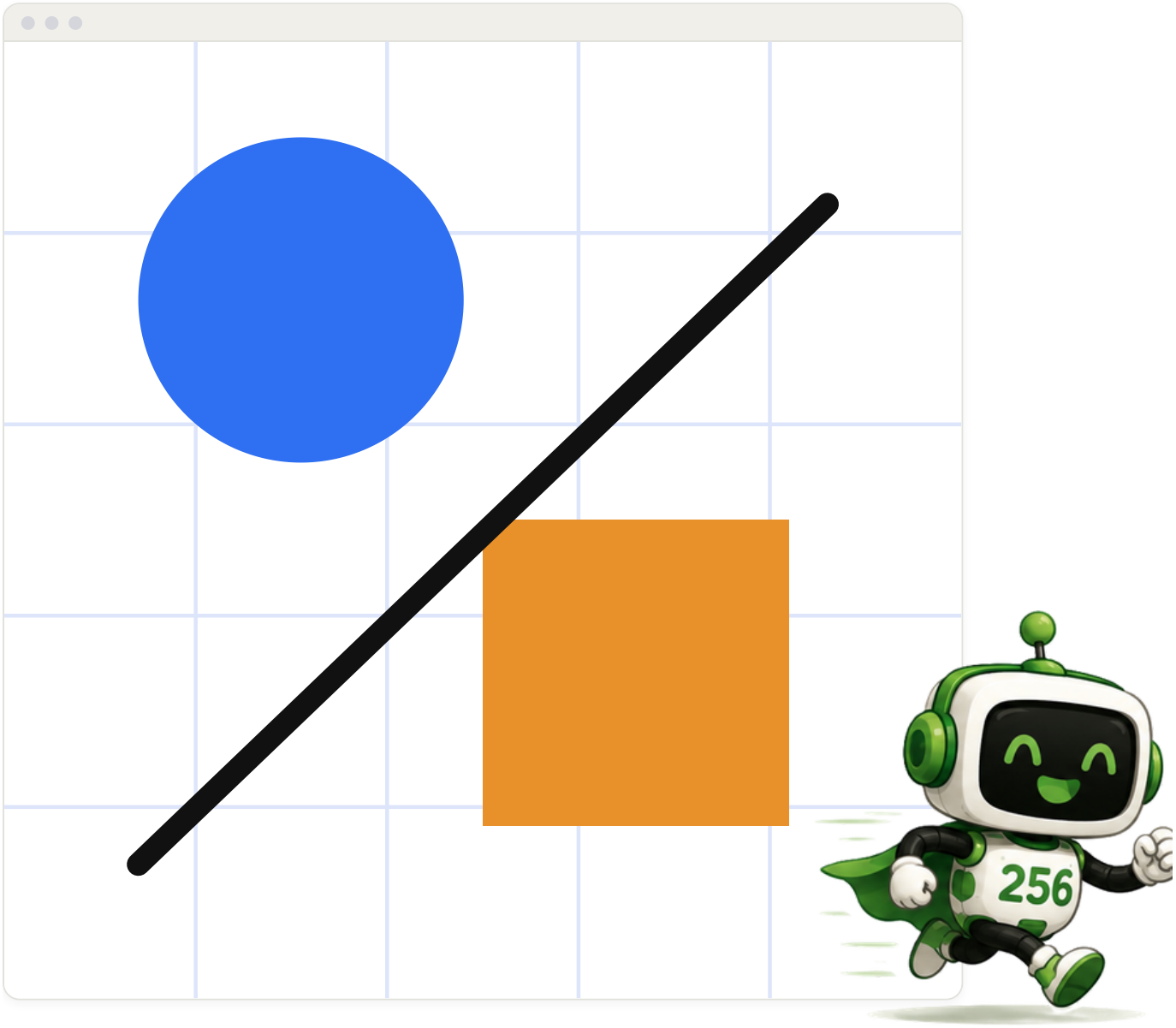
LEARN TO CODE · GRAPHICS

Chapter 2

Drawing Basic Shapes

Time to get graphical. You will draw rectangles, circles, lines and points in every colour, building pictures on a pixel canvas. As the programs grow, you will also learn to save your work and load it back to run or change later.



© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

Drawing rectangles and circles

Let’s start with a graphical program that draws a traffic light — a green light, specifically. Type it in and run it.

traffic_light.kbs BASIC-256

```
# traffic_light.kbs
# Show a traffic light and say a message.

clg

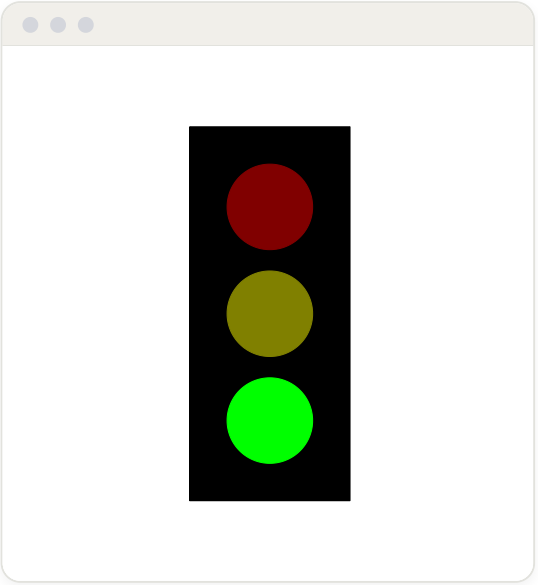
color black
rect 175,75,150,350

color darkred
circle 250,150,40

color darkyellow
circle 250,250,40

color green
circle 250,350,40

say "Green light. You may go."
```



says **“Green light. You may go.”**

Program 9 · Traffic Light

Let’s walk through it line by line. The first two lines are **remark** (comment) statements. A remark is a note for the programmer that BASIC-256 ignores completely — a good place to record the program’s name, what a tricky block does, or who wrote it.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
#
rem
```

Both `#` and `rem` begin a **remark**. Everything after them on the line is ignored by the computer, so you can write notes to yourself.

On the next line is `clg`. It works like `cls` from Chapter 1, but it clears the **graphics** output area instead of the text.



NEW CONCEPT

```
clg
clg color_name
clg rgb( red, green, blue )
clg hsv( hue, saturation, value )
clg "svg_color_name"
```

`clg` erases the graphics area so you have a clean place to draw. Add a colour after it — a name, an `rgb()` or `hsv()` mix, or an SVG colour name in quotes — to fill the whole window with that colour.

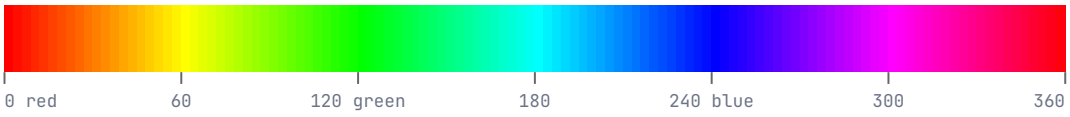
The `color` lines tell BASIC-256 which colour to use for the next drawing action. There are three ways to say which colour you want: pick one of the **eighteen named colours**, mix your own with numbers using **RGB** or **HSV**, or write one of the many **SVG colour names** in quotes.

Mixing with RGB

The first numeric method, `rgb(red, green, blue)`, mixes the three primary colours of light. Keep each value between **0 and 255**. Zero means none of that light; 255 means as bright as possible. So `rgb(255,255,255)` is white (all light) and `rgb(0,0,0)` is black (no light). This is called an **RGB triplet**.

Mixing with HSV

The second numeric method, `hsv(hue, saturation, value)`, describes a colour the way people often think about it. The **hue** is an angle around the colour wheel from **0 to 360** degrees — 0 is red, 120 green, 240 blue and 360 is red again. **Saturation** (how strong the colour is) and **value** (how bright it is) are percentages from **0 to 100**. So `hsv(120,100,50)` is a darker green, and `hsv(0,0,50)` is a mid grey whatever the hue. HSV is handy when you want to walk through colours: counting the hue from 0 to 360 goes through the whole rainbow with a single number.



`hsv(hue, 100, 100)` as the hue goes from 0 to 360.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SVG colour names

Finally, put a colour name **between quotes** and BASIC-256 understands the 147 standard SVG (web) colour names, such as `"tomato"`, `"skyblue"` or `"mediumaquamarine"`. You can even give a web-style hex code such as `"#ff8800"`.

`"tomato"` `"gold"` `"skyblue"` `"orchid"`
`"seagreen"` `"chocolate"` `"navy"` `"mediumaquamarine"`

Watch out: a quoted name follows the SVG standard, which is not always the same as BASIC-256's own named colour. `green` is the bright 0,255,0, but `"green"` is the darker 0,128,0 — and `"grey"` is 128,128,128 rather than 164,164,164.



NEW CONCEPT

