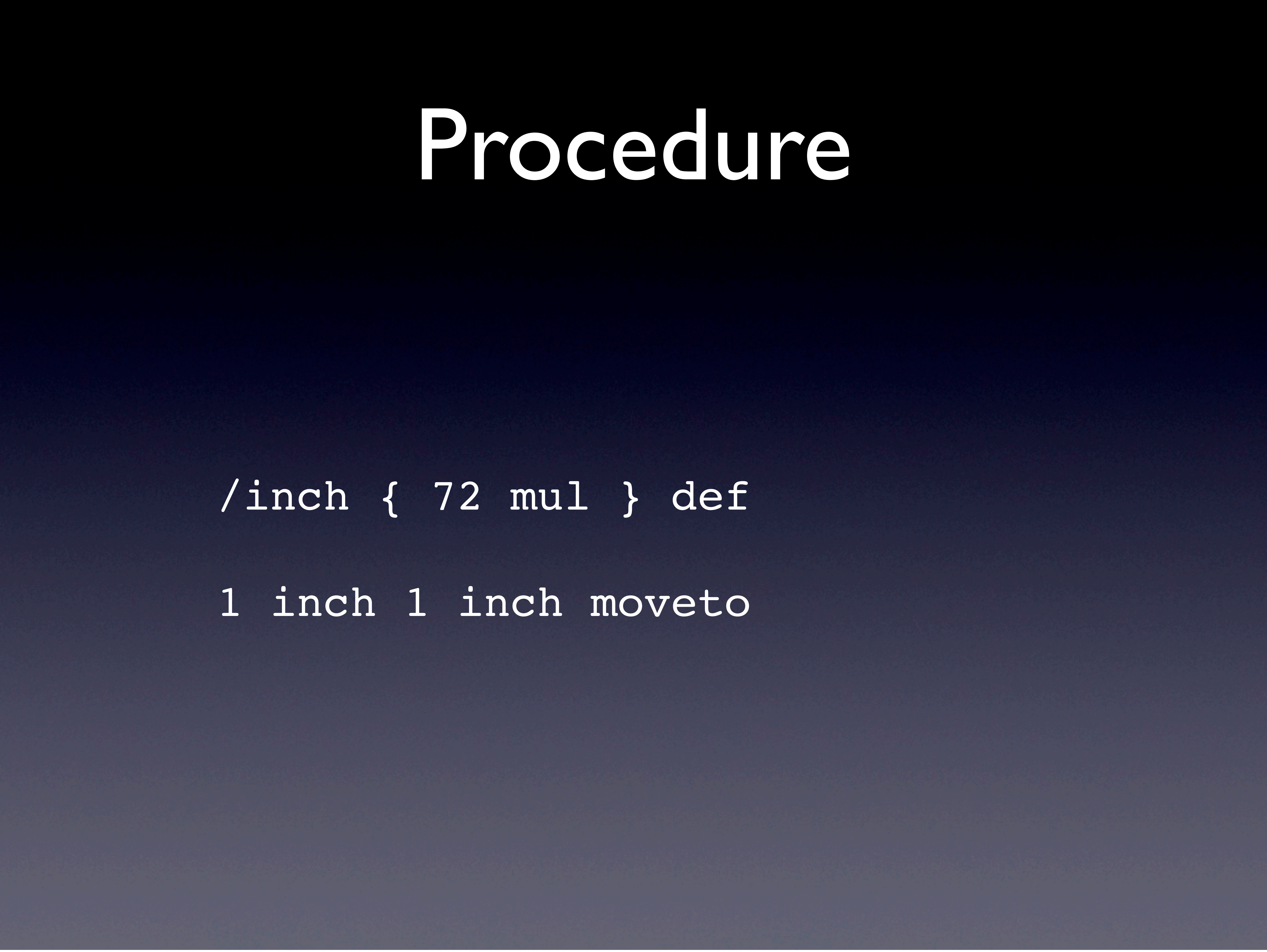
Variables

- If the values are already on the stack (e.g. if returned from an operator)
- Then we can use `exch` to manipulate the stack to get them in the right order

```
… pathbbox /ury exch def /urx exch def …
```

Procedures

- Implemented in the same way as variables
- This time we associate an executable array with a name
- Then using the name calls the code to be executed



Procedure

```
/inch { 72 mul } def

1 inch 1 inch moveto
```

A very simple procedure that converts from inches to points
NOte how it takes parameters

Parameters

- Easy to pass parameters to a procedure
- Just push them on the stack before calling the procedure
- The procedure can then access them
- Fundamentally the same as what happens in C behind the scenes
- Made explicit in PostScript

Local Variables

- Danger of variable corruption if a procedure uses a name used elsewhere
- Can simulate local variables by pushing a dictionary onto the dictionary stack
- All variables are then defined in that dict
- Can pop the dict off the stack when procedure ends

Local Variables

- Create a dictionary using dict, need to specify an initial size
  ```
  5 dict
  ```
- Push it on the dictionary stack using `begin`
- Execute procedure code
- Pop it off the stack using `end`

Best to get the initial size to match the number of variables

Local variables

```
% x y r drawSphere

/drawSphere
{
5 dict begin

% draw sphere code
…

end
} def
```

A very simple procedure that converts from inches to points
NOte how it takes parameters

Static Variables

- Can even simulate static variables in the same manner
- This time precreate the dictionary and associate it with a name in `userdict`
- Then call the dictionary up using the name and push it on the stack

Local variables

```
/shaddict 5 dict def

/shadow
{
shaddict begin

end
}
```

A very simple procedure that converts from inches to points
NOte how it takes parameters

Conditionals

- Postscript allows code to be executed conditionally too
- The `if` operator will execute code only if a boolean is true

  ```
  bool { … } if
  ```

- The executable array is only execute if the bool is true

Conditionals

- if…else works in the same way using the ifelse operator
  bool { … } { … } ifelse
- If the bool is true, the first execute the first executable array
- Else execute the second executable array

Conditionals

- But how do we generate the bool
- Postscript provides a series of comparison operators
- These compare the top two elements on the stack — can combine this with `if`/ `ifelse`

  ```
  12 45 le { … } if
  ```

Drawing

- Comparison operators are

`eq`	equal
`ne`	not equal
`ge`	greater than or equal
`gt`	greater than
`le`	less than or equal
`lt`	less than

Also standard boolean logic operators, and/or/xor/not etc