Chapter 9: Custom Graphics – Creating Your Own Shapes.

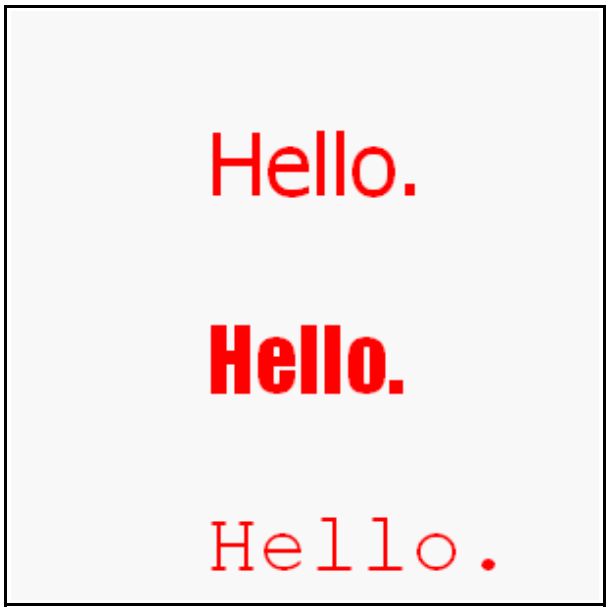
This chapter we will show you how to draw colorful words and special shapes on your graphics window. Several topics will be covered, including: fancy text; drawing polygons on the graphics output area; and stamps, where we can position, re-size, and rotate polygons. You also will be introduced to angles and how to measure them in radians.

Fancy Text for Graphics Output:

You have been introduced to the *print* statement (Chapter 1) and can output strings and numbers to the text output area. The *text* and *font* statements allow you to place numbers and text on the graphics output area in a variety of styles.

```
1     # graphichello.kbs
2     # drawing text
3
4     clg
5     color red
6     font "Tahoma",33,100
7     text 100,100,"Hello."
8     font "Impact",33,50
9     text 100,150,"Hello."
10    font "Courier New",33,50
11    text 100,250,"Hello."
```

Program 48: Hello on the Graphics Output Area



Sample Output 48: Hello on the Graphics Output Area

`text x, y, expression`

Draw the contents of the *expression* on the graphics output area with it's top left corner specified by *x* and *y*. Use the font, size, and weight specified in the last **font** statement.

New Concept

```
font font_name, size_in_point, weight
```

Set the font, size, and weight for the next *text* statement to use to render text on the graphics output area.

Argument	Description
font_name	String containing the system font name to use. A font must be previously loaded in the system before it may be used. Common font names are displayed below.
size_in_point	Height of text to be rendered in a measurement known as point. There are 72 points in an inch.
weight	Number from 1 to 100 representing how dark letter should be. Use 25 for light, 50 for normal, and 75 for bold.

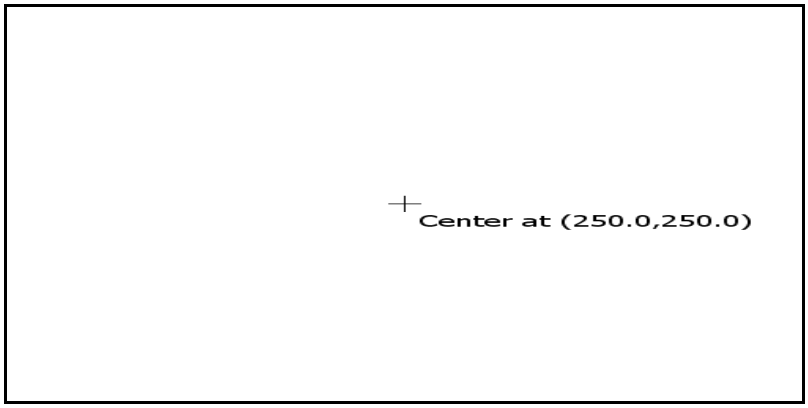
Illustration 17: Common Windows Fonts

Resizing the Graphics Output Area:

By default the graphics output area is 300x300 pixels. While this is sufficient for many programs, it may be too large or too small for others. The **graphsize** statement will re-size the graphics output area to what ever custom size you require. Your program may also use the **graphwidth** and **graphheight** functions to see what the current graphics size is set to.

```
1     # resizegraphics.kbs
2     # resize the graphics output area
3
4     graphsize 500,500
5     xcenter = graphwidth/2
6     ycenter = graphheight/2
7
8     color black
9     line xcenter, ycenter - 10, xcenter, ycenter + 10
10    line xcenter - 10, ycenter, xcenter + 10, ycenter
11
12    font "Tahoma",12,50
13    text xcenter + 10, ycenter + 10, "Center at (" +
      xcenter + "," + ycenter + ")"
```

Program 49: Re-size Graphics



Sample Output 49: Re-size Graphics


```
graphsize width, height
```

Set the graphics output area to the specified *height* and *width*.


```
graphwidth or graphwidth()
graphheight or graphheight()
```

Functions that return the current graphics height and width for you to use in your program.

Creating a Custom Polygon:

In previous chapters we learned how to draw rectangles and circles. Often we want to draw other shapes. The *poly* statement will allow us to draw a custom polygon anywhere on the screen.

Let's draw a big red arrow in the middle of the graphics output area. First, draw it on a piece of paper so we can visualize the coordinates of the vertices of the arrow shape.

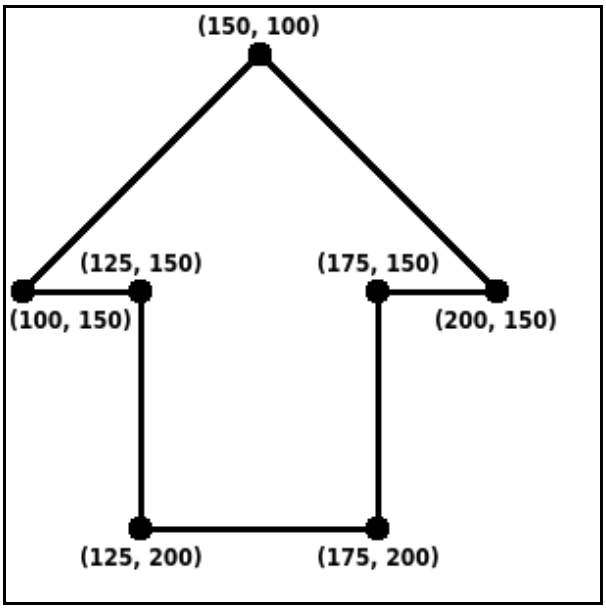
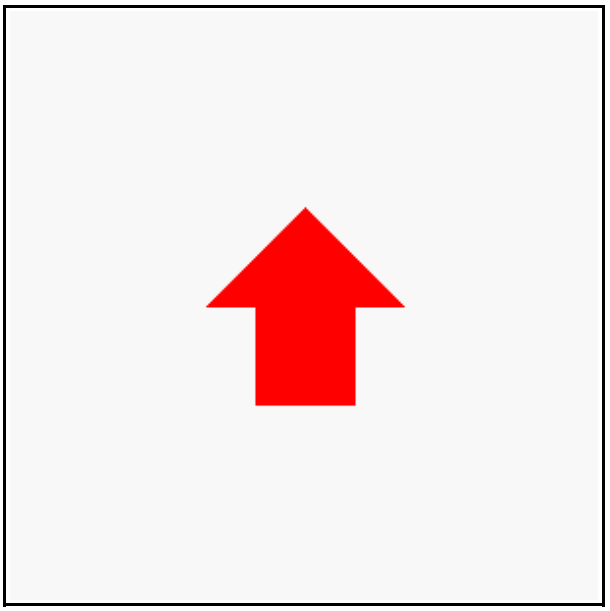


Illustration 18: Big Red Arrow

Now start at the top of the arrow going clockwise and write down the x and y values.

```
1     # bigredarrow.kbs
2     clg
3     color red
4     poly { 150, 100, 200, 150, 175, 150, 175, 200, 125,
      200, 125, 150, 100, 150 }
```

Program 50: Big Red Arrow



Sample Output 50: Big Red Arrow


```
poly {x1, y1, x2, y2 ... }
poly numeric_array[]
```

Draw a polygon using the points for the corners. The array is evaluated by taking two values at a time and using them for the x and y values to plot a vertex.

Stamping a Polygon:

The **poly** statement allowed us to place a polygon at a specific location on the screen but it would be difficult to move it around or adjust it. These problems are solved with the **stamp** statement. The **stamp** statement takes a location on the screen, optional scaling (re-sizing), optional rotation, and a polygon definition to allow us to place a polygon anywhere we want it in the

screen.

Let's draw an equilateral triangle (all sides are the same length) on a piece of paper. Put the point (0,0) at the top and make each leg 10 units long (see Illustration 19).

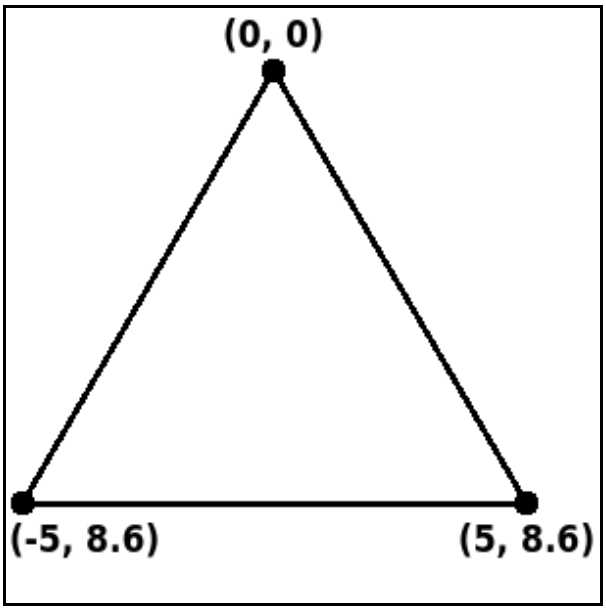


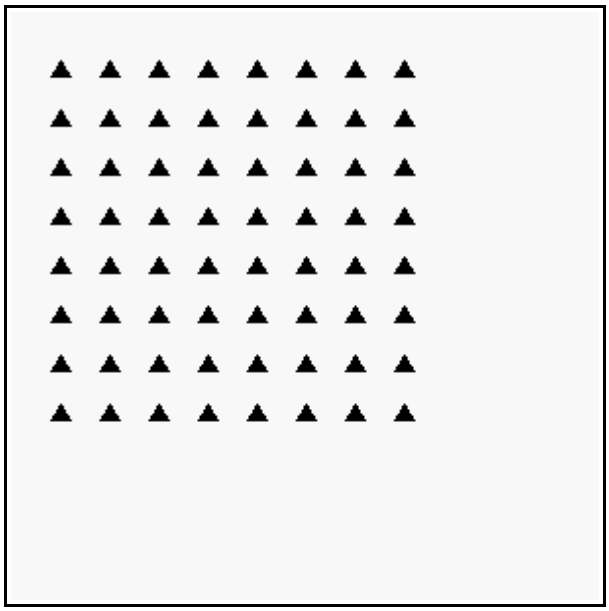
Illustration 19: Equilateral Triangle

Now we will create a program, using the simplest form of the **stamp** statement, to fill the screen with triangles. Program 51 Will do just that. It uses the triangle stamp inside two nested loops to fill the screen.

```
1     # stamptriangle.kbs - use a stamp to draw many
      triangles
2
3     clg
4     color black
```

```
5     for x = 25 to 200 step 25
6        for y = 25 to 200 step 25
7           stamp x, y, {0, 0, 5, 8.6, -5, 8.6}
8        next y
9     next x
```

Program 51: Fill Screen with Triangles



Sample Output 51: Fill Screen with Triangles


```
stamp x, y, {x1, y1, x2, y2 ...}}
stamp x, y, numeric_array[]
stamp x, y, scale, {x1, y1, x2, y2 ...}
stamp x, y, scale, numeric_array[]
stamp x, y, scale, rotate, {x1, y1, x2, y2 ...}
stamp x, y, scale, rotate, numeric_array[]
```

Draw a polygon with it's origin (0,0) at the screen position (x,y). Optionally scale (re-size) it by the decimal scale where 1 is full size. Also you may also rotate the stamp clockwise around it's origin by specifying how far to rotate as an angle expressed in radians (0 to 2π).