```
color color_name
color rgb( red, green, blue )
color hsv( hue, saturation, value )
color "svg_color_name"
```

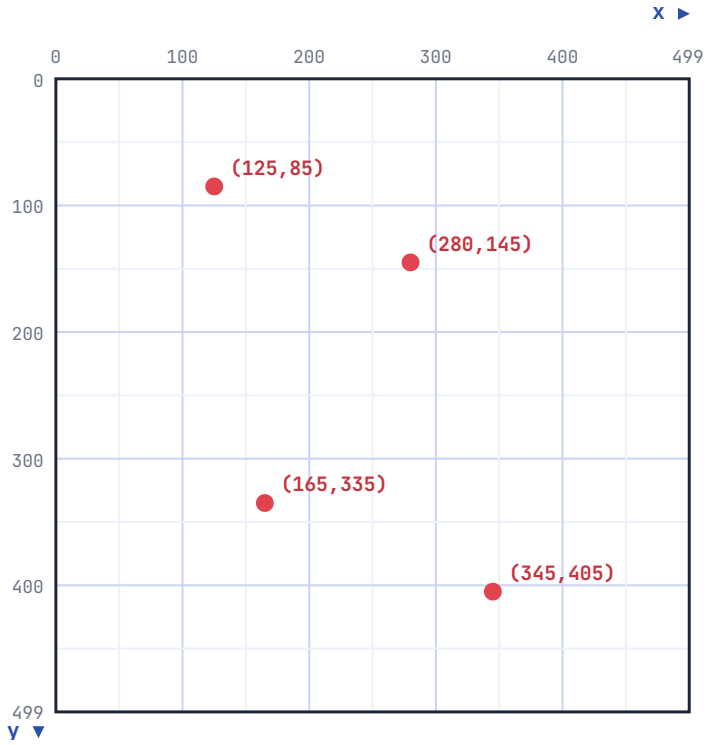
`color` sets the colour drawn next (British spelling `colour` works too). Follow it with a name from the table below, mix one with `rgb()` or `hsv()` to reach over 16 million colours, or give an SVG colour name in quotes.

THE EIGHTEEN NAMED COLOURS

black	0, 0, 0	white	255, 255, 255
red	255, 0, 0	darkred	128, 0, 0
green	0, 255, 0	darkgreen	0, 128, 0
blue	0, 0, 255	darkblue	0, 0, 128
cyan	0, 255, 255	darkcyan	0, 128, 128
purple	255, 0, 255	darkpurple	128, 0, 128
yellow	255, 255, 0	darkyellow	128, 128, 0
orange	255, 102, 0	darkorange	170, 51, 0
grey/gray	164, 164, 164	darkgrey/darkgray	128, 128, 128

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

The graphics area is **500 pixels wide by 500 pixels high** by default. A pixel is the smallest dot the screen can show. The top-left corner is the origin `(0,0)` and the bottom-right is `(499,499)`. Every point is two numbers: **x** (how far across) and **y** (how far down). Mathematicians call this the **Cartesian coordinate system**.

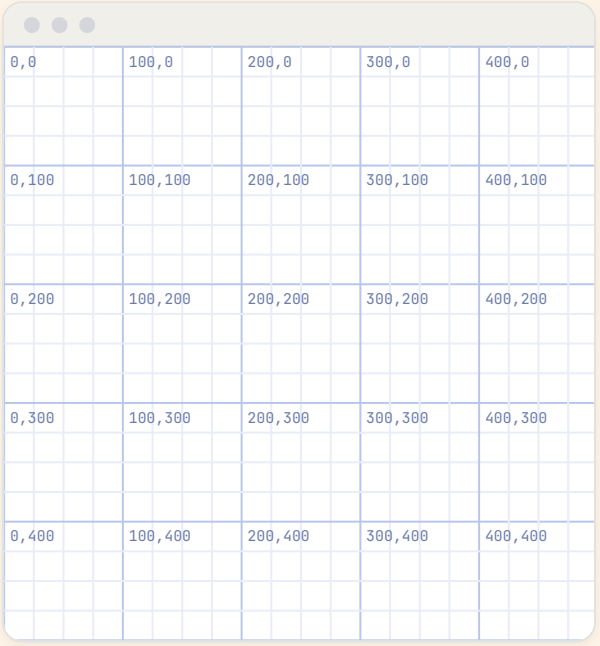


The Cartesian grid of the 500 × 500 graphics area. x counts pixels across, y counts pixels down from the top-left origin (0,0).

EXPLORE

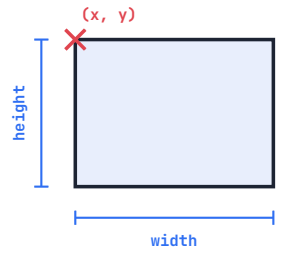
Turn on **View ▸ Graphics Window Grid Lines** in the editor to overlay a coordinate grid on the drawing area — handy for lining up your shapes.





© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

The `rect` statement draws a rectangle. It takes four numbers: how far over the left edge is, how far down the top edge is, and then the width and height — all in pixels.



`rect x, y, width, height` — the first two numbers place the top-left corner; the last two set the size, in pixels.

In the traffic-light program, the rectangle starts at `(175,75)` and is 150 pixels wide and 350 tall.

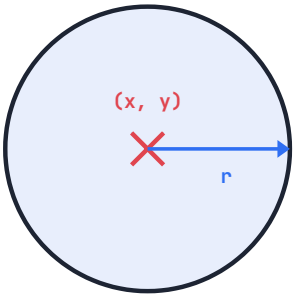


NEW CONCEPT

```
rect x, y, width, height
```

`rect` draws a filled rectangle in the current colour. The first two numbers set the top-left corner; the last two set the width and height.

The three `circle` lines draw the lights. `circle` takes three numbers: the x and y of the centre, and the radius in pixels.



`circle x, y, radius` — a filled circle centred on (x, y) with the given radius, in pixels.



NEW CONCEPT

```
circle x, y, radius
```

`circle` draws a filled circle in the current colour, centred on (x, y) with the radius you give.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

More programs with circles and rectangles

Here are two more programs using `clg`, `color`, `rect` and `circle`. Type them in and change them — make a frowning face, an alien, or someone you know.

rectanglesmile.kbs BASIC-256

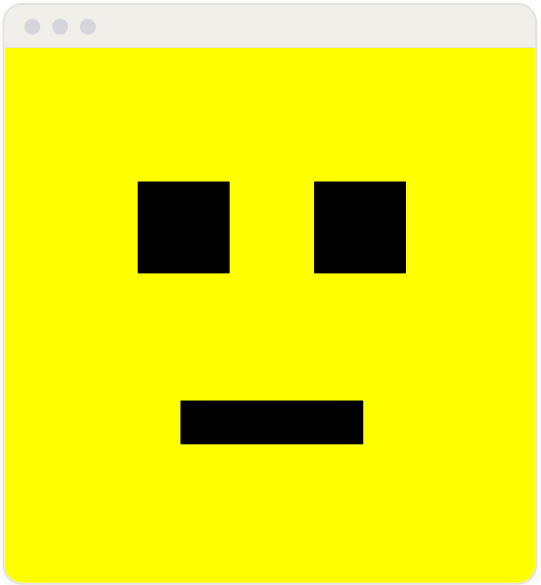
```
# rectanglesmile.kbs

# make the screen yellow
clg yellow

# draw the mouth
color black
rect 165,330,170,40

# put on the eyes
color black
rect 125,125,85,85
rect 290,125,85,85

say "Hello."
```



says **“Hello.”**

Program 10 · Face with Rectangles

The next one builds a rounder face. It draws a big black circle for the mouth, then covers the top of it with a yellow circle to leave a smile.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

circlesmile.kbs BASIC-256

```
# circlesmile.kbs

# clear the screen
clg white

# draw the face
color yellow
circle 250,250,250

# draw the mouth: a big black circle, then cover
# the top part with yellow to leave a smile
color black
circle 250,335,115
color yellow
circle 250,250,115

# draw the eyes
color black
circle 165,165,50
circle 335,165,50
```



Program 11 · Smiling Face with Circles

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Saving your program and loading it back

Now that programs are getting longer, you'll want to save them. Use the **Save** button on the toolbar or **File ▸ Save**. For a new program it asks for a file name; otherwise it replaces the file with your changes.

To keep the old version and store a copy under a new name, use **File ▸ Save As**. To reopen a saved program later, use the **Open** button or **File ▸ Open**.

Opening programs you didn't write

Sooner or later you will open a program from a friend, a forum or a download. A program can do much more than draw — it can read and write files, run other software or talk over the network — so BASIC-256 has safety measures that stop a program you open from quietly harming your computer:

Files stay in their folder. A program may freely read and write files in the folder it was loaded from (and the folders inside it). If it tries to touch anything *outside* that folder — with `open`, `kill`, `mkdir`, `imgsave`, `dbopen`, or even a file named inside a database command — BASIC-256 stops and shows you exactly which file it wants, with the choices **Don't allow**, **Allow once** and **Allow for this run**. There is no "never ask again" box, so any permission ends when the program stops. A file you pick yourself in a file dialog counts as already allowed.

No running other programs. The `system` statement, which can start other programs or commands, is switched **off** by default, and so are `portin` and `portout`, which talk directly to hardware.

Network stays at home. A program that listens for network connections with `netlisten` only accepts them from your own computer (127.0.0.1) unless you allow network access.

Settings stay private. A program can only read and change its *own* saved settings, not those of other programs.

No mass deletes. The `rmdir` statement, which could wipe a folder and everything in it, has been removed for good.

The browser version is sandboxed. BASIC-256 in a web browser runs inside the browser's sandbox and cannot reach the files on your computer at all.



STAY SAFE

Each of these can be changed in **Preferences ▸ Advanced** — but only loosen them for a program you trust. A good habit: read through a program you were given *before* you press Run, and if BASIC-256 asks about a file you didn't expect, choose **Don't allow**.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

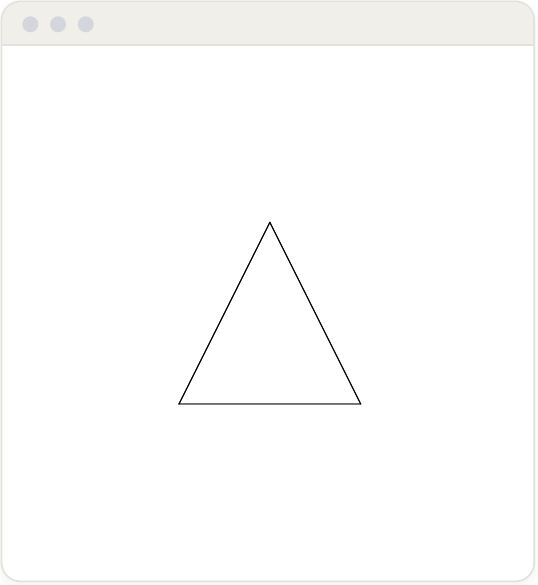
Drawing with lines

The `line` statement draws a one-pixel line, in the current colour, from one point to another. Program 12 draws a triangle.

triangle.kbs — BASIC-256

