Advanced Graphic Functions

This section describes some of the more "advanced" functions which you can use to create some more sophisticated or complicated shapes and effects. These functions are not "simple". You don't need these functions to produce the basic simple graphs, however if you wish to create some more interesting effects, you can use some of the techniques described below.

It is assumed that you are already familiar with the topics presented in "Intermediate Graphics". If not, it is recommended that you **read that document first**.

QuickCalc graphics is designed to be easy to learn. Start with the basic stuff first. Then, if you are brave or have a lot of time to experiment, try some of the advanced stuff described in this section.

Advanced graphic functions include:

- Polylines
- Bezier Curves
- Smoothing and Averaging Plots and Polylines
- Connected (Complex) Shapes
- Outline Text Characters
- Selecting Shapes with the Mouse
- Deleting Shapes

Polylines

A "Polyline" is a series of connected points. It is similar to a polygon, except that it is *not* a closed shape, even if it returns to its start point. It is drawn as a series of line segments.

To draw a polyline, you provide a list of x,y points in an array, similar to the way you do when drawing a polygon, except that you must provide all points including the first and last. Then you issue the **SHAPE POLYLINE** statement, referencing the array and the number of points in the array.

The array must be dimensioned with **DIM** *polyarray* (*n* ,2), where *n* is at least as large as the number of points.

Example:

```
DIM  points (3,2)
points (0,0)=20: points (0,1)= 10
points (1,0)=20: points (1,1)= 0
points (2,0)= 0:  points (1,1)= 0
```

SHAPE POLYPOINTS=*points*, **POLYCOUNT**=3, **POLYLINE**

Points in a Polyline may be smoothed and/or averaged (see below).

A polyline resembles a plot of data points, but keep in mind the following differences:

- A Plot may be sorted (x, y, or none).
- Polylines are always drawn in the order the points appear in the array.
- Plots can happen in real time, as the points are calculated.
- For Polylines, the entire array must be calculated before the polyline is drawn.
- Polylines may be part of a Complex Connected Shape (see below) – plots may not.

A Polyline may also resemble a Polygon, however:

- A Polygon is a closed shape. It may be filled with a color or pattern.
- A Polyline is an open figure, and cannot be filled, even if it ends at its start point.
- A Polyline may be used as part of a Complex Connected Shape (see below).
- A Polygon does not specify its final point, which is assumed to be the same as its start point.
- A Polyline specifies all points, including both ends.
- When smoothing or averaging a Polyline, the end points are not changed (see below). For Polygons, all points are smoothed or averaged.

Bezier Curves

A [cubic] Bezier curve is a smooth curved line generated by a cubic equation. You can achieve curves of almost any shape using one or more Bezier curves.

> **Note**: You can find out all kinds of information about Bezier curves on the internet. I won't attempt to describe the math involved here.

A (cubic) Bezier curve is specified by giving the endpoints (**LINESTART** and **LINEEND**) plus two control points (**CTLPT1** and **CTLPT2**). Think of the control points as "attractors" which "pull" the curve in the direction of the control points. The farther the control points are away from the line, the more the line is distorted. (If the control points lie along the line, the result will be a straight line.)

You can experiment with different values of **CTLPT1** and **CTLPT2** to see which give you the best results. The curve will always approach the end point along a line between the end point and its associated control point. This helps you set the

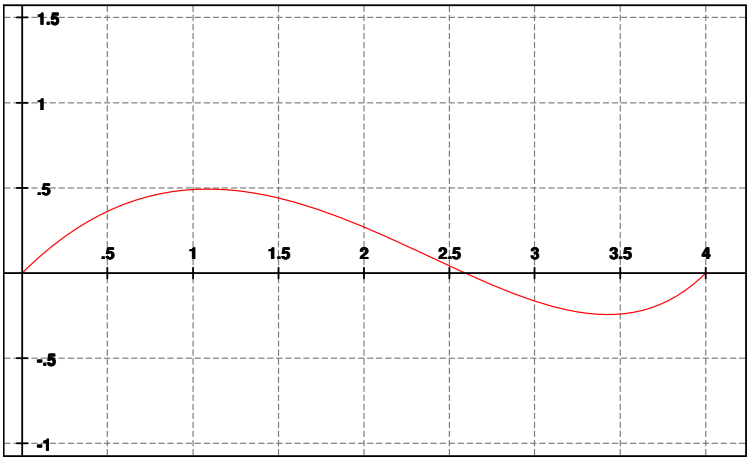
tangent to the curve as it reaches the end point. Again, refer to the internet and other literature for examples of how to set Bezier control points.

To plot a Bezier curve, specify the endpoints and control points, then issue the **SHAPE BEZIER** statement.

Example:

```
GRAPH HEIGHT=5, WIDTH=10, EQUALSCALES
SHAPE LINESTART = (0, 0)
SHAPE CTLPT1    = (1.5, 1.5)
SHAPE CTLPT2    = (3, -1)
SHAPE LINEEND   = (4, 0)
SHAPE LINECOLOR=(255, 0, 0), BEZIER
```

The graph looks like:



You can create more complex shapes by stringing Bezier curves end-to-end.

> **Note**: Automatic scaling includes the control points, which may result in some unwanted white space around the curves.

Bezier curves may be used, along with lines and polylines, as parts of Complex Connected Shapes (see below).

Smoothing and Averaging plots, polylines and polygons

A **PLOT** is basically a series of connected points. So is a **POLYLINE**. A **POLYGON** is simply a polyline which closes on itself, that is, it ends back at its starting point and defines a closed shape.

There are two ways that QuickCalc BASIC can smooth a plot:

Averaging

When you average a plot, you look at the previous and next points, compute their average, and then move the original point in that direction. This is done for all points except the start and end.

The effect of averaging is to lessen the extremes, or make the line less jagged. In this way, trends become more visible. **The data plotted is no longer exact**. This can be useful when looking at the performance of a stock over time without being concerned with the daily highs and lows.

Note: Averaging works best when the distance between the points on one axis is nearly uniform. If a bunch of points is clustered together, averaging will not have much effect around those points.

Averaging may be used in **PLOT**s, **POLYLINE**s, and **POLYGONS**.

To specify averaging for **PLOT**s, in the **PLOT** statement specify **AVERAGE**=*n*, where *n* may vary between 0 and 10. Zero (the default) means no averaging; the actual points are plotted. 10 is the maximum – each point is moved half the distance to the average of the previous and next points.

For **PLOT**s, the points may be sorted or unsorted.

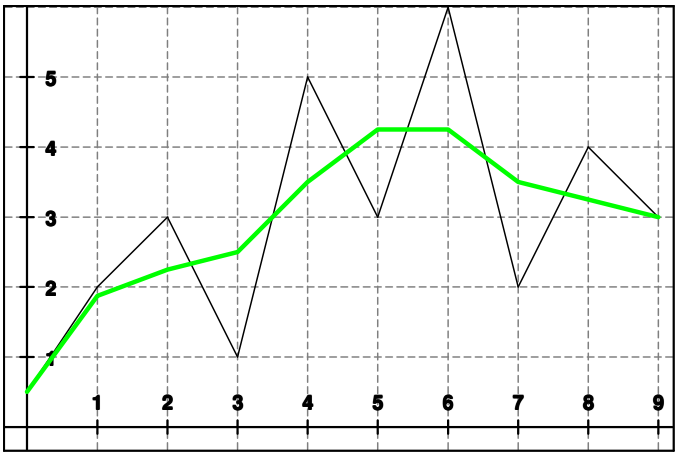
To specify averaging for **POLYLINE**s and **POLYGON**s, specify **AVERAGE**=*n* in the **SHAPE** statement just preceding the polygon or polyline.

For **POLYLINE**s, the polyline may be part of a path, or stand alone. Remember the start and end points will not be changed.

For **POLYGON**s, the start and end points are the same and ***will*** be averaged like all the other points.

Note: Be sure to reset **AVERAGE** to 0 after using it to avoid unintentionally propagating it to the next shape.

The following graph illustrates how a plot is affected by averaging (the green line is the same data plotted with **AVERAGE**=10):



Smoothing

Smoothing changes the jagged line into a smooth(er) curve, or more precisely, a series of Bezier curves.

The smoothed curve **passes through all of the points** in the plot (or polyline or polygon).

Smoothing may be combined with averaging. If the points have been changed using the **AVERAGE** parameter, the curve will pass through the new [averaged] points. If you do this for a plot, place the **SMOOTH** and **AVERAGE** parameters on the same **PLOT** statement, e.g.,

PLOT SMOOTH=8, **AVERAGE**=9, …

The tangent to the curve as it passes through each point is parallel to the line between the previous and next points.

Smoothing may be used in **PLOT**s, **POLYLINE**s, and **POLYGONS**.

To specify smoothing for **PLOT**s, in the **PLOT** statement specify **SMOOTH**=*n*.

The amount of smoothing is determined by the parameter *n*, where *n* may be from 0 to 10. Zero (the default) produces no smoothing. 10 produces the maximum smoothing (widest curves), while low numbers produce very tight curves at the points. Experiment to see which you like best.

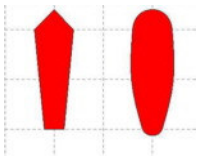
For **PLOT**s, the points may be sorted or unsorted.

To specify smoothing for **POLYLINE**s and **POLYGON**s, specify **SMOOTH**=*n* in the **SHAPE** statement just preceding the polygon or polyline.

For **POLYLINE**s, the polyline may be part of a path, or stand alone. Remember the start and end points will not be smoothed.

For **POLYGON**s, the start and end points are the same and ***will*** be smoothed like all the other points. Smoothing can have rather drastic effects on polygon shapes, like turning a rectangle into an ellipse, a star into a flower, a pentagon into a potato, etc.

Smoothing a polygon can sometimes simplify creating a difficult shape. The following diagram shows the result of plotting a simple polygon with **SMOOTH**=10 to create an unusual shape without having to resort to Bezier curves:

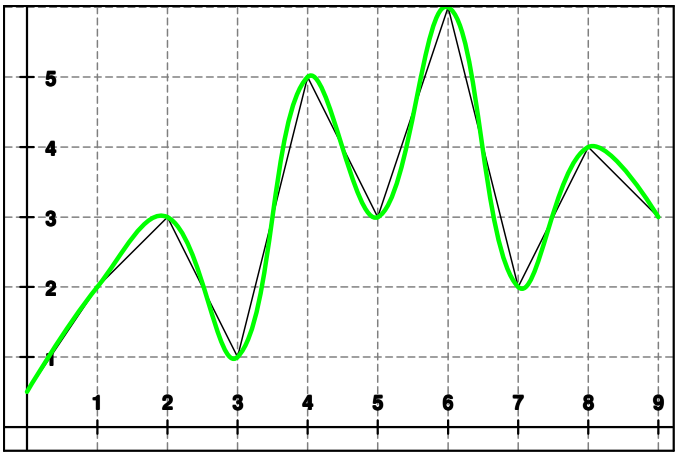


Note: If you are trying to plot a shape where some parts of it are smoothed and others have sharp corners, just using the **SMOOTH** parameter will not work. Instead, create a complex connected shape (see "Complex Connected Shapes", below) where the smooth part is a polyline which is smoothed and the other parts are figures which are not smoothed.

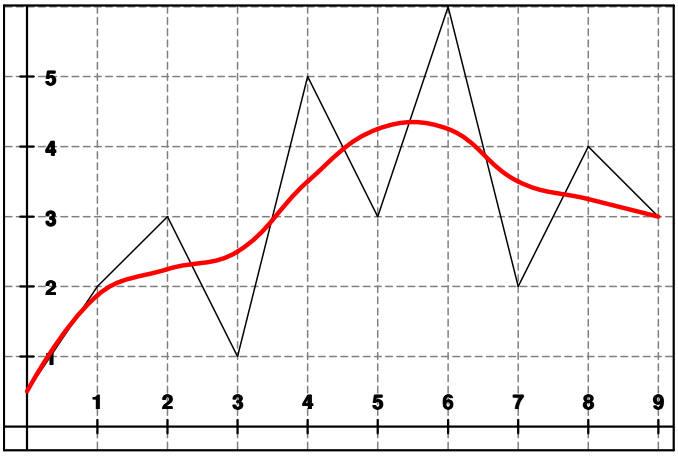
An example of this would be a plot where the area under the plot is to be filled.

Note: Be sure to reset **SMOOTH** to 0 after using it to avoid unintentionally propagating it to the next shape.

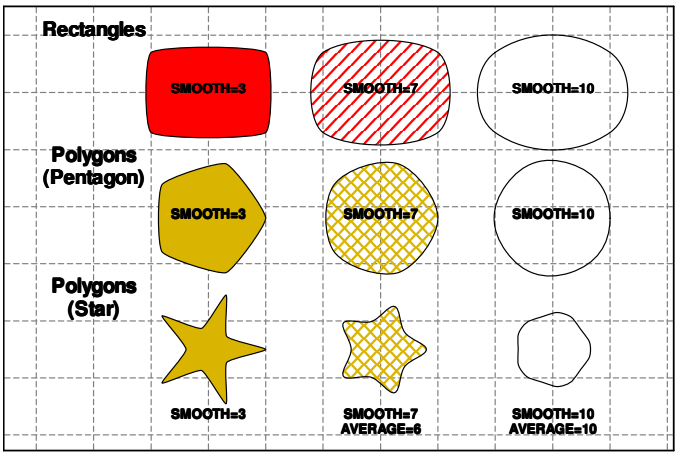
The following graph illustrates the effect of smoothing a plot (the green line is the same data plotted with **SMOOTH** = 8:



The following graph illustrates the effect of **smoothing and averaging** a plot (the red line is the same data plotted with **AVERAGE**=10 and **SMOOTH**=8):



The following graph shows examples of how polygons are affected by smoothing:



Smoothing and averaging may be used on Polylines which are part of a complex connected shape (see below). The Polyline will be smoothed and/or averaged just as if it were drawn as a separate figure.

Complex Connected Shapes

Sometimes, you may wish to draw a shape that is not a simple ellipse or polygon. You may wish to create a shape that is made up of curves and lines, or has holes cut out of it. Fonts are an example of complex shapes – the letter "B" has two holes in it, while the letter "i" has a separate dot over it. Using Complex Connected Shapes, you can draw virtually any shape you want.

Complex Connected Shapes may contain any other shapes. Polylines and polygons may be smoothed and/or averaged (see above), if desired. These figures are strung together end-to-end and finally closed (a line is drawn from the end point to the start point) to form a closed shape. If one figure does not start where the previous one left off, they will be connected. The final closed figure may then be outlined, filled, or both, just like any of the other solid shapes.

You start a Complex Connected Shape by specifying the **SHAPE BEGINPATH** statement. You then specify a sequence of **SHAPE** statements that outline the shape you desire. These shapes will not be drawn until the entire Complex Connected Shape is finished, at which time you specify the **SHAPE ENDPATH** statement.

Note: If you include a solid shape as part of the outline, remember that solid shapes are closed (return to their starting point). Normally, you would specify either a series of open shapes (line, Bezier, and polylines) or a single closed shape (like a rectangle, polygon or ellipse).

Note: Complex connected shapes may not contain other complex connected shapes. This includes outline text shapes (see below), since they are also complex connected shapes.

Now that you have defined the complete shape, you can cause it to be outlined with the **SHAPE STROKE** statement, or outlined and filled with the **SHAPE STROKEFILL** statement.

> **Note**: Do not attempt to draw other unrelated shapes while you are defining the complex connected shape.

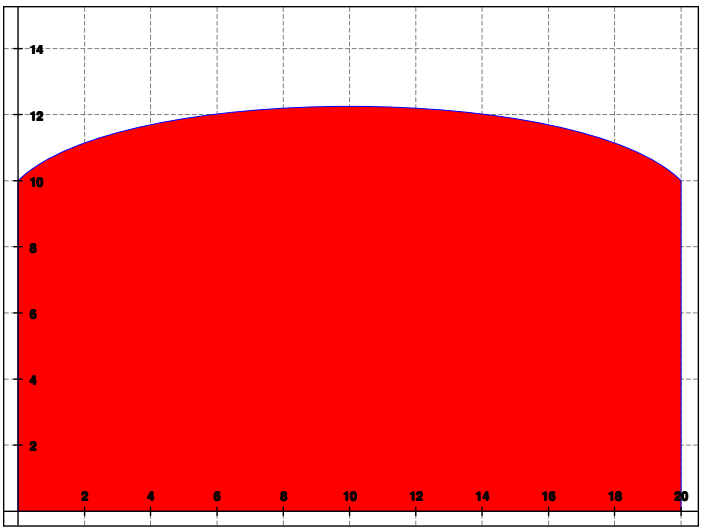
The line weight, line color, fill color and/or hatch pattern are specified with the same parameters used for other shapes (described above). The values in effect at the time of the **SHAPE STROKE** or **SHAPE STROKEFILL** statement will be used. You cannot have different values for different segments of the shape.

Note: If you do **STROKEFILL** and the current **FILLTYPE=NONE**, it will be treated as a **STROKE** (the figure will not be filled).

A **simple example** is as follows:

```
DIM points (3,2)
GRAPH  EQUALSCALES, LANDSCAPE, HSCALE=2.1
SHAPE BEGINPATH
SHAPE LINESTART=(0,0), LINEEND=(0,10), LINE
SHAPE LINESTART=(0,10), LINEEND=(20,10)
SHAPE CTLPT1=(3,13),CTLPT2=(17,13), BEZIER
points (0,0)=20: points (0,1)= 10
points (1,0)=20: points (1,1)= 0
points (2,0)= 0:  points (1,1)= 0
SHAPE POLYPOINTS=points, POLYCOUNT=3, POLYLINE
SHAPE ENDPATH
SHAPE LINECOLOR=(0,0,255), FILLCOLOR=(255,0,0),FILLTYPE=SOLID
SHAPE STROKEFILL
```

This will produce the following:



The "complex" shape has a line on the left side, a Bezier curve on the top, and a polyline completing the right and bottom sides. The interior is filled with red, although it could have been filled with any color or hatch pattern.

Multiple Closed Figures in a Complex Connected Shape

This is where it gets interesting. Suppose you want a shape which has a "hole" in it that doesn't get colored and shows whatever is behind it.

Many characters in fonts fit that pattern. Such a shape is created using (at least) two figures – one inside and one outside.

To create such a shape, draw the outer figure first, then draw the inner figure **in the opposite direction**. To separate the two figures so you don't get a line connecting them, use the statement **SHAPE CLOSEFIGURE** in between them, as follows:

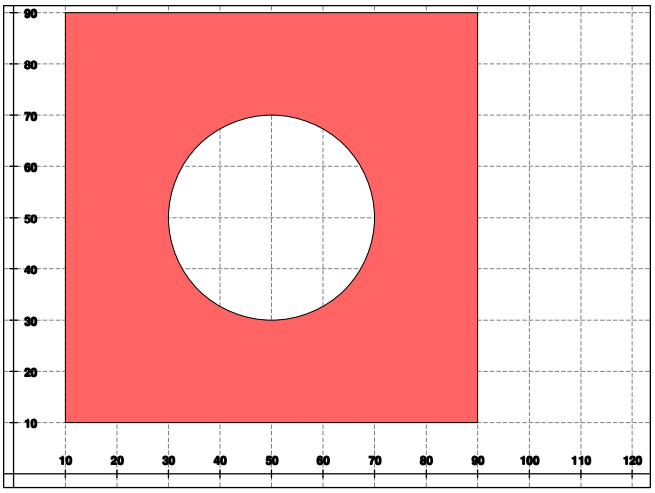
SHAPE BEGINPATH
… (**SHAPE** statements for the outer figure)
SHAPE CLOSEFIGURE
… (**SHAPE** statements for the inner figure)
SHAPE ENDPATH

Note: **BEGINPATH**, **ENDPATH** and **CLOSEFIGURE** should each be on separate **SHAPE** statements.

Note: To draw a closed shape, like an ellipse, in the "opposite" direction, use the **DIRECTION=**[NORMAL | REVERSE] parameter. Don't forget to reset the direction back after you draw the shape. An **ARC** is normally drawn counter-clockwise, but can be modified with the **DIRECTION** parameter.

The following is a simple example in which a circular hole is cut into a square figure:

```
// hole cut in square
GRAPH EQUALSCALES,LANDSCAPE
SHAPE BEGINPATH
SHAPE FILLCOLOR= (255, 100, 100)
SHAPE CENTER= (50, 50),RADIUS=40, RECTANGLE
SHAPE CLOSEFIGURE
SHAPE RADIUS=20, DIRECTION=REVERSE,  ELLIPSE
SHAPE ENDPATH
SHAPE DIRECTION=NORMAL
SHAPE STROKEFILL
```



To create a separate (disjointed) figure, such as the dot over an “i”, you can use the same syntax, however the second shape *does not have to* go in the opposite direction.

The following is an example of a more complex shape which has a several shapes “cut” out of the inside of it and a part outside the main figure:

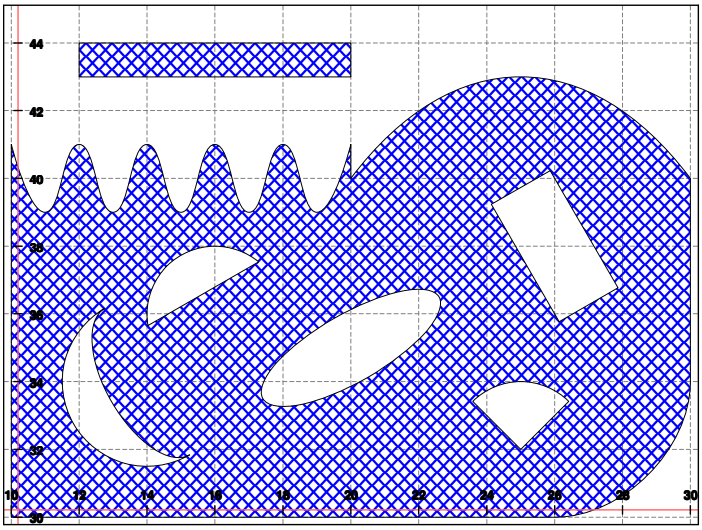
```
degrees
dim polyarray (12,2)
graph landscape,equalscales
graph includexaxis=0,includeyaxis=0

shape
fillcolor=(0,0,255),filltype=hatch,hatch=diagcross
gosub 1000

500

end

//-----------------------------------------------
1000  // subroutine to draw complex shape
shape beginpath

// Outer figure:
shape linestart=(10,30),lineend=(10,40),line
j=1
for i = 0 to 10:  // set up a zig-zag line
  polyarray (i,0) = 10+i
  polyarray (i,1) = 40+j
  if j = 1 then j = -1 else j = 1
next
shape smooth=10
shape polypoints=polyarray, polycount=11,polyline
shape smooth=0
shape linestart=(20,40),lineend=(30,40)
shape ctlpt1   =(23,44),ctlpt2 =(27,44),bezier
shape linestart=(30,40),lineend=(30,34),line
```

```
shape linestart=(30,34),lineend=(26,30)
shape ctlpt1   =(30,32),ctlpt2 =(28,30),bezier
shape linestart=(26,30),lineend=(10,30),line

shape closefigure

// Inner Figures:

// ellipse
shape rotate=30
shape center=(20,35),hradius=3,vradius=1,ellipse
shape rotate=0
shape closefigure

// rectangle
shape rotate=120
shape center=(26,38),hradius=2,vradius=1,rectangle
shape rotate=0
shape closefigure

// crescent (made from arcs)
shape rotatecenter="center",rotate=30
shape center=(14,34),radius=2.5
shape startangle=90,endangle=270,direction=normal
shape arc
shape hradius=1.2
shape startangle=270,endangle=90,direction=reverse
shape arc
shape rotate=0,direction=normal
shape closefigure

// pie segment
shape center=(25,32),radius=2
shape startangle=45,endangle=135,pie
shape closefigure

// chord
shape center=(16,36),radius=2
shape startangle=50,endangle=190,chord
shape closefigure

// Disjointed figure
shape left=12,right=20,top=44,bottom=43,rectangle

shape endpath
shape strokefill
return
```

In the above program, the entire complex connected shape is drawn in the subroutine at statement number 1000. The fill color is preset before calling the subroutine.

The inner figures are drawn with the “normal” direction because the outer figure runs clockwise. Some of the inner shapes are rotated.

The above program will produce the following graph:



Although there is no practical use for the above figure, it serves to illustrate how you can generate virtually any shape that you can imagine. Note how smoothing turned the zig-zag line into something resembling a sine curve.

Repeating a Complex Connected Shape

Once you have created a custom shape in this way, you may want to display it in different places on your graph. Unfortunately, once the shape is finished, it cannot be changed.

The best way do draw the same custom shape more than once is to place it in a user-defined function subroutine, where the starting point is a parameter of the subroutine and all points are offsets from that location. Other characteristics of the shape, such as its color and rotation, could also be parameters, or else set before the subroutine is called.

Rotating a Complex Connected Shape

A Complex connected shape **may be rotated** like any other shape, around any desired center of rotation. This rotation is ***in addition to*** the rotation of individual shapes contained within the complex shape. For example, if the complex shape contained an ellipse which was rotated to 45 degrees, and the entire shape is also rotated 45 degrees, the ellipse would end up rotated 90 degrees.

To rotate the complex shape, specify **ROTATECENTER** and **ROTATE** before the **SHAPE BEGINPATH**. The entire complex figure will be drawn rotated according to those parameters.

> If **ROTATE** is not specified for the complex shape, the rotation angle will be the value that was in effect when the **SHAPE BEGINPATH** was executed.
>
> If **ROTATECENTER** is not specified, or is set to "center", the complex shape will be rotated around the origin. It will not use the center point of the last shape.

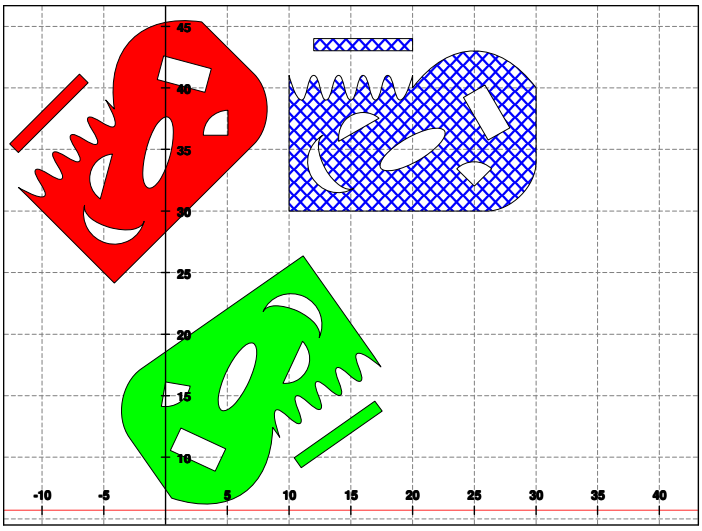
Individual objects inside the complex shape will have a default rotation of zero (relative to the complex shape), and no center of rotation specified, just like when the program started. You may specify **ROTATE** and **ROTATECENTER** for the individual shapes, just like normal shape drawing.

When the **SHAPE ENDPATH** is given, the rotation parameters are reset to the values in effect before the **BEGINPATH**.

In the program above, the complex shape was drawn in a subroutine at 1000. If we insert the following lines after statement 500:

```
shape rotate=45,rotatecenter=(10,10)
shape fillcolor=(255,0,0),filltype=solid
gosub 1000

shape rotate=215,rotatecenter=(10,28)
shape fillcolor=(0,255,0)
gosub 1000
```

the entire complex shape will be redrawn at different rotations and in different colors. Auto-scaling will re-size the graph to show all three figures, as follows:



Notice how the individual shapes rotated with the complex shape.

Outline Character Shapes

Some of the most interesting shapes are the characters in the scalable fonts (True Type, Adobe Type 1, etc.) that we use all the time on the computer. These characters are complex connected shapes made up of lines, polylines and Bezier curves. When they are printed or displayed on the screen, they are rendered into bitmap patterns.

QuickCalc has a function called **SHAPE TEXT** that allows you to draw those character shapes just like any other shape. They can be made any size, rotated, filled and/or outlined. All that you have to do is to specify where you want them drawn, how large, which font, which characters, and outline and fill information, just like any other shape. This information is provided in the usual **SHAPE** statements:

SHAPE LINEWEIGHT=*num-expr*,	thickness of line, in points
SHAPE LINECOLOR=(*r, g, b*),	color of outlining line
SHAPE FILLTYPE=[**SOLID** | **HATCH** | **NONE**]	
SHAPE HATCH=[**HORIZONTAL** | **VERTICAL** | **CROSS** | **RIGHT** | **LEFT** | **DIAGCROSS**]	
SHAPE FILLCOLOR=(*r, g, b*)	fill or hatch color

To specify the character(s) to draw, use the following new parameters:

SHAPE FONT=*string-expr*	Specifies the name of a font to be used for this operation. This must be an outline font (TrueType, Type 1, etc.) installed on your computer, and must match the font name exactly

(except for case). If there is no match, a generic sans-serif font will be used.

If you don't know the names of the installed fonts, go to the Control Panel and select "Fonts". Then select View / Hide Variations. Double-clicking on the font icon will show you what the font looks like. You can also use the function **CHOOSEFONT$** to select a valid font name.

SHAPE BOLD = *num-expr* — Specifies the weight (from 0 to 1000) of the font used in the **SHAPE TEXT** function. Default is 0, which means use a standard or "normal" weight for the font. 1000 is the boldest.

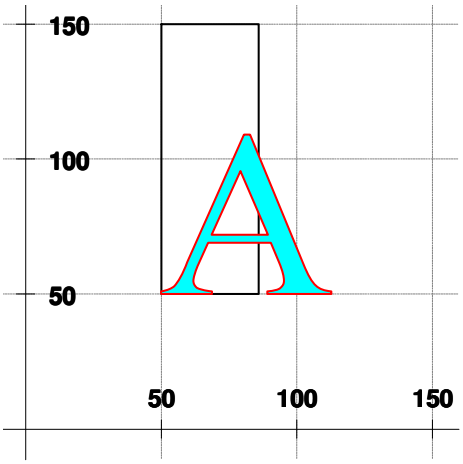
SHAPE ITALIC = *num-expr* — Specifies if you want normal or italic for the font used in the **SHAPE TEXT** function. Default is 0, which means normal. 1 means italic.

SHAPE CHARACTERS=*string-expr* — Specifies the character or characters you want to display (a maximum of 31). If any of these characters do not exist in the font you selected, blanks will be substituted. All characters in the string will be drawn with the same color, size, etc.

SHAPE JUSTIFY=[LEFT | CENTER | RIGHT]
Positions the text so that it starts, is centered about, or ends at the given starting point. Default is LEFT.

Note: This function is **not** intended for large amounts of text, or for small text. Small text does not look good when you try to outline it, and this function is ***much*** slower than the **TEXT** statement. It is more useful for larger decorative font effects and for logos.

To specify the location and size, use the parameters you would use to define a rectangle. Imagine a rectangle sitting where you want the first character to be drawn:



The bottom of the rectangle defines the baseline upon which the character(s) will sit. The left side defines the start point (usually the left edge) of the character(s). The height of the rectangle determines the size, including the descenders (so it appears taller than the character), and the width determines the average character width.

Use the parameters **SHAPE BOTTOM**, **SHAPE LEFT**, **SHAPE TOP**, and **SHAPE RIGHT** to specify this imaginary rectangle.

> **Note:** For center justification, the start point (bottom-left of imaginary rectangle) defines the center point of the character string. For right justification, it defines the right extent of the string. This is also the default center of rotation in all cases.

Aspect Ratio

The character you will be creating is a **shape**, which may appear stretched or compressed depending on the horizontal and vertical scale factors. If you want your font to appear "normal", consider the following:

- Shapes have their normal appearance (circles are circles, not ellipses) when the horizontal and vertical scale factors are **equal** (specify **GRAPH EQUALSCALES**).

  If you are using **EQUALSCALES**, you can set the **width of the imaginary box to zero**, i.e., set the right side to the same value as the left side. This will cause the characters to be drawn at the correct ("normal") aspect ratio.

- If your scale factors are not equal, set the box width to an aspect ratio that looks correct with the scale factors you are using. It should look about 3 times as high as it is wide, with your selected scale factors. It is best to

specify the scale factors explicitly in this case, since auto scaling may change the scale factors if other objects are added to the graph.

In either case, zooming the graph in the horizontal direction only (left- or right-arrows) changes the aspect ratio and will stretch or compress your drawn character shapes.

Rotation

You can rotate the characters to any angle. The string is rotated around the starting point (lower-left corner of the imaginary box). Use the statement

SHAPE ROTATE=*num-expr*

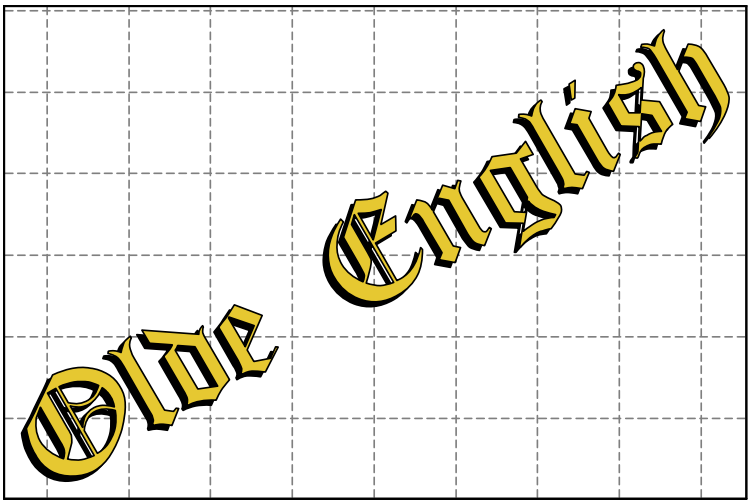
where the *num-expr* specifies an angle from 0 to 2π radians (or 0 to 360 degrees, if **DEGREES** is specified).

Note: Shapes which are rotated may appear slanted if the horizontal and vertical scale factors are different.

As with other shapes, you can also rotate the characters around an arbitrary point with the parameter **SHAPE ROTATECENTER=**(*x, y*).

Example:

```
DEGREES
GRAPH EQUALSCALES, INCLUDEXAXIS=0, INCLUDEYAXIS=0
GRAPH WIDTH=6, HEIGHT=4, NOAXES
SHAPE BOTTOM=100, LEFT=100, TOP=200, RIGHT=100
SHAPE LINECOLOR=(0, 0, 0), FILLTYPE=SOLID, ROTATE=30
SHAPE FILLCOLOR=(0, 0, 0)  // black
SHAPE FONT="Old English Text MT"
SHAPE BOLD=100, ITALIC=0
x$="Olde English"
SHAPE CHARACTERS=x$, TEXT
SHAPE LEFT=103, BOTTOM=103, TOP=203, RIGHT=103
SHAPE FILLCOLOR=(230, 200, 50)  // gold
SHAPE TEXT
```



Applications

You can use outline character shapes to create extremely large characters, and zoom way into them. This can be used to put a custom logo on your chart or create banners and signs. Interesting effects can be created by varying the fill colors and patters, line weights, aspect ratio, etc. Remember, white (255,255,255) is a valid fill color, which is different from transparent or no fill. Drawing the same text over itself but offset slightly can create interesting shadow effects (see above). Have fun! Get creative! Explore the fonts on your computer close-up. Print your girl/boy-friend's name in big colorful letters.

Logarithmic Scales

When plotting character outlines on logarithmic scales, the shapes will appear distorted. This is to be expected. It can produce some interesting effects. However, if you want the text to appear undistorted, consider using the **TEXT** statement instead.

Differences between SHAPE TEXT and TEXT.

Although these two statements can both produce text on your graph, their functions and intended uses are quite different.

Feature / Function	TEXT	SHAPE TEXT
Intended Use	Labeling points or objects, titles, etc.	Letters as **shapes**. Logos, signs, designs
Maximum characters	255	32
Multiple Lines	Yes	No
Maintains Shape	Yes, regardless of scale factors and log scales.	No. May become distorted. May create interesting effects.

Outline / Fill / Transparent / Hatch	No	Yes
Justify	Default is CENTER	Default is LEFT
Data Size	Yes	Determined by SHAPE parameters
Point Size	Yes	No
Location on Physical Page	Yes, Use H= or V=	No
Use to Label Axes	Yes	No
Appearance	Looks better for small fonts. Anti-aliasing.	Poor appearance for very small fonts.
Boxed	Yes	No
Sliding Text	Yes	No
Speed	Faster for small fonts	Faster for large fonts
Auto-Scaling	Doesn't work	Works like any other shape.
Rotation	Rotates around start point (ANGLE=)	Rotates around start point (ROTATE=) or external point (ROTATECENTER=)
Superscripts and Subscripts	Yes	No
Any color	Yes	Yes, outline color and fill color may be different.
Can be selected with the mouse or deleted	No	Yes

Getting an Outline of a text character.

This is similar to the **SHAPE TEXT** function, except that instead of drawing the character, it **returns the points** required to draw that character **in an array**. In the BASIC program, you may then use the points any way you want (like laying out crop circles…).

The points returned are in data units. They are scaled, rotated and translated to the desired location as if you were drawing them, except that nothing is drawn.

Even though nothing is drawn, you must still have a **GRAPH** active.

You specify the same **SHAPE** parameters as for the **SHAPE TEXT** function:

BOTTOM
TOP
LEFT
RIGHT (usually set to the same value as LEFT)
FONT

CHARACTERS="*a-single-character*"
ROTATE
BOLD
ITALIC

remember: values for these parameters set in previous **SHAPE** statements will still be in effect, unless you override them.

In addition, specify an array in which the text shape information will be returned:

POLYPOINTS=*array*

This array must have been **previously dimensioned** using **DIM** *array* (*n*,2). *n* may be any value you like, since the array will be replaced with a new array which is the correct size to contain the text shape information.

Note: If you are going to draw the character outline which is returned in the array, once the array has been used to draw the character, it may be re-used to generate another character pattern, if you wish.

You **don't need** the following descriptive parameters, since this function does not draw anything on the screen:

LINEWEIGHT
LINECOLOR
FILLTYPE
FILLCOLOR
HATCH

Finally, specify the new parameter:

SHAPE CHAROUTLINE

This will select the specified font and get the outlines for the specified character. The outlines are returned in the array specified in **POLYPOINTS**. They will be **scaled, offset and rotated** just like for the **SHAPE TEXT** function. All values in the array are DOUBLE. If the array entry specifies a point, then *array*(*n*,0) is the *x*-value and *array*(*n*,1) is the *y*-value.

Note: If the first entry in the array is zero, then there is no outline because the character is either a space or an invalid character for that font.

Format for the Array.

The shape consists of one or more "**contours**". Each contour is a closed shape consisting of one or more "**curves**". Curves may be polylines or poly-Bezier curves. In each curve, the start point (which is not given) is

assumed to be the end point of the previous curve. If the last curve does not end at the contour start point, it is assumed that it will be closed up when the contour is finished. Poly-Bezier curves consist of 3 points (ctlpt1, ctlpt2, and end) for each segment, where the end point is also the start point for the next segment.

```
first "point":          (# contours,    index-to-metrics)
for each contour:
        1st "point"     (# curves,      0)
        2nd point       (start point x, y)
        for each curve:
                1st "point"     (# points,   type)
                                        type: 1=polyline, 3=Bezier
                ...             (point-x,       point-y)
                ...             etc.
         (next curve structure)
        ...
 (next contour)
...
```

"metrics" data:

first "point"	*x*- and *y*-offset to the point **where the next character should be drawn**, if another character is to follow this one. These coordinates are rotated along with all the points in the structure above. Although this function only generates one character outline at a time, you can call it multiple times to form a "word", in which case, the offset here will tell you where to start the next character, assuming you want the characters to lie along the same line.
bounding-box	The bounding box consists of 4 points which define a rectangular box surrounding the [rotated] character. The bounding box itself is not rotated, but encloses all the points of the [rotated] character shape. This can be used to determine whether a rotated character will fit within another shape or overlap another character or shape. **Note**: The bounding box of a rotated character may be slightly larger, since it must contain the rotated

boundary box of the un-rotated character.

Plotting the returned shape

Although it is much faster and easier to simply use the **SHAPE TEXT** function to draw character outlines on the screen, you may want to do one or more of the following:

- pick the outline apart,
- modify the outline,
- reflect the character ("mirror writing"),
- distort the shape,
- fill the shape with a pattern,
- rotate it about some external point,
- graph only part of the shape,
- write the points to a file,
- print out points to study them,
- use the data to lay out a large sign,
- draw the character rotated to a different angle,
- combine it with another shape before drawing it,
- write the array to a file, then read it back later in another program.

Note: If you are only generating the character outline in order to view or analyze its structure, it is best to generate it with its **start point at the origin** and **un-rotated**.

You can plot the character outline by using the statement **SHAPE PLOTCHAROUTLINE**. This statement interprets the array and creates a "Complex Connected Shape" (see above) and draws it using the current parameters:

LINEWEIGHT
LINECOLOR
FILLTYPE
FILLCOLOR
HATCH
ROTATE
ROTATECENTER

Simply reference the array with the **POLYPOINTS** parameter, as follows:

SHAPE POLYPOINTS=*array-name*, **PLOTCHAROUTLINE**

Note: You can also plot the character outline by writing a BASIC routine to go through the table and plot all the lines and curves. An example is provided in the web site sample program "character outlines.txt". This is

not simple, and the **PLOTCHAROUTLINE** function will handle it more easily, but the example is there to show you how to parse through the outline array and plot the outline.

Once the shape is plotted, the array of points still remains, and you can use it to plot the same character again with a different rotation, color, *etc*., or re-use the array for a different outline. You could also offset every point in the structure as you plot it to plot the character in a different location, or scale the points to a different size.

Note: It is faster (and easier) to re-generate the character outline with a different offset and/or rotation than to go through the structure and rotate and/or translate every point.

Note: If a **ROTATE** angle is in effect when you go to plot the character curves, the entire character outline will be rotated. If you specified a **ROTATE** angle when the outline was created, it will be rotated again when it is plotted (double rotation). The first rotation will be around the (**BOTTOM**, **LEFT**) point of the character's defining rectangle. The second rotation will be around the **ROTATECENTER** value in effect when you plot it.

Note: Although **SHAPE PLOTCHAROUTLINE** generates a Complex Connected Shape when plotted, it only generates **one entry** in the shape table. Therefore, the *lastshapeid* variable is correct after issuing the **SHAPE PLOTCHAROUTLINE** statement.

Plotting more than one character

If you want to plot several characters in a row, you need to know where to start the next character. This is given in the "metrics" part of the structure. The first "point" in the metrics is the *x* and *y* coordinates of where the next character should start. These coordinates are rotated and translated along with the character data, so you can use them as the start location for the next character.

Advantages of doing this instead of just using the **GRAPH TEXT** function:

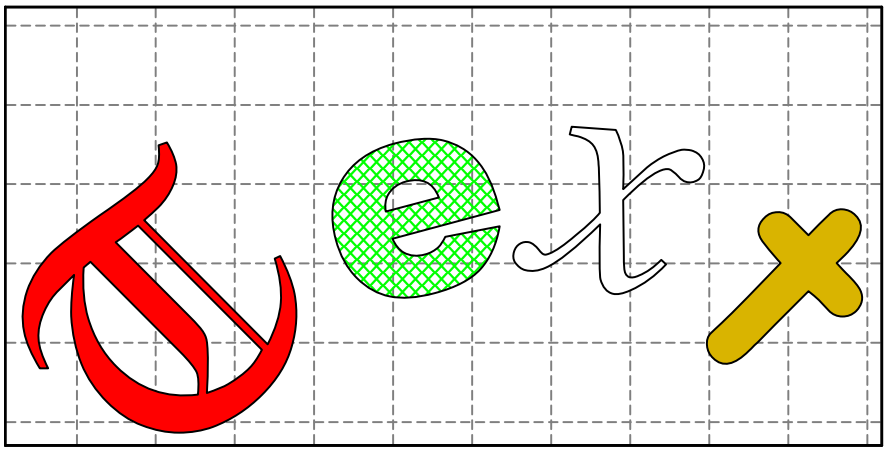
- Each character can be a different color or fill pattern, line weight, outline color.
- Each character can be a different font, bold, italic, etc.
- Each character can have a different rotation angle.
- All characters do not have to lie on a straight line.
- You can space the characters out or tighten them up by adjusting the next offset values (don't forget to rotate your "adjustments" so the characters stay on the line).

Some interesting designs can be created this way. It takes more work, but it enables you to create exactly the effect you want.

Example Program.

The following is an example of a BASIC program which will retrieve character outlines and then draw them. It illustrates some of the effects you can produce using character outlines.

First, the graph:



The BASIC program which created it is available on the website as "character outlines.txt".

Modifying the Character Outlines.

The biggest advantage in using **CHAROUTLINE** / **PLOTCHAROUTLINE** over just using **SHAPE TEXT** is the ability to modify the returned outlines before plotting them. In this way, you can create custom designs that aren't available using the fonts themselves. (Hint: try it with the "Wingdings" font.)

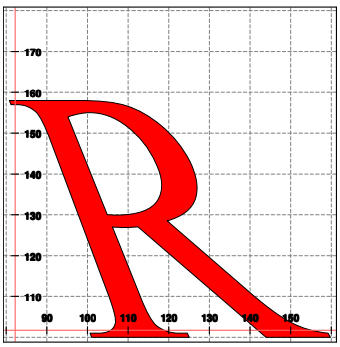
Once the outline has been returned, you may parse through it and apply transformations, rotations, scaling, or whatever you want, then plot the results.

CAUTION:

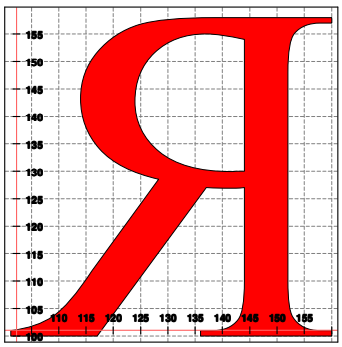
> The **structure must remain correct**. If you add a point, you must adjust all the offsets and counters to reflect the additional point(s). If you break the structure, **you could crash QuickCalc**, not just the BASIC program.
>
> If you add points, you will need to increase the size of the array, which means dimensioning a new array and copying all the points (plus the new ones) into it.

It is assumed that you know what you are doing.

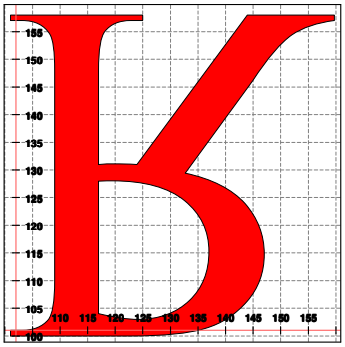
An **example** of how the structure can be modified is given below. This program allows you to modify a character outline by slanting it right or left, reflecting it horizontally, or flipping it vertically (or any combination). You can see where in the program the transformations are applied to the points. The following are sample graphs with different values of the parameters *slant*, *reflect*, and *flip*:



slant = -.2



reflect = -1
(like in "Toys R Us")



flip = -1

The example program which produced these graphs is shown below, and is also in the Sample Files as "warp character outlines.txt".

```
// This program demonstrates how to get character outlines,
//  modify them and plot them.

// In this example, the character outline is obtained, then
//   modified by slanting it forward or backward,
//   reflecting it and/or flipping it.

INCLUDE "standard_colors.txt"
// (make sure you have downloaded this file)
DEGREES
DIM array (10,2)

// Try different values of slant, reflect and flip
slant = -.2       // 0 is upright, + = slant right, - = left
reflect = 1       // 1 = normal, -1 = reflect horiz.
flip    = 1       // 1 = normal, -1 = flip vert.

GRAPH EQUALSCALES
GRAPH INCLUDEXAXIS=0, INCLUDEYAXIS=0
GRAPH WIDTH=6, HEIGHT=6
SHAPE FONT = "Times New Roman"
SHAPE CHARACTERS = "R"
SHAPE LEFT = 100,   RIGHT = 100
SHAPE BOTTOM = 100, TOP   = 200
SHAPE BOLD = 100, ITALIC = 0
SHAPE ROTATE = 0
```

```
SHAPE POLYPOINTS = array
SHAPE CHAROUTLINE
// Now array has new dimensions, and contains the outline.

// "Modify" the points in the array

i = 0
num_contours   = array(i, 0)              // number of contours
m              = array(i, 1)              // metrics offset
h   = array (m+3, 1) - array (m+2, 1)     // height
cx  = (array(m+2, 0) + array (m+1, 0))/2  // horiz center
b   = array(m+1, 1)                       // base
cy  = b + h/2                             // vert center

i = i + 1
WHILE num_contours > 0
  // Since we are only modifying the points, we can treat the
  // Polyline and Bezier curves the same.
  num_curves = array(i,0) // number of curves in this contour.
  i = i + 1
  // next point is start point for this contour.  Modify it.
  x = array(i,0):  y = array(i,1)  // get the point
  x = x * (1 + slant * (y-b) / h)  // slant   the point
  x = (x - cx) * reflect + cx      // reflect the point
  y = (y - cy) * flip    + cy      // flip    the point
  array(i,0) = x: array (i,1) = y  // put it back
  i = i + 1

  WHILE num_curves > 0
    num_points = array(i,0)   // number of points in curve
    i = i + 1

    WHILE num_points > 0
      x = array(i,0):  y = array(i,1)        // get the point
      x = x * (1 + slant * (y-b) / h)        // slant   the point
      x = (x - cx) * reflect + cx            // reflect the point
      y = (y - cy) * flip    + cy            // flip    the point
      array(i,0) = x: array (i,1) = y        // put it back
      i = i + 1
      num_points = num_points - 1
    WEND          // end of curve (polyline or Bezier)
    num_curves = num_curves - 1
  WEND            // end of contour
  num_contours = num_contours - 1
WEND              // end of shape

SHAPE ROTATE = 0
SHAPE FILLTYPE = SOLID, FILLCOLOR = red
SHAPE LINECOLOR = black

SHAPE PLOTCHAROUTLINE
END
```

Selecting Shapes With a Mouse

The purpose of QuickCalc graphics is *not* to create graphic-oriented programs like one does in C or C++. The idea was to provide easy-to-program, good-looking graphs that could be printed or included in other documents like reports.

That being said, it would be nice, sometimes, to click on a shape that you have drawn and have something happen.

> **Note**: This is not simple BASIC programming, but then, this *is* the Advanced Graphics document….

An attempt was made to keep this somewhat simple (like BASIC) but provide the functionality of being able to click on "objects" on the graph. This can provide you with the means to create a kind of menu or graphical selection tool.

Challenges

BASIC is not a multi-tasking environment, nor does it respond to interrupt-driven events. In order to respond to a mouse-click "event", you must be in a loop waiting for something to happen.

QuickCalc graphs are not static in size, shape, or scale. Using arrow keys, the mouse, or menu zoom functions, the graph can change its size and offsets. This means that the objects ("shapes") on the screen can change their size and position, and are sometimes partly off the screen. In order to be consistent with the design philosophy of QuickCalc graphics, mouse-clicking on shapes must work regardless of the zoom factor or offsets.

Some shapes are very irregular (see "Complex Connected Shapes", above). The program has to be able to identify the boundaries of any shape you can draw at any size, and recognize the inside of "hollow" shapes.

The user can already click on a graph, in order to "drag" it. For that reason, the mouse "click" will be tested when the button is released.

Making a Shape "Clickable"

When you code the **SHAPE** statement for the desired figure, include the parameter **MOUSECLICK** before the keyword that actually draws the shape.

> **Example**: To draw a circle, at (10,10) with a radius of 5, code:
>
> **SHAPE CENTER**=(10,10), **RADIUS**=5, **MOUSECLICK**, **ELLIPSE**

The **MOUSECLICK** parameter should appear before the **ELLIPSE**, and is **good for only one shape**. It must be re-specified for each additional shape you draw that you want to be "clickable".

If you place two keywords (like **ELLIPSE** and **RECTANGLE**) on the same **SHAPE** statement, only the first one will get the **MOUSECLICK** attribute. It is best to use separate **SHAPE** statements in this case.

Identifying the Shape

When you code a **SHAPE** statement that draws a figure, the information is placed into the "shape table", which is used over and over again as the graph is re-drawn. If you are going to be able to identify which shape was clicked on, you must know the identifying number of the shape.

When a shape is added to the table, the identifying number is placed into a variable **LASTSHAPEID**, which you should access immediately after the **SHAPE** statement and before you create any more shapes (another reason for using separate **SHAPE** statements for each figure).

Example:

SHAPE CENTER=(10,10), **RADIUS**=5, **MOUSECLICK**, **ELLIPSE**
circle_id = **LASTSHAPEID**

Later, when the circle is clicked on, the ID returned will be the value in *circle_id*.

You may make any (or all *) of the shapes "clickable".

Notes:

* Shapes like **LINE** and **BEZIER** are "open" (have no inside). If the line is only 1 pixel wide, it is very difficult to click on.

Complex Connected Shapes (see above) have several entries in the shape table. The ID you want is the first one. Copy it immediately after the **BEGINPATH**.

The line outlining a shape is also considered part of the shape. This is noticeable if you have a "fat" line weight.

The "cut-out" part of a text symbol or complex connected shape is ***not*** considered part of the shape. If you click in it, you will not select that shape.

It doesn't matter whether the shape is filled with solid color, hatch, or no fill. The mouse selection works the same.

If you click on a spot where two or more shapes overlap, the ID returned is that of **the most recent shape that was drawn** (shape with the highest shape ID).

Too many "clickable" shapes could slow the response of the program to mouse clicks.

If you have a lot of mouse-selectable shapes, you will probably want to create a table [array] of the ID's and meaning of the shapes. That is a programming decision.

If you are going to be clicking on your graph, you can use the statement **GRAPH LOCKSCREEN=**1 to prevent accidentally re-positioning or re-scaling the graph.

Graphs which include "clickable" shapes do not look any different than ordinary graphs. Printing a graph or creating a metafile or JPEG will work the same as before.

If you are not testing for mouse clicks, you will not know that anything has been clicked on.

Testing for Mouse Clicks on Shapes

You can test if the mouse (left button) was clicked on a shape by using the function **CHECKMOUSECLICK**. This returns the Shape ID of the most recent "clickable" object that was clicked on.

If the mouse has not been clicked on the graph since the last call to **CHECKMOUSECLICK**, the function will return -1

If the mouse was clicked on the background or a "non-clickable" shape, the function will return -2. "Dragging" the graph (with the mouse not pointing at a "clickable" object) will produce a -2 when the button is released. You may want to ignore this event.

After calling **CHECKMOUSECLICK**, the most recent ID is reset (to -1), and subsequent calls to **CHECKMOUSECLICK** will return -1 until the mouse is clicked again.

If **CHECKMOUSECLICK** returns a number (positive or zero), it represents a Shape ID of a "clickable" shape. This signifies an event (mouse clicked on a shape). What you do with that information is up to you.

You can put **CHECKMOUSECLICK** anywhere in your program. For example, you might be performing a very long calculation and want to click on a "button" to display your progress.

Normally, you would come to the point in you program when you are looking for a mouse selection. At that point, you would loop, testing for mouse clicks, as follows:

```
100  DELAY 10
id = CHECKMOUSECLICK
IF id < 0 THEN GOTO 100
… (code to check the value of id and act accordingly)
GOTO 100       // continue checking
```

Note: There is no way to break out of that loop, unless one of the shapes represents a "End" button.

Note: The **DELAY** is to allow other programs to run while keeping CPU usage to a minimum while you are waiting. Making the delay too long will make dragging the graph very jerky, and is unnecessary. 10 works well.

Note: Test for *id*= -2 if you want to know if the user clicked on something other than a "clickable" shape. This is one way to exit, but remember that "dragging" can also result in a -2.

Note: If you want to create a "button" with text on it, draw the shape (usually a rectangle) and make it clickable. Then draw the text on top of it (with the **TEXT** statement) , using the **DATESIZE=** parameter so it will still fit if you zoom the graph.

Getting a Character from the Keyboard

CHECKMOUSECLICK will also test if the user typed a keystroke. This is similar to **INKEY**$, but you don't have to wait for it. The most recent occurrence of a mouse click or keystroke will be returned.

Keystrokes (characters) are indicated by a negative value of the ASCII character typed. For example, if you type "a", the value returned will be **–ASC**("a").

The **TAB** key and the arrow keys have special meaning in the QuickCalc graphics window, and are not acknowledged by **CHECKMOUSECLICK**.

Note: **CHECKMOUSECLICK** only works if the graph window is active and has the focus. If you switch focus to the QuickCalc BASIC window, typed characters will be ignored until you switch back.

Getting the Mouse Coordinates of Where You Clicked

Every time the [left] mouse button is released (while pointing at the graph), the mouse coordinates where you clicked (in data units) are stored in system variables **MOUSEXCOORD** and **MOUSEYCOORD**, where you can access them. This is true whether or not you got a "hit" on an object. However, you should still call **CHECKMOUSECLICK**, so that you will know if an event happened.

If a scale is logarithmic, that coordinate is converted from a logarithm back into data units. If a scale is DATETIME, the value returned is a DATETIME variable, which you can format, if desired, using **FORMATDATETIME$.**

Note: You may still check for mouse clicks, even if there are no "mouse-clickable" shapes. In that case, **CHECKMOUSECLICK** will return -2 if the mouse had been clicked and -1 if not. You can also retrieve the mouse coordinates, as above.

Getting the State of the SHIFT and CONTROL Keys.

Every time the mouse button is released, the state of the **SHIFT** and **CONTROL** keys is stored in the system variable **KEYSTATE**. The values are:

0 = Neither key is pressed
1 = **SHIFT** key is down
2 = **CONTROL** key is down
3 = Both keys are down

You can check the variable **KEYSTATE** to see if either key was held down at the time the [left] mouse button was released. In this way, you can cause a mouse click to have more than one meaning, *e.g.*, "create" or "delete".

Note: Dragging the graph will not happen if the **SHIFT** or **CONTROL** key is held down. That changes the mouse function and allows you to use the mouse for other things without unintentional dragging. You can also use **GRAPH LOCKSCREEN=1** to avoid unintentional dragging.

Sample Program

An example is given in the sample program **`mouse_click_shapes.txt`** (on the web site). Run it and click on the various shapes.

Deleting Shapes

Once a shape has been drawn, it becomes part of the graph and remains there until the graph is "closed". If you want to remove a shape from the graph, you can do so with the **DELETESHAPE** statement.

In order to delete a shape, you must know the Shape ID (see the preceding section). Again, you must save the value returned in **LASTSHAPEID**, immediately after creating the shape (or after the **BEGINPATH**, for complex shapes). This Shape ID is used to specify which shape you want to remove.

DELETESHAPE (*shape-id*)

There is no indication of success or failure with this statement. If there is no shape with that ID, no action is taken. If the shape has already been deleted, it is gone and can't be deleted again. It can be re-drawn, but then it will get a new Shape ID.

Note: Only shapes with a valid Shape ID can be deleted. You can't delete something that is a part of a path (complex connected shape). You must delete the entire complex connected shape.

Note: Shape IDs do not change when a shape is deleted, so the values you save for mouse-click testing are still valid (except for the one you deleted).

Note: The shape will be deleted even if it is not currently visible, scrolled off the graph, covered up by another shape, etc.

By using **DELETESHAPE**, you can create a kind of motion on your graph by repeatedly deleting and redrawing the shape, offset slightly (also, check out the **CHANGESHAPE** statement, described below.) Remember, that this is not a motion-graphics program, so I wouldn't expect too much, but you could have some fun with it.

Changing Shape Parameters

Once a shape has been drawn it is possible to change [some of] its parameters without having to delete and re-draw it. This is done using the **CHANGESHAPE** statement. The format is:

> **CHANGESHAPE** *shape-id*, *list-of-parameters*

In order to change a shape, you must know the **Shape ID** (see the preceding section). Again, you must save the value returned in **LASTSHAPEID**, immediately after creating the shape (or after the **BEGINPATH**, for complex shapes).

> If the given *shape-ID* does not correspond to an existing shape, it will cause an error.

The allowable parameters are *nearly* the same as for the **SHAPE** statement **descriptive** parameters. See the description for **SHAPE** statement parameters in the "*Intermediate Graphics*" document.

Not all parameters apply to all shapes, and some cannot be changed. Those parameters that can be changed are summarized in the table below, followed by notes on their use.

	Rectangle	Line	Bezier	Polygon	PolyLine	Ellipse	Arc	Chord	Pie	Text	PlotCharOutline	Complex Shape
Top	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
Bottom	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
Left	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
Right	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
Center	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	n/a	n/a
HRadius	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
VRadius	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
Radius	OK	n/a	n/a	n/a	n/a	OK	OK	OK	OK	OK	n/a	n/a
FillType	OK	n/a	n/a	OK	n/a	OK	n/a	OK	OK	OK	OK	OK
Hatch	OK	n/a	n/a	OK	n/a	OK	n/a	OK	OK	OK	OK	OK
FillColor	OK	n/a	n/a	OK	n/a	OK	n/a	OK	OK	OK	OK	OK
LineWeight	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
LineColor	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
Rotate	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	NO	NO
RotateCenter	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	NO	NO
LineStart	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
LineEnd	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
CtlPt1	n/a	n/a	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
CtlPt2	n/a	n/a	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
StartAngle	n/a	n/a	n/a	n/a	n/a	n/a	OK	OK	OK	n/a	n/a	n/a
EndAngle	n/a	n/a	n/a	n/a	n/a	n/a	OK	OK	OK	n/a	n/a	n/a
Bold	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	OK	n/a	n/a
Italic	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	OK	n/a	n/a
Font	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	OK	n/a	n/a
Characters	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	OK	n/a	n/a
Justify	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	OK	n/a	n/a
Smooth	OK	n/a	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Average	OK	n/a	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Direction	n/a	n/a	n/a	n/a	n/a	n/a	OK	OK	OK	n/a	n/a	n/a
MouseClick	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK
PolyPoints	n/a	n/a	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	NO	NO
PolyCount	n/a	n/a	n/a	OK	OK	n/a	n/a	n/a	n/a	n/a	NO	NO

Notes:

Refer to the documentation for the **SHAPE** statement for the meaning of the various parameters for different shapes.

The **CHANGESHAPE** statement ***cannot*** be used to change the **type** of a shape, e.g., a rectangle cannot be changed into an ellipse. A circle can be changed into an ellipse because they are both ellipses – only the radii are changed.

You cannot change the **elements** of a Complex Connected Shape. The parameters you can change (*e.g.,* color) refer to the object as a whole.

Parameters which do not apply to the referenced shape will have no effect (see the table, above).

After changing the size and/or location of an object, automatic scaling *may* no longer work properly, *i.e*., the graph may not automatically adjust to contain the changed object. In most cases, the region will expand to contain the modified object, but will not contract if the object is moved closer to the center. If automatic scaling is not in effect, the object may disappear off the graph.

Changing parameters for a previously-drawn shape does ***not*** change the shape-ID number, unlike when the shape is deleted and re-drawn.

CHANGESHAPE does not affect the order in which shapes are drawn. The shape on top of another remains on top.

Descriptive parameters do not carry over to the next shape when using **CHANGESHAPE**, unlike what happens with the **SHAPE** statement.

Each **CHANGESHAPE** statement operates on only one shape, referenced by the shape-ID. The statement may contain many parameters.

You can successively modify the same shape with different **CHANGESHAPE** statements. Just use the same shape-ID.

If several parameters are being changed for the same shape, it is best to put them all on the same **CHANGESHAPE** statement, if possible. The shape is not redrawn until all the parameters have been processed.

Size and position parameters for rectangles and ellipses refer to the original un-rotated size and position of the shape. The shape is then re-rotated, if necessary. If you want the modified shape to be un-rotated, specify **ROTATE**=0.

Parameters are processed in the order given in the **CHANGESHAPE** statement. If the parameters are given in a different order, the result *may* be different.

Remember that a **SHAPE TEXT** statement is different from a **TEXT** statement. **CHANGESHAPE** only operates on ***shapes***.

Note that the **CENTER**=(x,y) parameter works differently for **TEXT** shapes, in that it refers to the lower-left corner of the defining rectangle.

Note that changing the outline and fill colors of a text shape to the background color will make it invisible. This can be useful to make flashing text or buttons,

Using **ROTATECENTER**="center" for shapes without a defined center (like lines) will rotate about the center value which was in effect when the shape was originally created, unless you override it with a new **CENTER** value. Note that changing the **CENTER** value for such objects does not move the object – it merely changes the default center of rotation.

The **DIRECTION** parameter is used primarily inside Complex Connected Shapes, where the **CHANGESHAPE** statement cannot be used. It will, however, have the effect of inverting a **PIE**, **CHORD** or **ARC** shape, *e.g.*, a 90-degree arc becomes a 270-degree arc.

The **MOUSECLICK** parameter is different here than in the **SHAPE** statement. In **CHANGESHAPE**, specify **MOUSECLICK**=1 to activate mouse click checking for the shape, and **MOUSECLICK**=0 to deactivate it.

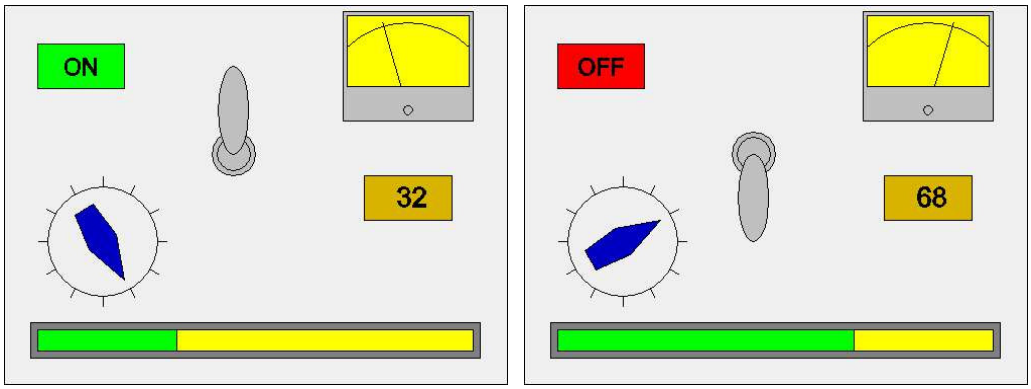
When changing the array of points or count of points for a **POLYGON** or **POLYLINE**, you must specify both **POLYPOINTS**=*array* and **POLYCOUNT**=*count* on the same **CHANGESHAPE** statement, *e.g.*,

> changeshape *id*, polypoints=*array1*, polycount=6.
>
> You must supply the entire array, not just the points that have changed. It can be a totally different array or the original array with some changed values.

Although you can't change or rotate the array used in **SHAPE PLOTCHAROUTLINE**, you can change all the parameters of a **SHAPE TEXT** shape, including characters, font, size, rotation, etc. This will usually give the results you desire.

Applications: Radio buttons, some kinds of animation, games, puzzles, status indicators, progress bars (see below).

The following graphics represent some of the ways you can use **CHANGESHAPE** to show switches and indicators. The program to produce this is in the sample programs as INDICATORS.TXT.



The toggle switch position is changed by rotating the "toggle" handle around the center of the switch. The meter position is changed by rotating the line representing the needle. The progress bar is changed by changing the right side of the green rectangle. The rotary switch is changed by rotating the polygon. The axes and grid lines were suppressed since they did not contribute to the display.