Radians 0 to 2π

Angles in BASIC-256 are expressed in a unit of measure known as a radian. Radians range from 0 to 2π. A right angle is π/2 radians and an about face is π radians. You can convert degrees to radians with the formula $r = d/180 * \pi$.

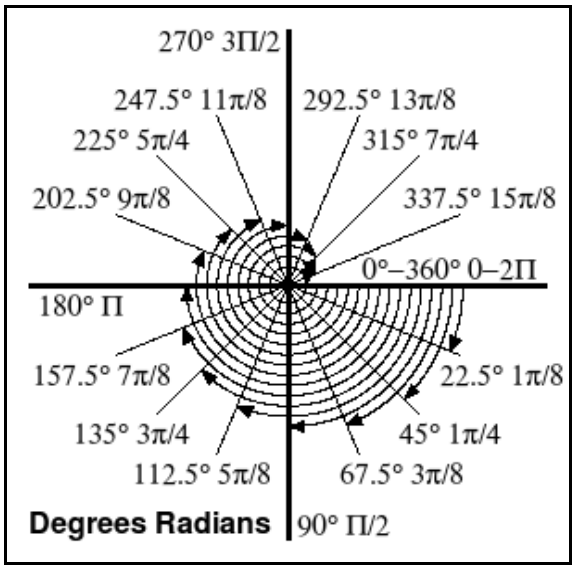
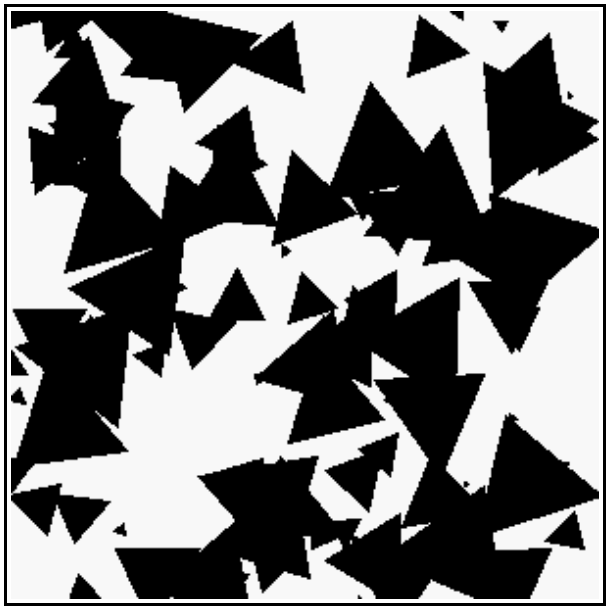


Illustration 20: Degrees and Radians

Let's look at another example of the stamp program. Program 52 used the same isosceles triangle as the last program but places 100 of them at random locations, randomly scaled, and randomly rotated on the screen.

```
1     # stamptriangle2.kbs - stamp randomly sized and
      rotated triangles
2
3     clg
4     color black
5     for t = 1 to 100
6       x = rand * graphwidth
7       y = rand * graphheight
8       s = rand * 7          # scale up to 7 times larger
9       r = rand * 2 * pi     # rotate up to 2pi (360
      degrees)
10      stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
11    next t
```

Program 52: One Hundred Random Triangles



Sample Output 52: One Hundred Random Triangles

pi

The constant *pi* can be used in expressions so that you do not have to remember the value of π. Π is approximately 3.1415.

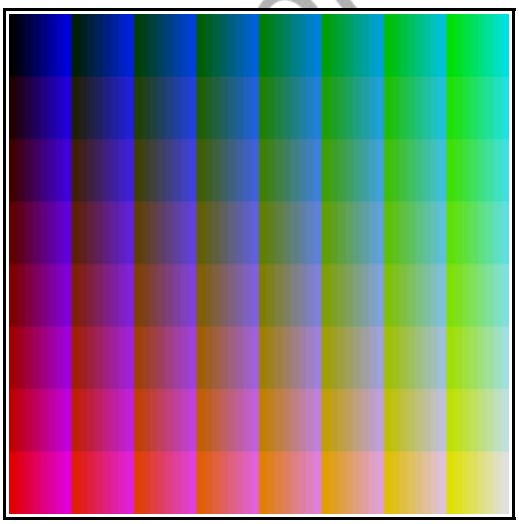
Sixteen Million Different Colors

BASIC-256 will allow you to define up to 16,777,216 unique colors when you draw. The RGB color model adds red (R), green (G), and blue (B) light together to form new colors. If all of the three colors are set to zero the color

Black will be created, if All three colors are set to the maximum value of 255 then the color will be white.

```
1     # 512colors.kbs - show a few of the 16 million colors
2     graphsize 256, 256
3     clg
4
5     for r = 0 to 255 step 32
6        for g = 0 to 255 step 32
7              for b = 0 to 255 step 32
8                 color rgb(r,g,b)
9                 rect b/8+g, r, 4, 32
10             next b
11          next g
12    next r
```

Program 53: 512 colors of the 16 million



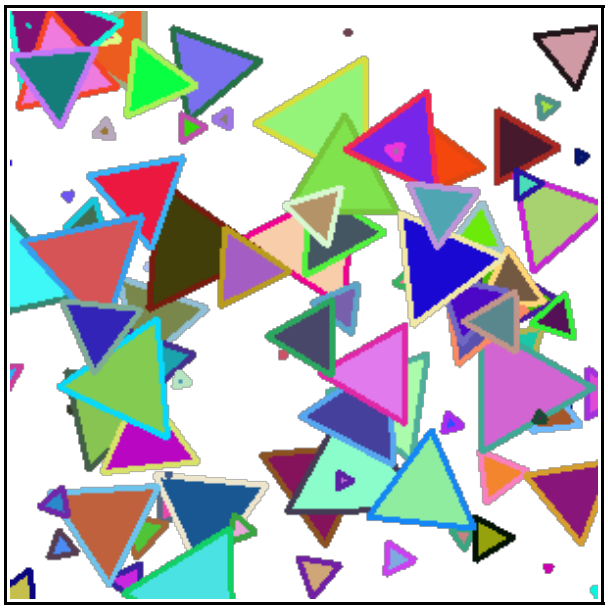
Sample Output 53: 512 colors of the 16 million

rgb(*red*, *green*, *blue*)
rgb(*red*, *green*, *blue*, *alpha*)

The **rgb** function returns a single number that represents a color expressed by the three or four values. The *red*, *blue*, and *green* values represent how much of those colors to include (255-on to 0-off). The optional alpha value represents how transparent the color is (255-solid to 0-totally transparent).

```
1     # stamptriangle3.kbs - stamp randomly colored, sized
      and rotated triangles
2
3     clg
4     penwidth 3
5
6     for t = 1 to 100
7        x = rand * graphwidth
8        y = rand * graphheight
9        s = rand * 7          # scale up to 7 times larger
10       r = rand * 2 * pi     # rotate up to 2pi (360
      degrees)
11       rpen = rand * 256     # get the RGBparts of a
      random pen color
12       gpen = rand * 256
13       bpen = rand * 256
14       rbrush = rand * 256   # random brush (fill) color
15       gbrush = rand * 256
16       bbrush = rand * 256
17       color rgb(rpen, gpen, bpen), rgb(rbrush, gbrush,
      bbrush)
18       stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
19    next t
```

Program 54: 100 Random Triangles with Random Colors



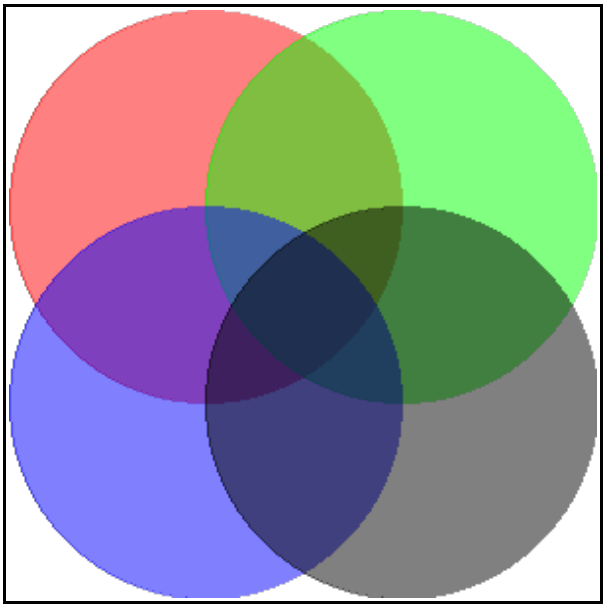
Sample Output 54: 100 Random Triangles with Random Colors

In addition to setting the exact color we want we can also define a color to be transparent. The RGB function has a fourth optional argument to set the alpha (transparency) property of a color. Zero is totally see through, and invisible, while 255 is totally opaque.

```
1     # transparent.kbs - show the nature of transparent
      colors
2     clg white
3
4     color rgb(255,0,0,127)
5     circle 100,100,100
6
7     color rgb(0,255,0,127)
8     circle 200,100,100
9
10    color rgb(0,0,255,127)
11    circle 100,200,100
12
13    color rgb(0,0,0,127)
```

```
14    circle 200,200,100
```

Program 55: Transparent Circles

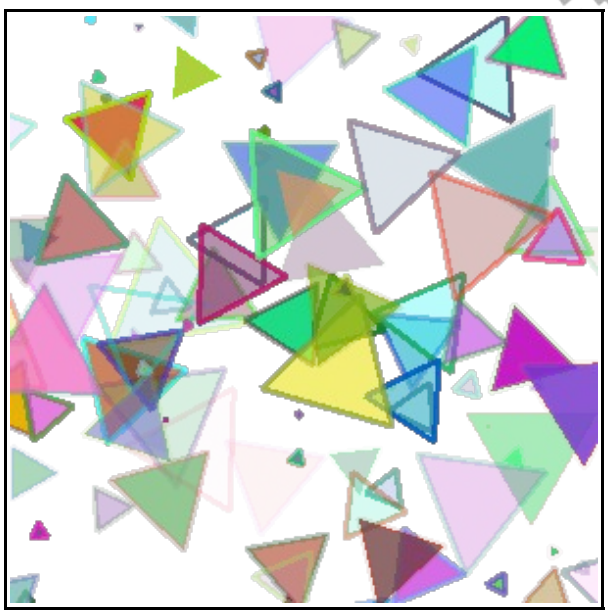


Sample Output 55: Transparent Circles

```
1     # stamptriangle4.kbs - stamp randomly colored, sized
      and rotated triangles
2
3     clg
4     penwidth 3
5
6     for t = 1 to 100
7        x = rand * graphwidth
8        y = rand * graphheight
9        s = rand * 7          # scale up to 7 times larger
10       r = rand * 2 * pi     # rotate up to 2pi (360
      degrees)
11       rpen = rand * 256     # get the RGBparts of a
      random pen color
12       gpen = rand * 256
13       bpen = rand * 256
```

```
14      apen = rand * 256
15      rbrush = rand * 256   # random brush (fill) color
16      gbrush = rand * 256
17      bbrush = rand * 256
18      abrush = rand * 256
19      color rgb(rpen, gpen, bpen, apen), rgb(rbrush,
      gbrush, bbrush, abrush)
20      stamp x, y, s, r, {0, 0, 5, 8.6, -5, 8.6}
21    next t
```

Program 56: 100 Random Triangles with Random Transparent Colors



Sample Output 56: 100 Random Triangles with Random Transparent Colors

Big Program	Let's send flowers to somebody special. The following program draws a flower using rotation and a stamp.

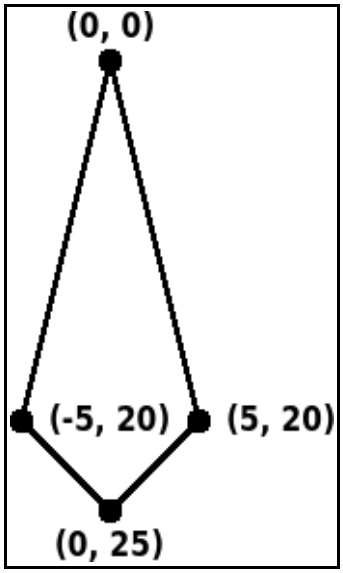
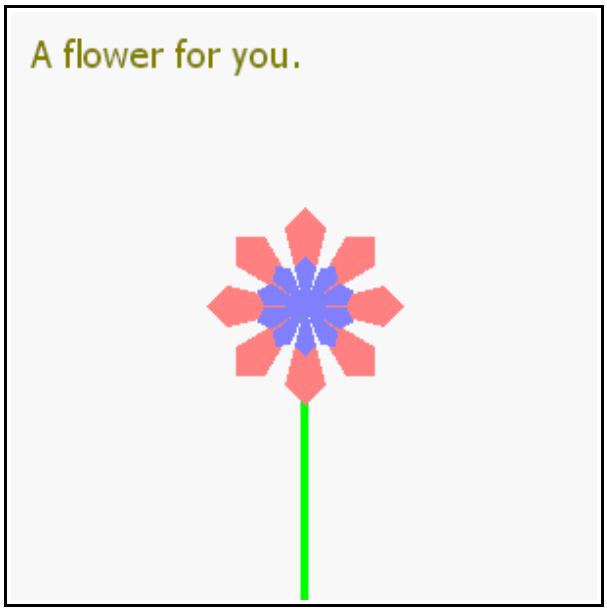


Illustration 21: Big Program - A Flower For You - Flower Petal Stamp

```
1     # aflowerforyou.kbs - use stamps to draw a flower
2
3     clg
4
5     color green
6     rect 148,150,4,150
7
8     color rgb(255,128,128)
9     for r = 0 to 2*pi step pi/4
```

```
10      stamp graphwidth/2, graphheight/2, 2, r, {0, 0, 5,
    20, 0, 25, -5, 20}
11  next r
12
13  color rgb(128,128,255)
14  for r = 0 to 2*pi step pi/5
15      stamp graphwidth/2, graphheight/2, 1, r, {0, 0, 5,
    20, 0, 25, -5, 20}
16  next r
17
18  message = "A flower for you."
19
20  color darkyellow
21  font "Tahoma", 14, 50
22  text 10, 10, message
23  say message
```

Program 57: Big Program - A Flower For You



Sample Output 57: Big Program - A Flower For You

Exercises:

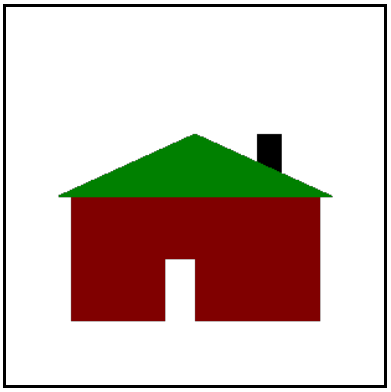
Word Search

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

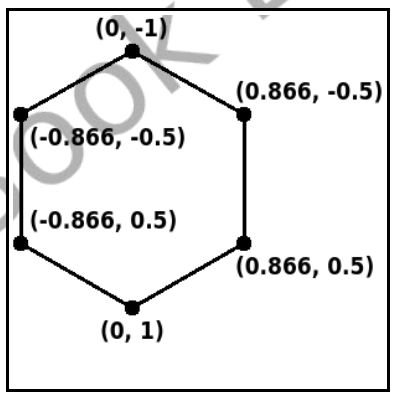
alpha, blue, degrees, font, graphheight, graphics, graphsize, graphwidth, green, pi, point, polygon, radian, red, rgb, stamp, text, transparent, weight

Problems

1. Use two **poly** and one **rect** statements to draw a simple house similar to the one shown below. Your house can be any combination of colors you wish it to be.



Use the hexagon below as a guide to help you to solve Problems 2 through 4. The sides of the hexagon are one unit long and the origin (0,0) is in the center of the shape.



2. Use a **color** statement with a clear brush and a single **poly** statement to draw a hexagon in the center of the graphics screen with each side 100 pixels long.

3. Rewrite #2 to use a **stamp** statement. Use the scale feature of stamp so that you may draw a hexagon of any size by only changing one number.

	4. Put the **stamp** statement from #3 inside a **for** loop and draw a series of nested hexagons by changing the scale. You may want to experiment with the step clause and with rotating the hexagon at the same time.