```
# triangle.kbs - draw a triangle

clg
color black
line 250, 165, 165, 335
line 165, 335, 335, 335
line 335, 335, 250, 165
```



Program 12 · Draw a Triangle



NEW CONCEPT

```
line start_x, start_y, finish_x, finish_y
```

`line` draws a straight line from the start point to the finish point using the current colour.

With enough lines you can draw anything. This next program stacks two squares and connects their four pairs of corners to draw a cube.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

cube.kbs BASIC-256

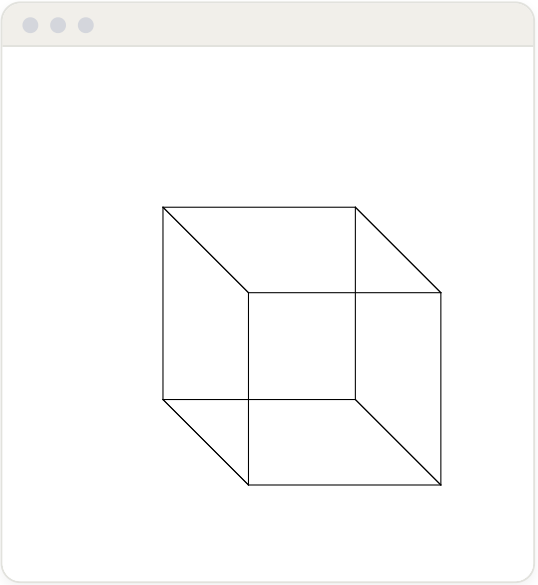
```
# cube.kbs - draw a cube

clg
color black

# draw back square
line 230, 230, 230, 410
line 230, 410, 410, 410
line 410, 410, 410, 230
line 410, 230, 230, 230

# draw front square
line 150, 150, 150, 330
line 150, 330, 330, 330
line 330, 330, 330, 150
line 330, 150, 150, 150

# connect the corners
line 150, 150, 230, 230
line 150, 330, 230, 410
line 330, 330, 410, 410
line 330, 150, 410, 230
```



Program 13 · Draw a Cube

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 5

Line width and shape outlines

By default a line is one pixel wide. The `penwidth` statement changes how thick lines — and the outlines of shapes — are drawn. This program also shows a richer `color` statement and the special colour `clear`.

shapeoutline.kbs — BASIC-256

```
# shapeoutline.kbs
# draw shapes with an outline

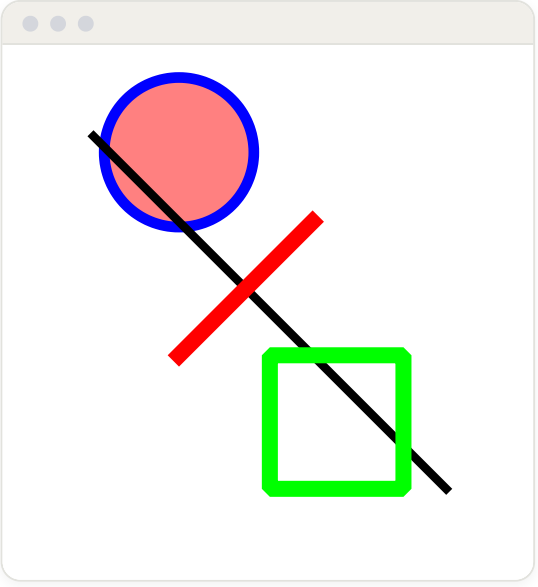
clg

# draw a pink circle with a blue outline
penwidth 10
color blue, rgb(255,128,128)
circle 165,100,70

# draw a thick black line
color black
penwidth 8
line 85,85,415,415

# draw another thick red line
color red
penwidth 15
line 290,165,165,290

# draw a green square that is not filled
color green, clear
penwidth 15
rect 250,290,125,125
```



Program 14 · Penwidth and Shape Outline

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



`penwidth n` sets how thick lines and shape outlines are drawn.



NEW CONCEPT

```
penwidth n
```

`penwidth` sets the width of the drawing pen — both the thickness of a line and the outline around a shape.



NEW CONCEPT

```
color pen_color, fill_color
```

Give `color` two colours and the pen (outline) and fill are set separately. With one colour, both are the same. Each colour can be a name, an `rgb()` or `hsv()` mix, or an SVG name in quotes.



NEW CONCEPT

```
clear
```

The special colour `clear` means “don’t fill.” Set the fill colour to `clear` to draw just the outline of a shape.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 6

Setting individual points

The last graphics statement in this chapter is `plot`, which sets a single point. On their own these dots are tiny — later you'll draw thousands of them to make detailed images.

pointplot.kbs BASIC-256

```
# pointplot.kbs - use plot to draw points

clg
color red
penwidth 34
plot 200,200

color orange
penwidth 21
plot 228,228

color yellow
penwidth 13
plot 245,245

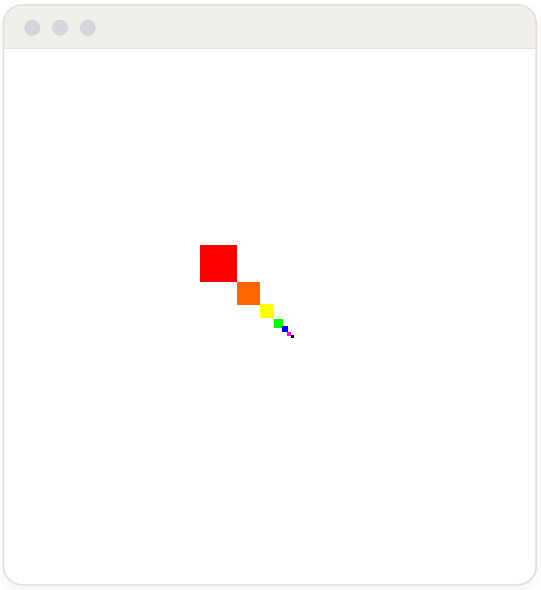
color green
penwidth 8
plot 256,256

color blue
penwidth 5
plot 262,262

color purple
penwidth 3
plot 266,266

color black
penwidth 2
plot 269,269
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



Program 15 · Use Plot to Draw Points



NEW CONCEPT

```
plot x, y
```

`plot` draws a point at (x, y) in the current pen colour and pen width.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 7

Choosing your own coordinates

Pixels are how the screen counts, but they are not always how *you* want to count. On graph paper in a maths lesson the point `0,0` sits in the middle, x is negative on the left, and y gets **bigger as you go up** — the opposite of the graphics area, where y grows as you go down. You could work out the pixel for every point yourself, but BASIC-256 can do the arithmetic for you.

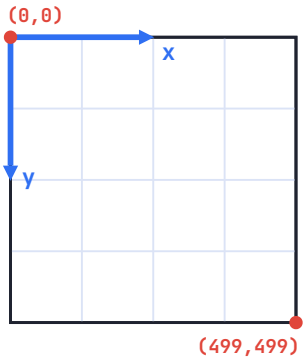


NEW CONCEPT

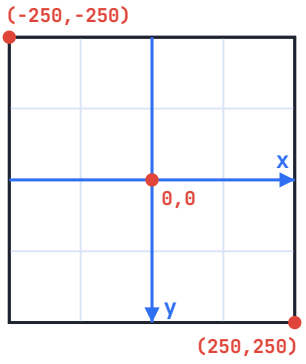
```
window x1, y1, x2, y2
window
```

`window` gives the corners of the graphics area numbers of your own choosing: **x1, y1** becomes the **top-left** corner and **x2, y2** the **bottom-right**. Every drawing statement after it uses those numbers instead of pixels. `window` on its own goes back to plain pixels.

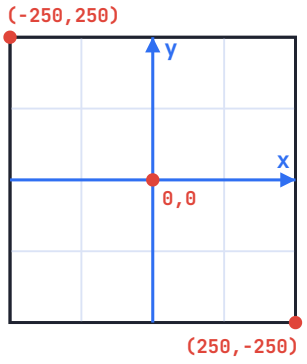
Because you name the top-left corner first, the **order** of the numbers decides which way each axis runs. Put the bigger y first and y counts upwards, like graph paper:



`no window`
pixels, y goes down



`window -250,-250,250,250`
0,0 in the middle, y goes down



`window -250,250,250,-250`
0,0 in the middle, y goes up

The same 500 × 500 graphics area under three different windows. The red dots are the two corners you name.

Nothing on the screen moves when you run `window` — the pixels are still there. You have only changed the names the corners go by, and so the names of every point between them.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

Program 16 sets up graph paper that runs from −10 to 10 both ways, with y going up, and draws on it. Every number in it is a graph-paper number; BASIC-256 turns each one into a pixel as it draws.

mathgraph.kbs BASIC-256

```
# mathgraph.kbs - graph paper with 0,0 in the middle
clg white
# left edge -10, top 10, right edge 10, bottom -10
window -10, 10, 10, -10

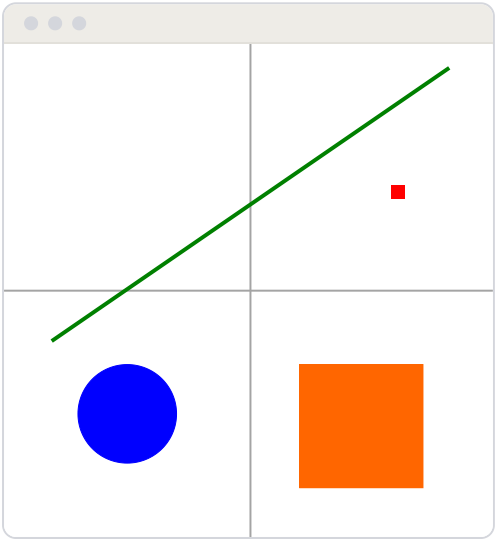
# the two axes cross at 0,0
color grey
penwidth 2
line -10, 0, 10, 0
line 0, -10, 0, 10

# a blue circle down and to the left
color blue
penwidth 1
circle -5, -5, 2

# an orange square; its x,y is now the BOTTOM-left
color orange
rect 2, -8, 5, 5

# a green line climbing up to the right
color darkgreen
penwidth 4
line -8, -2, 8, 9

# a big red dot up and to the right
color red
penwidth 14
plot 6, 4
```



With this window one unit is 500 ÷ 20 = 25 pixels, so the red dot at `6,4` lands 150 pixels right of the middle and 100 pixels **up**.

Program 16 · Graph Paper with WINDOW



TRY IT

Change the window line to `window -10, -10, 10, 10` — the same numbers with the y values swapped — and run the program again. The whole picture flips upside down, because now y counts downwards. Then try `window 0, 10, 20, -10`: `0,0` moves to the middle of the left edge.

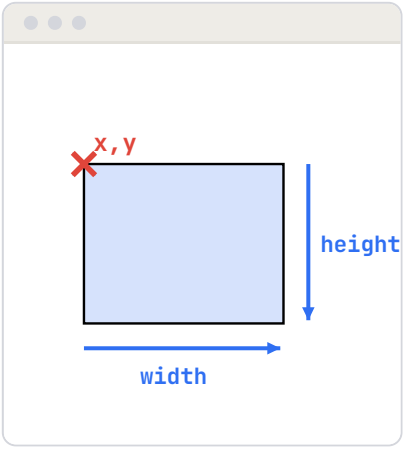
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

How the drawing statements follow the window

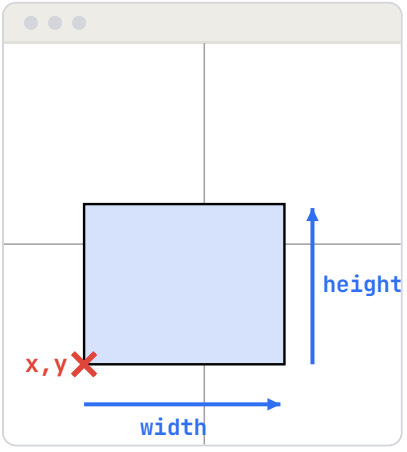
Once a window is set, every point you give is a window point, and every size — a width, a height, a radius — is measured in window units too. That has a few effects worth knowing about.

`line` **and** `plot` — each end of a line and each point is simply moved to where that window point is on the screen.

`rect` — the width and height are window units, and the rectangle always grows towards **bigger** x and **bigger** y. Without a window bigger y is further down, so `x, y` is the top-left corner. With y going up, bigger y is further *up*, so `x, y` becomes the **bottom-left** corner.



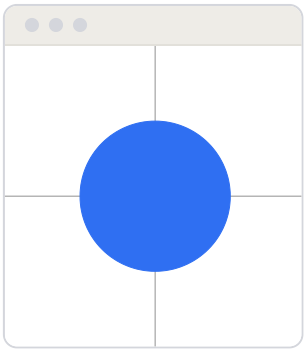
`rect 100,150,250,200`
no window: x,y is the top-left



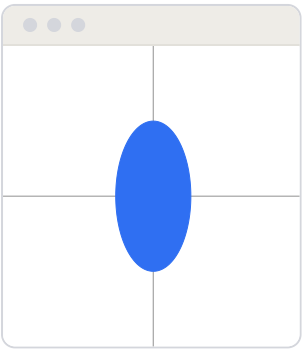
`window -10,10,10,-10` then `rect -6,-6,10,8`
y goes up: x,y is the bottom-left

`circle` — the centre is a window point and the radius is window units, measured across *and* down.

Watch out: keep the window the same shape as the graphics area. The area is 500 × 500, a square, so give the window the same distance across as down — 20 and 20, or 500 and 500. If the window is 40 units across but only 20 down, a unit across is squashed to half the width of a unit down, and every circle comes out as an oval.



`window -10,10,10,-10`
20 across, 20 down: round



`window -20,10,20,-10`
40 across, 20 down: squashed

Both programs then run `circle 0,0,5`. On the right a unit across is only half as wide as a unit down, so the circle is drawn as an oval.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

`penwidth` stays in **pixels**. `penwidth 3` is always 3 pixels, even when one unit is 25 pixels wide, so lines don't get fatter just because you zoomed in. The dot drawn by `plot` is penwidth pixels across for the same reason.

`clg` still clears the whole graphics area, whatever the window.

No zero-size windows. A window must have some width and some height: `window 0, 0, 0, 10` stops the program with *WINDOW needs a width and a height that are not zero*.

Every run starts in pixels. A window lasts until you change it or the program ends, and the next program always starts without one. If you change the size of the graphics area later, the window stretches to fit the new size.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

The Talking Face

Every chapter ends with a bigger program to type in and play with, using only what you've learned so far. This one makes a face talk: before it says each word, the lower half is redrawn with a new mouth shape for a rough little animation.

talkingface.kbs — BASIC-256

```
# talkingface.kbs

color yellow
rect 0,0,500,500
color black
rect 125,125,85,85
rect 290,125,85,85

# erase old mouth
color yellow
rect 0,250,500,250
# draw new mouth
color black
rect 210,290,80,165
say "i"

color yellow
rect 0,250,500,250
color black
rect 165,335,170,80
say "am"

color yellow
rect 0,250,500,250
color black
rect 210,290,80,165
say "glad"

color yellow
rect 0,250,500,250
color black
rect 210,335,80,80
say "you"

color yellow
rect 0,250,500,250
color black
rect 165,335,170,80
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
say "are"

color yellow
rect 0,250,500,250
color black
rect 210,335,80,80
say "my"

# draw a whole new face with a round smile
color yellow
rect 0,0,500,500
color black
circle 250,290,165
color yellow
circle 250,250,165
color black
rect 125,125,85,85
rect 290,125,85,85
say "friend"
```



says **“friend”**

Program 17 · Big Program — Talking Face

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

```
r e t a n i d r o o c
e e a r a e l c r u m
m e l c r i c e s s r
a c k v c e c c u y o
r y j l n t i i t p l
k a g t a h d h w l o
q n e n p a g i q o c
y r g a r i d p j t e
c l r e e t s a v e h
e g p h h u e n i l d
j r x p e n w i d t h
```

Find: center, circle, clear, clg, color, coordinate, cyan, graphics, height, line, penwidth, plot, radius, rectangle, remark, save, width



Problems

Try these — they only use what this chapter covered. Remember the graphics area is 500 × 500.

1. Type in **Program 11: Smiling Face with Circles** and change it to show **Mr. Yuck**. You may need `penwidth` to make the lines thicker.

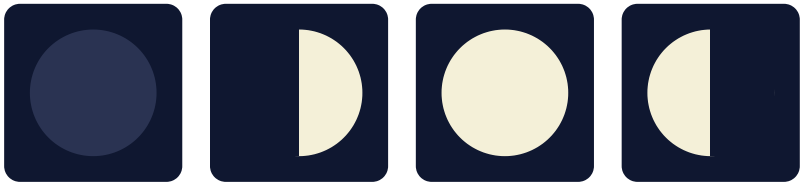


2. Write a program that draws a square and says “square”. Then clear the screen, draw a circle, and say “circle”. Finally clear the screen, draw several lines in any pattern you like, and say “lines”.

3. Use colours, lines and circles to draw an archery target with an arrow in the centre. Once the arrow is drawn, make the computer say “Bullseye!”. Try `hsv()` to pick the ring colours.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

4 Write a program that draws each phase of the moon — new, first quarter, full and third quarter — and speaks its name. Hint: draw the moon as a circle, then draw a rectangle over the part you don't want.



New Moon First Quarter Full Moon Third Quarter

5 Use `window -10, 10, 10, -10` so that 0,0 is in the middle of the screen and y goes up. Draw a snowman standing on the bottom edge (y = −10) against a cyan sky: three white circles stacked on top of each other, getting smaller as they go up, with two black eyes. Then make the computer say "Snowman". Hint: with y going up, each ball's centre has a bigger y than the one below it.
Bonus: change only the window line to `window -10, -10, 10, 10`. What happens to your snowman, and why?



Snowman

Chapter 2 · Drawing Basic Shapes. Modernised layout of the BASIC-256 beginner's guide, updated for the 500 × 500 graphics area, HSV and SVG colours, and the file-safety features of BASIC-256 2.3. Every program figure in this chapter is drawn from the code beside it, using the BASIC-256 drawing commands. Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence. Mascot: BitBot.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout