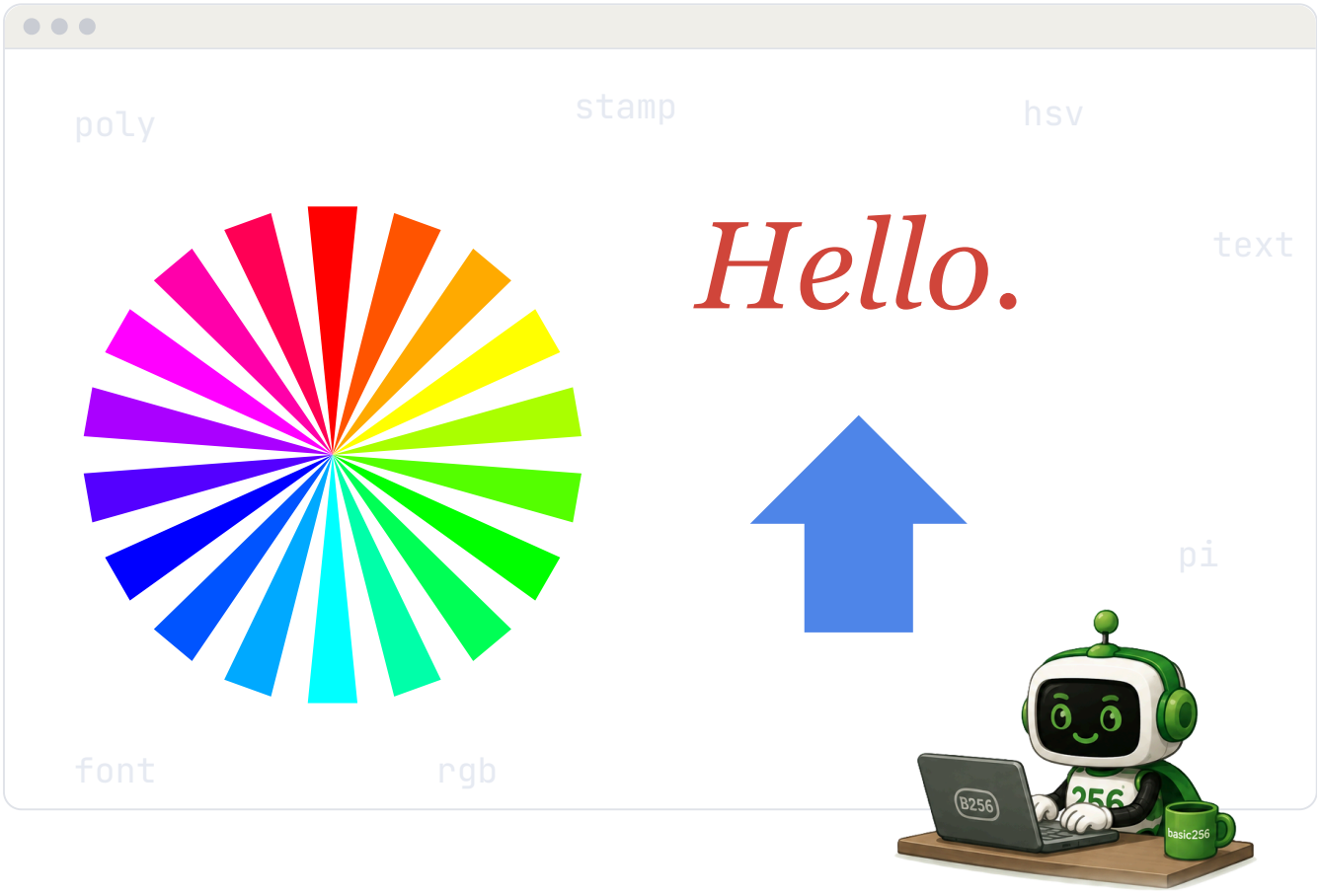
LEARN TO CODE · CUSTOM GRAPHICS

Chapter 9

Custom Graphics — Creating Your Own Shapes

In this chapter you draw colourful words on the graphics output, change its size, and make shapes of your own: polygons with as many corners as you like, and stamps that you can move, resize and turn. Along the way you meet a new way to measure angles, and colours you can see through.



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

SECTION 1

Text on the graphics output

With `font` and `text` you can put words and numbers on the graphics output, in many styles.

You know **print** from Chapter 1: it writes in the text output area. To put words on the *graphics* output instead, choose a font with **font** and draw with **text**.

graphichello.kbs BASIC-256

```
# graphichello.kbs - text on the graphics output
clg
color red
font "Tahoma", 40, 700
text 100, 60, "Hello."
font "Impact", 40, 400
text 100, 160, "Hello."
font "Courier New", 40, 400
text 100, 260, "Hello."
font "Georgia", 40, 400, true
text 100, 360, "Hello."
```

GRAPHICS OUTPUT



Program 54 · Hello on the Graphics Output



NEW CONCEPT

```
text x, y, expression
```

text draws the value of the expression — a string or a number — on the graphics output, in the current colour. Its **top left corner** is at *x*, *y*. It uses the font set by the last **font** statement.

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



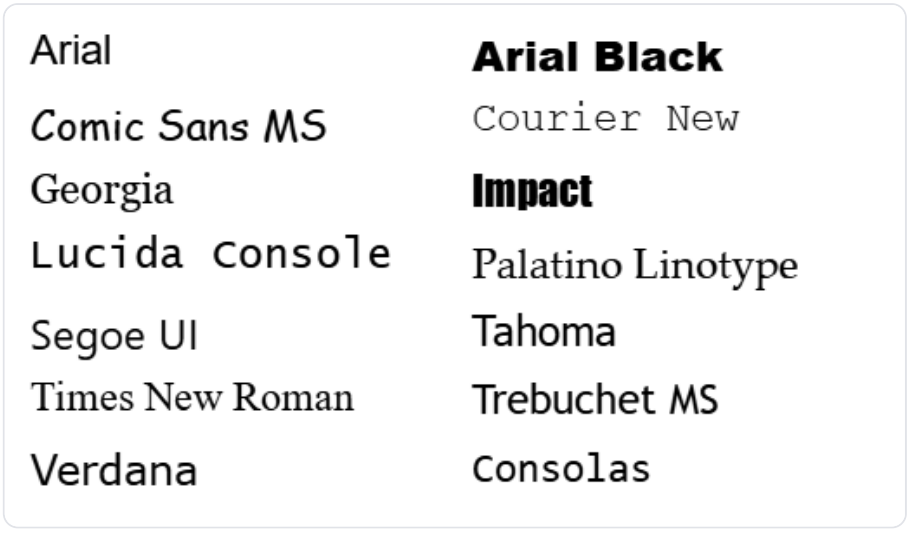
NEW CONCEPT

```
font name, size, weight
font name, size, weight, italic
```

font chooses the font for the next **text** statements.

name	The name of a font installed on the computer, as a string.
size	The height in points. There are 72 points in an inch.
weight	How thick the letters are, from 1 to 1000: **400** is normal, **700** bold and **900** extra bold. Most fonts only have normal and bold, so everything up to about 500 looks normal.
italic	Optional: **true** for slanted letters.

Older books use different weights. Earlier versions of BASIC-256 counted the weight from 1 to 100, with 50 for normal and 75 for bold. BASIC-256 2.3 uses the scale from 1 to 1000, so a weight of 75 or 100 now gives ordinary, not bold, letters. Use 700 for bold.



Some fonts that come with Windows, drawn by BASIC-256. On other computers different fonts are installed; if a font isn't there, BASIC-256 uses a similar one.

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

SECTION 2

Resizing the graphics output

The graphics output is 500 × 500 pixels — unless you change it with `graphsize`.

Every program starts with a graphics output of 500 by 500 pixels. That is plenty for most programs, but sometimes you want a different size: a wide strip for a banner, or a small square. The **graphsize** statement changes it, and the **graphwidth** and **graphheight** functions tell you how big it is at the moment.

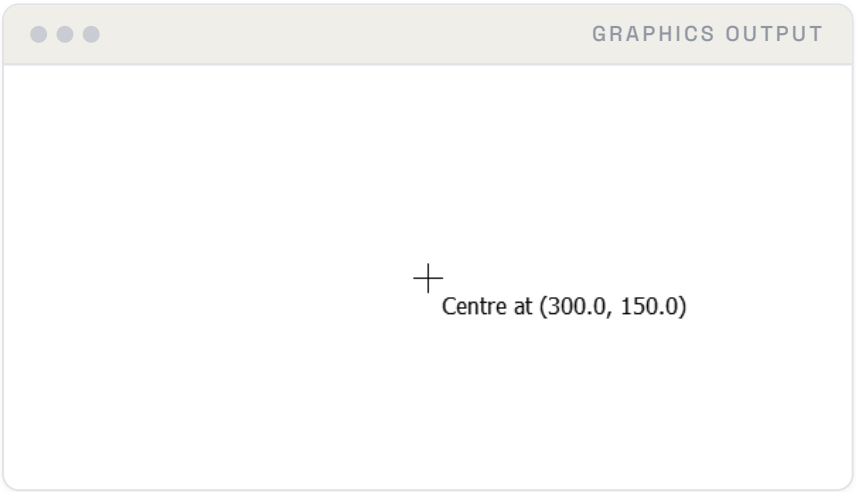
resizegraphics.kbs BASIC-256

```
# resizegraphics.kbs - change the size of the graphics output
graphsize 600, 300
xcenter = graphwidth / 2
ycenter = graphheight / 2

color black
line xcenter, ycenter - 10, xcenter, ycenter + 10
line xcenter - 10, ycenter, xcenter + 10, ycenter

font "Tahoma", 12, 400
message = "Centre at (" + xcenter + ", " + ycenter + ")"
text xcenter + 10, ycenter + 10, message
```

Program 55 · Resize the Graphics Output



Using **graphwidth** and **graphheight** instead of 600 and 300 means the cross stays in the middle even if you change the size in the first line. Dividing gives a floating-point number, so the message shows 300.0 and 150.0.

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



NEW CONCEPT

```
graphsize width, height
graphsize width, height, scale
graphwidth
graphheight
```

graphsize sets the width and height of the graphics output in pixels. It lasts until the program ends: the next run starts at 500 × 500 again. An optional third number *scale* shows each pixel bigger on screen — `graphsize 50, 50, 8` gives chunky pixel art — without changing the coordinates you draw with.

graphwidth and **graphheight** give the current width and height. The brackets are optional.

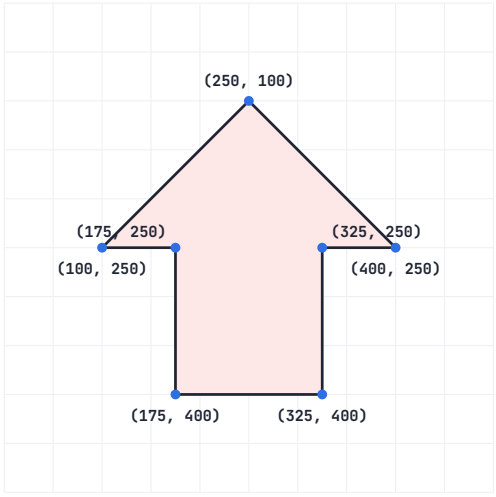
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Drawing your own polygon

The `poly` statement draws a shape with as many straight sides as you like.

You can already draw rectangles and circles. For any other shape with straight sides — a **polygon** — use **poly**. Let's draw a big red arrow in the middle of the graphics output. First sketch it on squared paper and write down the coordinates of every corner.



The arrow on a 500 × 500 grid: one line every 50 pixels.

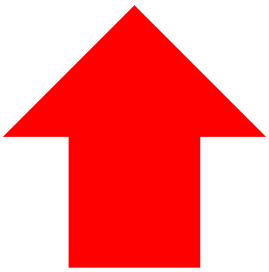
Now start at the tip and go round clockwise, writing down the *x* and *y* of each corner. These fourteen numbers, between curly brackets, are a **list**.

bigredarrow.kbs — BASIC-256

```
# bigredarrow.kbs - a polygon with seven corners
clg
color red
poly {250, 100, 400, 250, 325, 250, 325, 400,
      175, 400, 175, 250, 100, 250}
```

Program 56 · Big Red Arrow

GRAPHICS OUTPUT



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



NEW CONCEPT

```
poly {x1, y1, x2, y2, x3, y3 …}
```

poly draws a filled polygon. The list holds the corners: the numbers are taken two at a time, as the *x* and *y* of one corner, and the last corner is joined back to the first. A long list may go on to the next line after a comma. (In a later chapter you will store lists in arrays.)

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

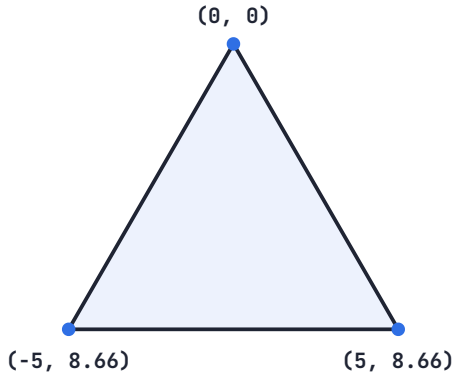
SECTION 4

Stamping a polygon

A `stamp` is a polygon you can put anywhere, make bigger or smaller, and turn.

With **poly** the shape is always in the same place: to move it you would have to change every number. The **stamp** statement solves that. You describe the shape once, around its own point (0, 0), and tell **stamp** where on the screen that point should go — and, if you like, how big to make the shape and how far to turn it.

Draw an equilateral triangle (all sides the same length) with the point (0, 0) at the top and sides 10 units long.



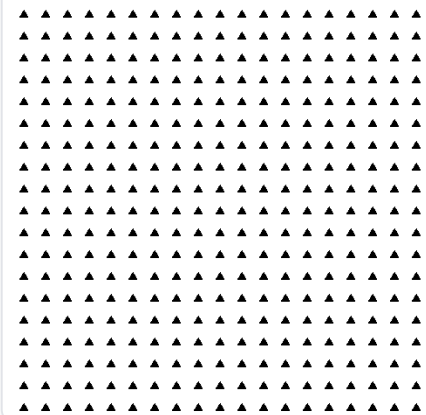
The triangle, with (0, 0) at the top. Its height is 8.66.

Program 57 uses the simplest form of **stamp**, inside two nested **for** loops, to fill the screen with triangles.

stamptriangle.kbs — BASIC-256

```
# stamptriangle.kbs - fill the screen with triangles
clg
color black
for x = 25 to 475 step 25
   for y = 25 to 475 step 25
      stamp x, y, {0, 0, 5, 8.66, -5, 8.66}
   next y
next x
```

GRAPHICS OUTPUT



Program 57 · Fill the Screen with Triangles

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



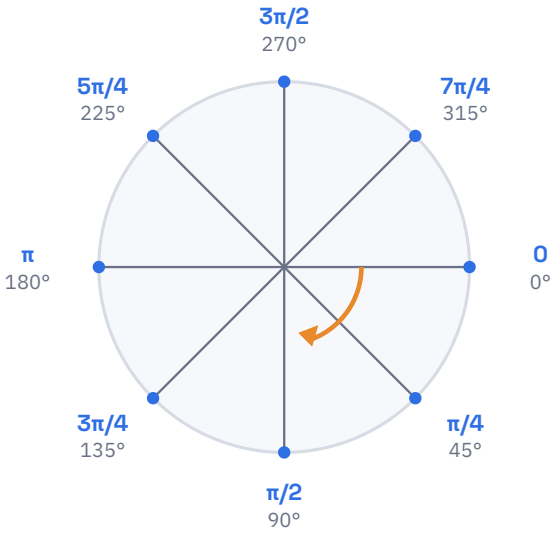
NEW CONCEPT

```
stamp x, y, {x1, y1, x2, y2 …}
stamp x, y, scale, {x1, y1, x2, y2 …}
stamp x, y, scale, rotate, {x1, y1, x2, y2 …}
```

stamp draws the polygon with its point (0, 0) at *x*, *y* on the screen. The optional *scale* resizes it: 1 is the size in the list, 2 twice as big, 0.5 half. The optional *rotate* turns it clockwise around (0, 0), by an angle measured in **radians**.

Measuring angles in radians

You probably measure angles in degrees: a full turn is 360°. BASIC-256 measures angles in **radians**, where a full turn is 2π — a little more than 6.28. A right angle is π/2 and turning round to face the other way is π.



How far **stamp** turns for each angle. Because *y* counts downwards on the screen, the angle grows clockwise.



NEW CONCEPT

```
pi
radians(degrees)
degrees(radians)
```

pi is the number π, 3.14159..., so you don't have to remember it. A full turn is `2 * pi`.

To change degrees to radians, divide by 180 and multiply by π, or use **radians**: `radians(90)` is π/2. **degrees** goes the other way.

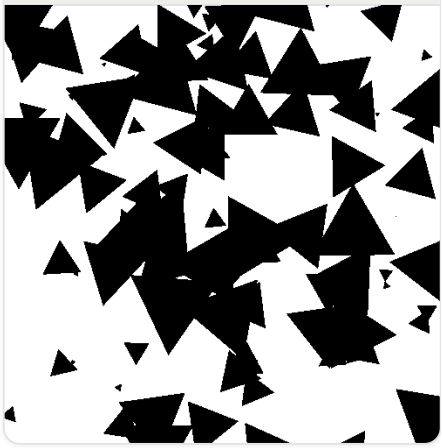
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

Program 58 stamps the same triangle 100 times at random places, each one a random size and turned by a random angle.

stamptriangle2.kbs BASIC-256

```
# stamptriangle2.kbs - random sizes and turns
clg
color black
for t = 1 to 100
   x = rand * graphwidth
   y = rand * graphheight
   s = rand * 10          # up to 10 times as big
   r = rand * 2 * pi      # turn up to a full circle
   stamp x, y, s, r, {0, 0, 5, 8.66, -5, 8.66}
next t
```

GRAPHICS OUTPUT



Program 58 · One Hundred Random Triangles

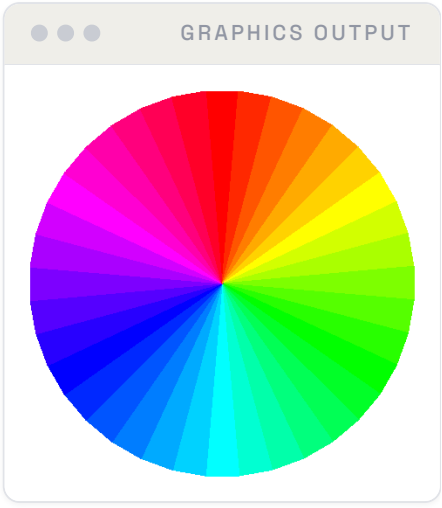
Angles in degrees are often easier to count with. Program 59 turns a long thin triangle 10° at a time, all the way round, and colours each one with **hsv** from Chapter 2, whose hue is also an angle on the colour wheel. So the angle of each triangle and its colour come from the same number.

colorwheel.kbs BASIC-256

```
# colorwheel.kbs - a colour wheel of stamped triangles
clg white
for d = 0 to 350 step 10
   color hsv(d, 100, 100)
   stamp 250, 250, 22, radians(d), {0, 0, -0.9, -10, 0.9, -10}
next d
```

Program 59 · Colour Wheel

GRAPHICS OUTPUT



The triangle is long and thin: its point (0, 0) is in the middle of the wheel and it reaches 10 units up, so with a scale of 22 it is 220 pixels long. **radians(d)** turns it to the angle *d*, and **hsv(d, 100, 100)** gives it the colour at that same angle — red at the top (0°), yellow at 60°, green at 120° and so on round the wheel.

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

SECTION 5

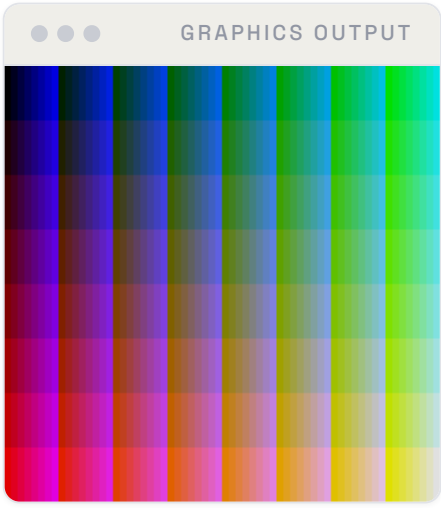
Sixteen million colours — and some you can see through

Mix your own colours with `rgb` or `hsv`, and make them transparent.

In Chapter 2 you met the two ways to mix a colour of your own. **rgb** adds red, green and blue light, each from 0 (none) to 255 (full). That gives 256 × 256 × 256 = 16,777,216 colours. **hsv** describes the same colours by hue (0 to 360 degrees round the colour wheel), saturation and value (both 0 to 100). Program 60 shows 512 of them, with three nested loops that count each part in steps of 32.

512colors.kbs — BASIC-256

```
# 512colors.kbs - show a few of the 16 million colours
graphsize 256, 256
clg
for r = 0 to 255 step 32
   for g = 0 to 255 step 32
      for b = 0 to 255 step 32
         color rgb(r, g, b)
         rect b / 8 + g, r, 4, 32
      next b
   next g
next r
```



graphsize 256, 256 makes the output smaller.

Program 60 · 512 of the 16 Million Colours

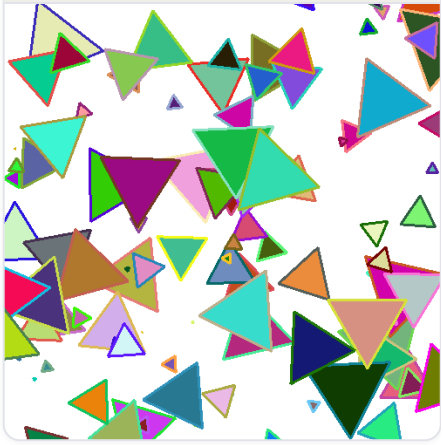
Because **rgb** and **hsv** give back a single number, you can keep a colour in a variable. Program 61 picks a random outline colour (the pen) and a random fill colour (the brush) for each triangle. **rgb** ignores the part after the decimal point, so `rand * 256` gives a whole range from 0 to 255.

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

stamptriangle3.kbs BASIC-256

```
# stamptriangle3.kbs - triangles in random colours
clg
penwidth 3
for t = 1 to 100
   x = rand * graphwidth
   y = rand * graphheight
   s = rand * 10
   r = rand * 2 * pi
   # a random colour for the outline (the pen)
   pen = rgb(rand * 256, rand * 256, rand * 256)
   # and one for the inside (the brush)
   brush = rgb(rand * 256, rand * 256, rand * 256)
   color pen, brush
   stamp x, y, s, r, {0, 0, 5, 8.66, -5, 8.66}
next t
```

GRAPHICS OUTPUT



Program 61 · 100 Triangles in Random Colours



TRY IT

Random **rgb** colours are often dull and greyish. Change the two colour lines to `pen = hsv(rand * 360, 100, 100)` and `brush = hsv(rand * 360, 100, 100)`: any hue, but always full strength and full brightness. Now every triangle is bright.



NEW CONCEPT

```
rgb(red, green, blue)
rgb(red, green, blue, alpha)
hsv(hue, saturation, value)
hsv(hue, saturation, value, alpha)
```

Both **rgb** and **hsv** take an optional fourth number, **alpha**: how solid the colour is. Watch out — the ranges differ. For **rgb** alpha goes from 0 (completely see-through, invisible) to 255 (solid). For **hsv** it is a percentage, from 0 to 100, like the other three. So `rgb(255, 0, 0, 128)` and `hsv(0, 100, 100, 50)` are the same half-transparent red. Without alpha a colour is solid.

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

transparent.kbs BASIC-256

```
# transparent.kbs - colours you can see through
clg white

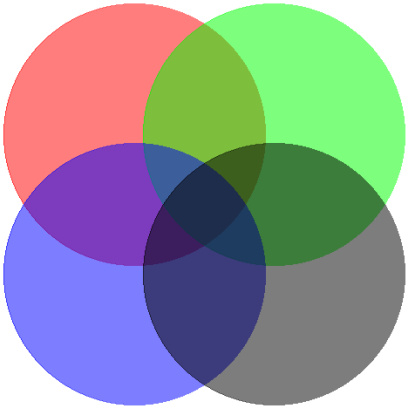
color rgb(255, 0, 0, 127)
circle 170, 170, 150

color rgb(0, 255, 0, 127)
circle 330, 170, 150

color rgb(0, 0, 255, 127)
circle 170, 330, 150

color hsv(0, 0, 0, 50)
circle 330, 330, 150
```

GRAPHICS OUTPUT



Program 62 · Transparent Circles

Where the circles overlap, you see both colours mixed. The fourth circle uses **hsv**: saturation 0 and value 0 is black, and alpha 50 makes it half transparent.

Remember that **rgb** and **hsv** are functions: they only work out a colour number. A line with nothing but `hsv(0, 0, 0, 50)` on it is a syntax error. Give the number to **color**, or keep it in a variable.



TRY IT

Give every circle a random see-through colour: change each **color** line to `color hsv(rand * 360, 100, 100, 50)`.

Make the triangles of Program 61 see-through too: add a fourth number to both colours, for example `pen = rgb(rand * 256, rand * 256, rand * 256, rand * 256)`, or `brush = hsv(rand * 360, 100, 100, 50)`.

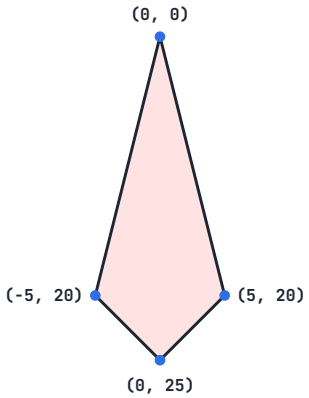
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

A Flower for You

Let's send flowers to somebody special. This program draws a flower by stamping one petal shape over and over, turning it a little each time.



The petal stamp. Its point (0, 0) goes in the middle of the flower.

aflowerforyou.kbs BASIC-256

```
# aflowerforyou.kbs - stamps make a flower
clg
cx = graphwidth / 2
cy = graphheight / 2

# the stem
color green
rect cx - 4, cy, 8, graphheight - cy

# eight big pink petals, 45 degrees apart
color rgb(255, 128, 128)
for d = 0 to 315 step 45
   stamp cx, cy, 5, radians(d), {0, 0, 5, 20, 0, 25, -5, 20}
next d

# ten small blue petals, 36 degrees apart
color hsv(240, 50, 100)
for d = 0 to 324 step 36
   stamp cx, cy, 2.5, radians(d), {0, 0, 5, 20, 0, 25, -5, 20}
next d

message = "A flower for you."
color darkyellow
font "Tahoma", 24, 400
text 20, 20, message
say message
```

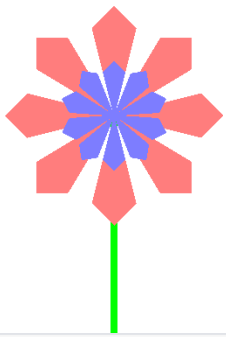
Program 63 · Big Program — A Flower for You

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

The eight big petals are 45° apart (8 × 45 = 360) and the ten small ones 36° apart (10 × 36 = 360). Counting the angle in degrees and changing it with **radians** makes sure each petal is drawn exactly once. The small petals use **hsv**: hue 240 is blue, and saturation 50 makes it a soft blue.

GRAPHICS OUTPUT

A flower for you.



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



PRACTICE

Exercises

WORD SEARCH

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

Find: alpha, blue, degrees, font, graphheight, graphsize, graphwidth, green, hsv, pi, point, polygon, radian, red, rgb, stamp, text, transparent, weight

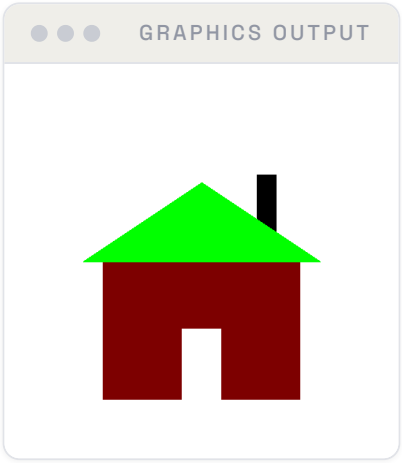
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



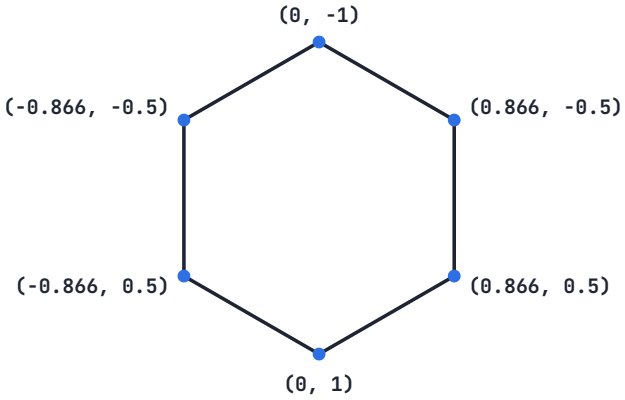
Problems

Try these — they only use what this chapter and the earlier ones covered.

1 Use **poly** and **rect** statements to draw a simple house like the one below, in any colours you like. Hint: draw the chimney before the roof, so the roof covers its bottom end.



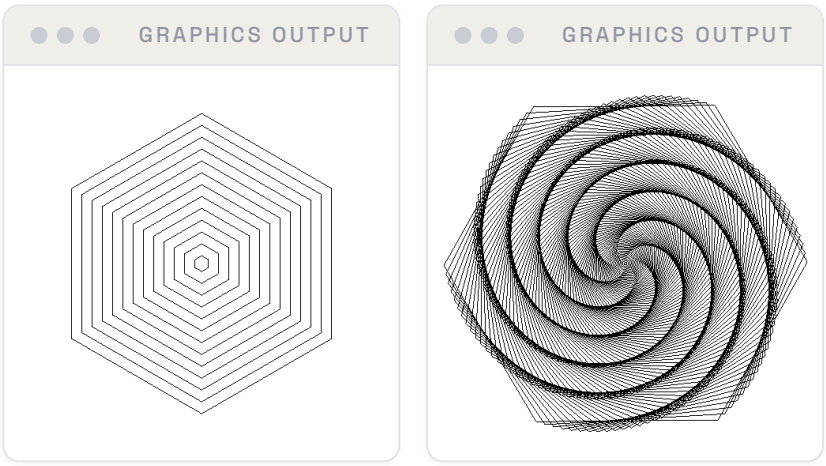
2 Use the hexagon below to solve problems 2 to 4. Its sides are 1 unit long and its point (0, 0) is in the middle. Draw a hexagon in the middle of the graphics output with sides 100 pixels long, using one **poly** statement and a **color** statement with a **clear** brush, so that only the outline shows.



3 Rewrite problem 2 with a **stamp** statement. Use its scale, so that you can draw a hexagon of any size by changing just one number.

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

4 Put the **stamp** statement from problem 3 inside a **for** loop and draw hexagons inside each other by changing the scale. Then try turning the hexagon as it grows — make the angle depend on the scale.



5 Write your name ten times down the graphics output with **text**, each time in a bigger font and in the next colour of the rainbow. Hint: in a **for** loop, use the loop variable for the *y* position, the font size and the hue of **hsv**.

6 Draw a clock face: a circle, and twelve hour marks made with one **stamp** of a thin rectangle inside a **for** loop. The marks are 30° apart.

Chapter 9 · Custom Graphics — Creating Your Own Shapes. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3. Programs are numbered 54–63, continuing from Chapter 8; Program 59 (a colour wheel with `hsv` and `radians`) is new, and the transparent triangles program became a TRY IT. Corrected for 2.3: the graphics output starts at 500 × 500, not 300 × 300; font weights go from 1 to 1000 (400 normal, 700 bold); `font` can make italic text; `graphsize` has an optional scale; `hsv` takes alpha as a percentage while `rgb` uses 0 to 255; `radians` and `degrees` convert angles. Also fixed: the triangle was called both equilateral and isosceles (it is equilateral, height 8.66), "fill the screen" now fills it, and the flower no longer draws its first petal twice. Programs are redrawn for the 500 × 500 graphics output; array forms of `poly` and `stamp` wait for the arrays chapter. Every picture was produced by running the program in BASIC-256 2.3. 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.