`DrawString` vs. `WWrite` et al.:

`WWrite()` and `WWrites()` produce output at the current position of the text cursor and appropriately update the position of the text cursor.

The text cursor's position can be set with `GotoRC()` and `GotoXY()`. Its position can be queried via the attributes `x` and `y` (coordinates) and `row` and `col`.

`DrawString()` produces output at the specified position and does not update the text cursor.

`DrawString()` changes only the pixels of the characters; `WWrite()` outputs a rectangle of pixels.

`DrawString()`, in conjunction with `drawop=reverse`, can be used to animate text. (But this does not work on Windows.)

Bottom line:

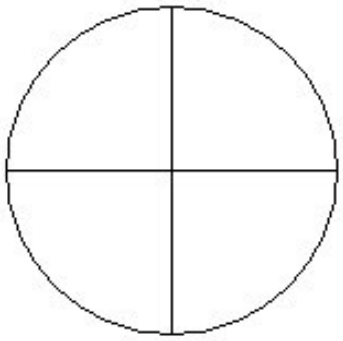
> `WWrite()` is convenient, especially with monospaced text.
>
> `DrawString()` provides full control.

`DrawString`/`TextWidth` and `GotoXY`/`WWrites` are roughly equal "teams".

Coordinate translation

The `dx` and `dy` attributes specify a *translation* of the X and Y coordinates. If `dx` and/or `dy` have a non-zero value the value is automatically added to the X and/or Y coordinate specified in subsequent graphics calls.

Consider this figure:



Here is code to draw it centered at (100,00) with a radius of 75:

```
x :=  y := 100
r := 75
DrawCircle(x, y, r)
DrawSegment(x-r, y, x+r, y, x, y-r, x, y+r)
```

Here is code that uses translation:

```
WAttrib("dx=100","dy=100")
r := 75
DrawCircle(0, 0, r)
DrawSegment(-r, 0, r, 0, 0, -r, 0, r)
```

Changes to `dx` and `dy` are <u>not</u> cumulative.

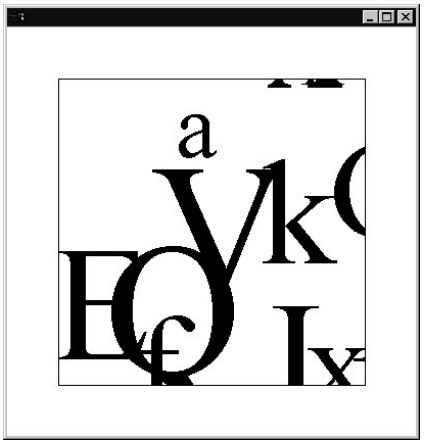
Clipping

Graphics libraries and/or host operating systems typically constrain graphical output to the target window—if a figure extends beyond the bounds of the window the out of bounds pixels are simply not drawn.

In some cases it is desirable to limit drawing to a portion of a window. The procedure `Clip(x, y, w, h)` sets a *clipping region*—no pixels will be drawn outside the specified rectangle.

The following program draws randomly sized characters at random positions on the screen. A clipping region is used to constrain the output to the center of the window.

```
procedure main() # clip1
    WOpen("size=400,400")
    center_square := [50,50,300,300]
    DrawRectangle!center_square
    Clip!center_square
    repeat {
        Font("serif,"||(60+?200)) | stop()
        DrawString(?400, ?400, ?&letters)
        if *Pending() > 0 then
            Event() & Event()
        WDelay(70)
        }
end
```



Example: Clipping and translation

This program draws random circles. A square clipping region is initially established at the center of the window and gradually increased.

When the clipping region reaches the full size of the window, the foreground and background colors are reversed (via the `reverse` attribute), the window is erased, and the process repeats.

Coordinate translation is used both for drawing and defining the clipping region.

```
procedure main() # clip2
    WOpen("size=400,400","dx=200","dy=200")

    rev := create |!["on","off"]
    side := 400
    repeat {

        every i := 1 to side by 5 do {
            WAttrib("dx="||200-i/2,
                    "dy="||200-i/2)

            Clip(0,0,i,i)
            every 1 to 20 do
                DrawCircle(?i, ?i, ?25)

            if *Pending() > 0 then
                Event() & Event()
            WDelay(70)
            }
        WAttrib("reverse="||@rev)
        EraseArea()
        }
end
```

Color specification

A window has attributes for the foreground and background colors (`fg` and `bg`). They can be set via `WAttrib()` or with the `Fg(s)` and `Bg(s)` procedures.

Routines such as `DrawCircle` and `FillRectangle` draw pixels in the foreground color, which is `black` by default.

A simple way to specify a color is by naming one of these *hues*:

```
black       orange
gray        yellow
white       green
pink        cyan
violet      blue
brown       purple
red         magenta
```

One way to think of hue: The basic nature of a color.

Example:

```
procedure main() # color1
    WOpen("size=300,300")
    colors := split("black gray white pink _
       violet brown red orange yellow green _
       cyan blue purple magenta")

    every color := !colors do {
        Bg(color)
        EraseArea()
        until Event() === &lpress
        }
end
```

Color specification, continued

Icon's color naming system was inspired by a 1982 paper by Berk, et al.: *A New Color-Naming System for Graphics Languages* that uses natural language to describe a color. Here is the full form:

lightness	saturation	*hue1*	*hue2*

`pale`
`light`
`medium`
`dark`
`deep`
	`weak`
	`moderate`
	`strong`
	`vivid`
		`hue``[ish]`	*`hue`*

Saturation is a measure of how far the color is from a gray.

Lightness is the intensity of a color.

Examples:

```
pale green
pale weak green
yellow green
greenish yellow
pale greenish yellow
moderate pinkish red
dark bluish purple
```

All elements are optional except for *hue2*. The defaults of `medium` and `vivid` are underlined.

A specification like "`yellow orange`" selects a color halfway between yellow and orange. "`yellowish orange`" specifies a color 3/4 of the way toward orange.

Color specification, continued

The `colrbook` program in the IPL displays a hue with varying levels of lightness and saturation.

Here's a simple program for testing color specifications:

```
procedure main() # color2
    WOpen("size=300,600")

    WAttrib("font=serif,30")
    WWrite()

    y := WAttrib("fheight")
    striph := 75

    while GotoRC(1,1) &
          WWrites(repl(" ",100),"\r") &
          color := WRead() do {
        if *color = 0 then { # <Enter> clears
            EraseArea()
            y := WAttrib("fheight")
            next
            }
        Fg(color) | next
        FillRectangle(0,y,300,striph)
        Fg("black")
        DrawString(10,y+striph/2,color)
        y +:= striph
        }
end
```

Numerical color specification

A color can also be specified numerically, in terms of the brightness of red, green, and blue light. One form is a comma-separated triple of decimal integer values in the range 0 to 65,535:

```
<red>,<green>,<blue>
```

Examples:

```
Fg("60000,0,0")     # bright red
Fg("0,0,30000")     # fairly dark blue
Bg("50000,50000,50000") # light gray
Fg("40000,30000,50000") # pale purple
```

Zero for all three yields black; maximum values yield white.

Alternatively, values can be specified using triples of 1-4 hex digits:

```
Fg("#f00")
Bg("#ff21a")
Fg("#7ffa00b88")
Bg("#123456789abc")
```

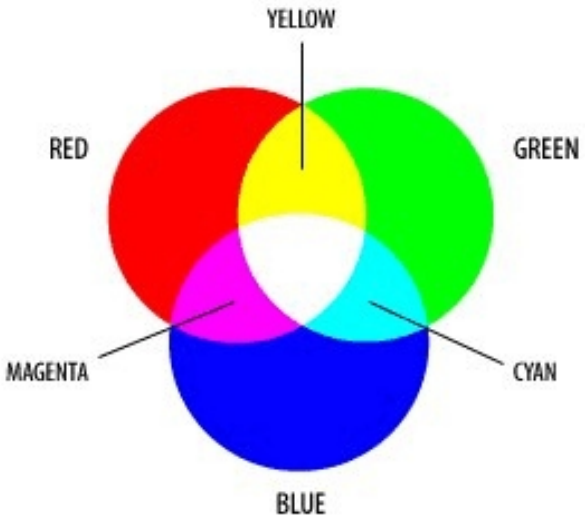
With the hexademical form the number of digits must be a multiple of three.

The procedure `ColorValue(s)` produces a string that is the decimal triple form of the color named by the string `s`.

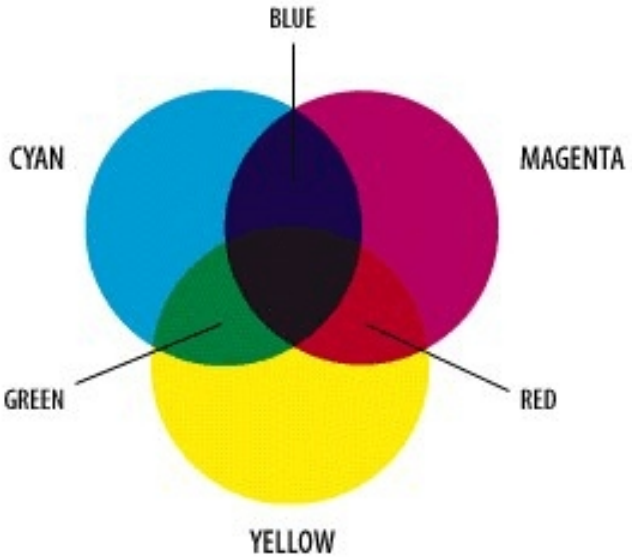
The sample program `color2a` is simply `color2` augmented to show the result of `ColorValue()`.

Color models

The RGB color model is *additive*—light from three different component colors contribute to the final value.



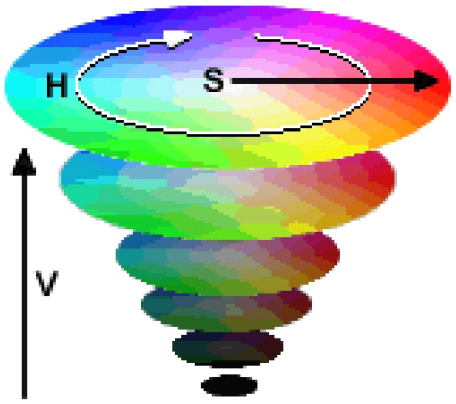
The CMY color model is commonly used when printing colors. It is called a subtractive model because ink is used to subtract colors from the image. The colors cyan, magenta, and yellow reflect no red, green, or blue light, respectively.



Diagrams from Adobe.com

Color models, continued

A third color model is HSV (Hue, Saturation, Value). "Value" is the brightness of the color. Here is a conical view of the HSV space from `www.wikipedia.org`:



The IPL program `colrpick` can be used to see the correspondence between the RGB and HSV models: