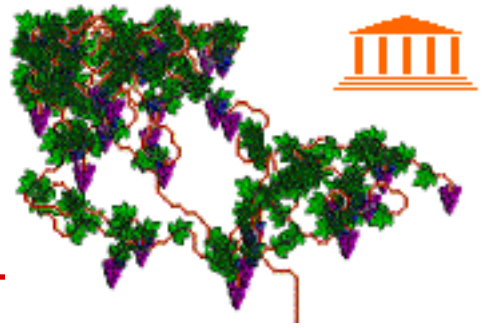

Images Show Concepts

The VINES Project

by John Arscott

Working with images.

Several years ago I decided to write a small program to demonstrate the scope and flexibility of the graphics functions in TrueBasic. To capture the imagination of the user, the program would have to be visually attractive, and use animation. I also wanted to include random elements to sustain the interest of the user and produce a different result each time the program was run.

Eventually I decided to use the growth of a grape vine as the theme of my program which I called VINE for obvious reasons. That was the easy bit.

Before you start any program you need to do a little bit of thinking about how you intend solving the problem in hand. Very often you will jot down a few notes about all the things you need to consider and the order in which you intend to tackle them. You should also devote a little bit of time planning how you intend setting out your program, otherwise it will grow like Topsy and you will have great difficulty finding your way through the logic.

Here is my nice little simple plan:

1. Program housekeeping
2. Prepare the working images
3. Define the rules
4. Enter a continuous loop
5. Check if the user wants to quit
6. Grow a short length of vine
7. Grow a leaf (randomly)
8. Create more branches or terminate at a bunch of grapes
9. Change the direction of growth (randomly)
10. Repeat the loop

Let's just take a look at these 10 steps in a bit more detail.

1. Program housekeeping

Under this heading comes a whole bunch of gritty bits pieces, some of which you have to do and others which you ought to do. If you write a lot of programs you will soon forget what you have written or even what the program is supposed to do, so make a habit of starting your programs with a few REM: statements to

remind yourself of the purpose of the program, when you wrote it, who wrote it (you may be famous one day), when and how it was updated and the current version number, e.g.

```
REM: The VINE program
REM: dated 13 DEC 1996
REM: Author J.R.Arscott
REM: Version 2.0 for TBSILVER

REM: developed from the original VINE PROGRAM dated 12 Nov 1995

! This is a demonstration of superimposing irregular images
! onto a background using a masking technique. This involves
! BOX SHOW using 4 and 7. The blank mask is shown first using 4
! to blank out any colors beneath the new image whilst
! leaving the background undisturbed. The new image is then
! shown using 7.
! The routine make_mask generates the blank mask
! Graphics include 4 leaf arrangements and 1 bunch of grapes
! Vine growth and direction is random
! Branchmax limits the number of simultaneous growing shoots
! Exit program by pressing any key
! TC_event not used because it slows down animation
```

You can then follow this with essential stuff like library statements, any PUBLIC variables you may be using, a list of dimensioned arrays, any DEF functions you may need, e.g.

```
Library "C:\TBV5\TBlibs\TRUEctrl.trc"
Library "C:\TBV5\TBlibs\TRUEdial.trc"

PUBLIC xpix,ypix,trunk,branchmax,message$, leafchance, grapechance
PUBLIC branchx(8),branchy(8),old_branchx(8),old_branchy(8)
PUBLIC direction(8),old_direction(8), trunk, branch
PUBLIC grape$,grapemask$,leaf$(8),leafmask$(8)
```

If you are using True BASIC **Silver** then this is also a good time to initialize the **TrueCtrl library**, to set your working units, to set the **ShowDefault** flag and to create all the windows and objects you may want to use in the rest of the program. I prefer to do this last chore in a separate sub-routine because it keeps things nice and tidy; beside I know where to find it if I need to make any changes, e.g.

```
CALL TC_init
CALL TC_SetUnitsToPixels
CALL TC_ShowDefault(0)
CALL makemain   ! the routine where you create window objects
```

This is what the **makemain** routine looks like:

```
SUB makemain
REM: uses default window-sets size to full screen
CALL TC_Win_SetTitle (0,"The VINE random growth Program")
CALL TC_setRect(0,4,636,476,24)
LET message$="To end this demonstration press any key"
```

```
END SUB
```

The next little task is to display what we have done so far in preparation for manipulating our graphic images, e.g.

```
CALL TC_show(0) ! shows the window created in makemain
CALL TC_win_switch(0) ! makes the window active and target
```

If you are not using Version 5.x then you can ignore all this TC stuff but you will need to set the graphics mode with:

```
SET MODE "graphics" ! for the PC
SET MODE "color" ! for the MAC
```

The True BASIC Silver editons all have paper white screens. If you are using a classic DOS edition of TrueBasic on a PC you will need to invert black and white with

```
SET COLOR MIX (0) 1,1,1 ! this makes the screen white
SET COLOR MIX (15) 0,0,0 ! this is now black
```

Next you need to set a graphics window in whatever version you are using. Since you will often need to do this in Version 5.x I have used a prefix similar to normal TC calls. This routine merely sets the pixel dimensions for the default window which is already open, e.g.

```
CALL TCX_win_open     ! sets graphics window
```

2. Prepare the working images

This heading is a bit misleading because in practice I created the original images on a PC using the “Paintbrush” program, then I converted the BMP images to boxstring format before saving them as

the files LEAF_A to LEAF_D and GRAPE. These files are in boxstring format which differs from system to system, so I have adopted the following convention for filename extensions:

TBX for standard TrueBasic on PCs

MBX for standard TrueBasic on the Mac

WBX for Version 5.x on PCs

VBX for Version 5.x on the Mac

I will explain what the principal differences are later, but for the moment all you need to worry about is making sure you are using the right filenames.

Our first task is to import these files into our program and we do this by reading these image files as ordinary byte files using the routine **read_image**, e.g.

```
LET extn$=".WBX" ! image file extension
LET dir$=""                          ! insert directory if required
```

```
LET filename$=dir$ & "grape" & extn$
CALL read_image(filename$,image$)
LET grape$=image$
CALL make_mask(image$,mask$)
LET grapemask$=mask$
```

This is what the sub-routine read_image looks like:

```
SUB read_image(filename$,byte$)
   CLOSE #1
   OPEN #1: NAME filename$, CREATE newold, ORG byte
   ASK #1: FILESIZE total
   READ #1, bytes total: byte$
   CLOSE #1
END SUB
```

It works by opening a byte file called ***filename$*** (the image file) then we ask the file to tell how big it is ***total*** so that we can read in the correct number of bytes which are stored in the temporary string ***byte$*** which TrueBasic is smart enough to know is the same as our variable so it transfers the contents of ***byte$*** to ***image$*** .

The great thing about boxstrings is that we can now show this image directly on the screen using BOX SHOW anywhere we want without doing anything more to the data. What is more you can read images off your hard drive and show them on screen fast enough for most animation sequences.

Now that we have the bunch of grapes image in the program we need to do one more thing; we need to make a mask of that image. Why so you ask? Good question, so pay attention to the technical stuff because here is the answer.

Basically, if you BOX SHOW any image it will appear on the screen like a postage stamp. What we need is just the shape of the bunch of grapes. We dont need the background to our bunch of grapes to show. On the contrary we want whatever is behind the bunch of grapes to show through everywhere except through the grapes.

The way to get round this problem is to create a mask which is the same shape as the grapes but is completely opaque with a background which is completely transparent. Opaque is represented by a color whose bits consist entirely of 1's (white) and transparent is represented by a color whose bits consist entirely of 0's (black). This mask must be shown with:

```
BOX SHOW mask$ AT x,y USING 4  ! note the use of USING 4
```

If you are still with me we'll go on to the second part. If not don't worry just take it from me and let the sub-routines do the thinking. The next part of the process is to show the image itself on top of the mask with:

```
BOX SHOW image$ AT x,y USING 7 ! note the use of USING 7
```

BOX SHOW using 7 is the full transparency mode. In other words the colors in

your image will be influenced by any background colors. They will essentially show through producing all sorts of mixtures except where the background is white in which case you will see your image in its correct colors. This is exactly what our mask has done for us. It has blanked out the area of the grapes in white, leaving the remainder in their normal colors. Get it? Well maybe not but don'y worry because the sub-routines do it all and they know what's going on. It is not necessary to understand how a sub-routine works, you just need to know how to use it.

Now you know why we need a mask so its time to look at the routine which does the job:

```
SUB make_mask(image$,mask$)
    REM: this routine will vary according to MAC or PC
    LET mask$=image$
    LET width=unpackb(image$,1,8)
    LET height=unpackb(image$,17,8)
    LET bitlength=(int((width+7)/8))*8  ! nearest byte boundary
    LET startbit=64 ! TBX images use 64 - WBX images use 96
    FOR row=1 to height
        FOR col=1 to width
            LET pix$=""
            LET bitposition=startbit+(row-1)*bitlength*4+col
            LET k=unpackb(image$,bitposition,1)
            LET pix$=pix$ & str$(k)
            LET k=unpackb(image$,bitposition+bitlength,1)
            LET pix$=pix$ & str$(k)
            LET k=unpackb(image$,bitposition+2*bitlength,1)
            LET pix$=pix$ & str$(k)
            LET k=unpackb(image$,bitposition+3*bitlength,1)
            LET pix$=pix$ & str$(k)
            IF pix$<>"0000" then ! colour present so blank in mask
REM: PC box strings are on 4 planes each 1 bitfield apart
               CALL packb(mask$,bitposition,1,1)
               CALL packb(mask$,bitposition+bitlength,1,1)
               CALL packb(mask$,bitposition+2*bitlength,1,1)
               CALL packb(mask$,bitposition+3*bitlength,1,1)
            END IF
        NEXT col
    NEXT row
END SUB
```

This is the routine for the PC where the image data is stored in four consecutive bitfields. In simple terms the routine scans the whole image pixel by pixel in rows and columns to check if the image has any color other than the background. If so it paints a white pixel in the corresponding position in the mask.

The pixel row and column are used to calculate which bit in the boxstring we are looking at ***bitposition*** . The size of the bitfield ***bitlength*** determines where the other planes are located to give us all four bits which make up the pixel color. These four bits are separately extracted using the function UNPACKB. If these bits are anything other than all zeros then the pixel must have a color so

the corresponding bits are set to 1 in the mask boxstring using the routine PACKB to insert these 1s.

Each boxstring carries a series of bytes containing information about the size of the image which this routine reads. As a result you can use this routine to generate masks for any boxstring. Just feed it an image and it will send back a mask.

If you are using a PC with Version 5.x the ***startbit*** number must be changed to 96. If you are using a Macintosh you will need to use the sub-routine **make_mac_mask** instead because these boxstrings are not formatted the same way as the PC.

The Mac boxstrings are much larger because they service a larger color range and because the string also contains a full 256 color palette. In fact the actual image data only begins after the first 2108 bytes which is why the ***startbit*** is set at 16864 (bits). The remainder of the routine is similar to the PC routine in that the whole image is scanned by rows and columns. Instead of bit planes the Mac images use a complete byte to determine the color so the mask is prepared one byte at a time.

So we now have the grapes in the bag, but we still need the leaves. Why so many you ask? Well the number of leaf positions is quite arbitrary but I chose four positions mainly on the basis that it was easy to draw one leaf and flip it horizontally and vertically to create leaf images in the other positions. More of this later. For the moment let us concern ourselves with reading these four images into our program. In fact all we need to do is repeat what we did for the grapes, and just change the filename four times, e.g.

```
FOR n=1 to 4
    LET filename$=dir$ & "leaf_" & chr$(n+64) & extn$
    CALL read_image(filename$,image$)
    LET leaf$(n)=image$
    CALL make_mask(image$,mask$)
    LET leafmask$(n)=mask$
NEXT n
```

Note how the filename has been generated using the counter (n). By adding 64 to the counter and converting this number into a character from the alphabet we can generate the letters A through D.

3. Defining the rules

In this section I will set the rules which govern how the vine will grow and what determines whether it grows leaves or produces bunches of grapes. In other words the rules of the game.

Rule 1 The vine must grow to a minimum height ***trunk*** before it can form leaves or grapes or make extra branches.

Rule 2 The vine stem grows in short lengths ***stemlength*** before it can change direction.

Rule 3 The maximum number which can grow simultaneously is determined by the type of computer which running the program. For most machines a value of ***branchmax*** = 4 works well. This can be increased on faster machines.

Rule 4 The direction the vine stem grows is random but limited to 45 degrees either side of its current direction. A total of 8 directions is therefore possible.

Rule 5 The vine must not grow beyond the edges of the window or below the minimum height ***trunk*** .

Rule 6 Leaves can only grow at the end of a period of stem growth ***stemlength***

Rule 7 Leaf growth is random. The percentage chance of a leaf growing is determined by ***leafchance*** .

Rule 8 The leaf axis is in line or at 45 degrees to the direction of stem growth. A total of four leaf images are therefore necessary to cover the 8 stem directions.

Rule 9 Grapes can only grow when the stem direction is down.

Rule 10 Grape formation ***grapechance***. is random. Grape formation automatically stops the stem from growing any more.

So let's define all our starting parameters, e.g.

```
REM: set up growth start position
LET trunk=150                ! minimum height of main vine stem
LET branch=1                 ! number of branches currently growing
LET branchmax=4              ! maximum number of branches
LET stemlength=8             ! one pixel per unit = 8 pixels
LET direction(1)=1           ! upwards
LET branchx(1)=xpix/2        ! start position for main stem (x)
LET branchy(1)=ypix-10       ! start position for main stem (y)
LET leafchance=30            ! percentage chance of leaf growth
LET grapechance=50           ! percentage chance of forming grapes
RANDOMIZE                    ! seeds the random number generator
```

4. Enter a continuous loop

After all that hard stuff you are going to like this bit. Just one thing before we do the loop, perhaps it would be a good idea to let the user know what to do. We can keep it simple like “Press any key to quit”. If you are using Version 5.x then you can use a message dialog box to show this instruction, e.g.

```
CALL TD_Message("INFORMATION",message$," OK ",1,result)
```

We are not really interested in the response from the user who only presses the OK button to clear the dialog box so we have no need to test the variable ***result*** .

If you are working with standard TrueBasic then a PLOT TEXT statement will be sufficient, e.g.

```
PLOT TEXT message$ AT 30,ypix-30
```

Now we can start the loop going with:

```
DO
   REM: The rest of the program goes here
LOOP
```

Here is a little tip for you. There will be times when you write programs which contain complicated nested DO LOOPS or nested IF THEN structures and it is very easy to lose your way. Every time you write a DO make some space and write a LOOP as well, then carry on writing your program in the space you have made. Do this for IF THEN and END IF as well as FOR and NEXT loops. This way you will always have the correct number of loop terminations. This tutorial follows this practice so that you can see more clearly how each part works.

5. Check if the user wants to quit

The rest of the program is inside a DO loop and each time the loop is cycled we can check to see if the user wants to quit with:

```
IF key input then ! check if user wants to end demo
   EXIT DO
END IF
```

This checks to see if a key has been pressed and if it has key input becomes true so the EXIT DO instruction is executed. Version 5.x users could have used TC_event here, but this particular all singing and dancing routine is so busy doing other things as well that it slows down our program and makes the animation a bit lumpy.

6. Grow a short length of vine

At this point we need to think about what our program is going to look like when it is running. In other words we need to account for the speed of animation. We

don't want things to go so slow like watching paint dry, on the other hand we don't want things to fly about like an F111. My intention was for the vine to grow like time lapse photography. To get that effect we need to grow each branch by one pixel each time and we need 8 of these units to make up a ***stemlength*** . Once we have grown a stem we can think about adding leaves and grapes, then we go round the loop again.

If we grow each branch by a stemlength then move on to the next branch the effect on the screen will be sudden bursts of jerky action in each branch. On the other hand if we grow each branch in turn by just a small unit and keep repeating this unit all have grown a stemlength then the result will be nice smooth action all round. This defines which way round we arrange our FOR NEXT loops, e.g.

```
FOR growth=1 to stemlength              ! growth increments
   FOR b=1 to branchmax     !  most frequent inner loop
      IF branchx(b)<>0 and branchy(b)<>0 then ! is stalk growing
         CALL drawbranch(b)     ! draws branches
            REM: The rest of the program goes here
         END IF
      NEXT b
   NEXT growth
```

The variables ***branchx (b)*** and ***branchy (b)*** are the current screen co-ordinates of the branch number ***b*** . If branches are not growing then these two variables are zero which is why we check to see if the branch is growing before we start drawing.

The variable ***branch*** keeps track of how many branches are growing and at the start we set the value to 1 so only one branch will grow at first until the rule is satisfied for more branches to grow.

The routine drawbranch doesn't actually draw the 1 pixel increments of stem growth, instead it checks the current direction of growth and then calls the appropriate drawing routine. You will recall that Rule 4 requires the stem to only grow in any one of eight possible directions. I have numbered these from 1 through 8 clockwise, starting at noon. This is why the initial direction for the first branch was set at 1.

Instead of eight separate drawing routines I have simplified them to just three; straight on, left or right. They are all very similar so we just need to look at one, the straight ahead version:

```
SUB branch_on(x,y,b)
    LET x=branchx(b)
    LET y=branchy(b)
    SET COLOR 6 ! muddy yellow
    PLOT x-2,y
    SET COLOR 4  ! dark red
```

```
        PLOT x-1,y
        PLOT x,y
        PLOT x+1,y
        SET COLOR 15 ! black-remember we reversed it
        PLOT x+2,y
    END SUB
```

The stem increment is 5 pixels wide with a light pixel on the left and a dark pixel on the right to give the stem shadow.

Note that SET COLOR 15 is only necessary if you are using standard TrueBasic and you have reversed black and white to give you a white background. In all other cases you would use SET COLOR 0.

When control passes back to the drawbranch routine the current values of ***branchx(b)*** and ***branchy(b)*** are up-dated to the latest position. The ***direction(b)*** variable remains the same until we have drawn a complete stemlength because Rule 2 says we can only change direction at this point.

7. Draw a leaf randomly

First of all we need to check that we have drawn a full stemlength, e.g.

```
IF growth=stemlength then      ! end of growth increments
           CALL drawnewleaf(b) ! draws leaves
           CALL remember(b)    ! remembers old positions
           REM: The rest of the program goes here
END IF
```

If only things were that simple. We have a small problem. Our lengths of stem are just like sticks and if we change their direction we will have little gaps left because each stem is 5 pixels wide. We need a routine which fills in these gaps at the nodes because leaf growth is random and there will not always be a leaf to hide these gaps. This is why we need the routine branch_node. This routine is very similar to the normal stem drawing routine except that it essentially draws a rectangular knob on the end of each stem.

Now we can get on with the interesting bit of drawing leaves with the routine drawnewleaf:

```
SUB drawnewleaf(b)
    REM: draws leaf at previous node
    REM: leaf type depends on direction of growth
    REM: chance of leaf growth depends on random number
    REM: cannot grow leaves on main trunk

    IF old_branchx(b)=0 and old_branchy(b)=0 then
    ! branch not growing
       EXIT SUB
    END IF
```

```
      LET leaf_percent=100*rnd  ! random leaf growth if<leafchance

      IF old_branchy(b)<ypix-trunk and leaf_percent<leafchance then

         SELECT CASE old_direction(b)
         CASE 5,6
               LET lx=old_branchx(b)-20
               LET ly=old_branchy(b)+20
               LET leafnumber=1
         CASE 7,8
               LET lx=old_branchx(b)-20
               LET ly=old_branchy(b)+15
               LET leafnumber=2
         CASE 1,2
               LET lx=old_branchx(b)-5
               LET ly=old_branchy(b)+10
               LET leafnumber=3
         CASE else
               LET lx=old_branchx(b)-5
               LET ly=old_branchy(b)+20
               LET leafnumber=4
         END SELECT

         BOX SHOW leafmask$(leafnumber) at lx,ly using 4
         BOX SHOW leaf$(leafnumber) at lx,ly using 7
      END IF

END SUB
```

As you can see, this routine:

(a) checks to see if the branch is still growing

(b) generates a random number for leaf growth

(c) checks to see if the branch is above the trunk line

(d) checks the direction of growth to select the right leaf image

(e) shows the mask and leaf images

Note that this routine uses ***old_branchx(b)*** and ***old_branchy(b)*** . These are the co-ordinates of the previous stemlength not the current one. After all, leaves grow from nodes not from the end of the stalk.

The other point you will notice is that these co-ordinates are modified by various numbers depending on the leaf position. The reason for this is that all images are referenced to the bottom left hand corner of the original rectangular image. We want the leaf to be positioned in a realistic manner in relation to the stem node so these numbers are necessary to adjust the leaf image so that it looks right on the stem.

Under Rule 8 we are allowed to use one leaf to cover two stem directions hence the SELECT CASE uses pairs of directions. Once again we are using ***old_direction(b)*** because this is the direction the previous stem was growing and not its current direction.

The variable ***leaf_percent*** is a random number between 0 and 100. This routine checks that this number is less than our defined limit of ***leafchance*** i.e. 30%

8. Grow more branches or terminate with grapes

In our main program this is yet another simple sub-routine call:

```
CALL more_branches(b,branch)       ! creates more
```

Here is the routine more_branches:

```
SUB more_branches(b,branch)
    IF direction(b)=5 then          ! select newbranch or grape
       IF branch<branchmax then     ! make more branches
          CALL newstalk(branch,b)
       ELSEIF branch>1 then ! saves the remaining branch
          CALL grow_grapes(branch,b)  ! also kill branch growth
       END IF
    END IF
END SUB
```

This routine checks to see if the current direction is downwards (direction number 5) and if so it first all checks to see if the full quota of braches are growing - if not it creates a new growing stem using the routine newstalk:

```
SUB newstalk(branch,b)
    REM: tacks new branch onto existing branch
    FOR j=1 to branchmax
        IF branchx(j)=0 and branchy(j)=0 then
           LET branchx(j)=branchx(b)
           LET branchy(j)=branchy(b)
           LET branch=branch+1
           EXIT FOR
        END IF
    NEXT j
END SUB
```

If the branch quota is full then it checks out the chance of growing a bunch of grapes using the routine grow_grapes:

```
SUB grow_grapes(branch,b)
    LET grape_percent=100*rnd   ! random grape growth

    IF grape_percent<grapechance then
    BOX SHOW grapemask$ at branchx(b)-13,branchy(b)+34 using 4
    BOX SHOW grape$ at branchx(b)-13,branchy(b)+34 using 7

    LET branchx(b)=0              ! indicates growth has stopped
    LET branchy(b)=0              ! indicates growth has stopped
    LET branch=branch-1           ! number of branches still growing
    END IF
END SUB
```

Notice that growth is stopped by declaring the branch co-ordinates are zero and that when this is done we need to up-date the counter which is keeping track of how many branches are still growing, i.e. ***branch***

9. Change the direction of growth randomly

So far we have grown branches, spread some leaves around or grown bunches of grapes. What we need to do now is slightly change the direction of growth. After all, plants don't grow in straight lines. WE do this with the routine change_direction, e.g.

```
CALL change_direction(b)      ! changes direction
```

Here is the routine itself:

```
SUB change_direction(b)
    REM: check to see if branches are near window edges
    REM: if so ignore random direction change and just deflect
branch

    IF branchx(b)<40 then           ! left hand side  of window
       IF direction(b)=1 or direction(b)=8 or direction(b)=7 then
          LET direction(b)=direction(b)+1
       ELSEIF direction(b)=5 or direction(b)=6 then
          LET direction(b)=direction(b)-1
       END IF
    ELSEIF branchx(b)>xpix-40 then      ! right hand side
       IF direction(b)=4 or direction(b)=5 or direction(b)=3 then
          LET direction(b)=direction(b)+1
       ELSEIF direction(b)=1 or direction(b)=2 then
          LET direction(b)=direction(b)-1
       END IF
    ELSEIF branchy(b)<50 then       ! top of window
       IF direction(b)=3 or direction(b)=1 or direction(b)=2 then
          LET direction(b)=direction(b)+1
       ELSEIF direction(b)=7 or direction(b)=8 then
          LET direction(b)=direction(b)-1
       END IF
    ELSEIF branchy(b)>ypix-trunk then  ! top of main trunk
       IF direction(b)=5 or direction(b)=6 or direction(b)=7 then
          LET direction(b)=direction(b)+1
       ELSEIF direction(b)=3 or direction(b)=4 then
          LET direction(b)=direction(b)-1
       END IF
    ELSE

       LET new_direction=int(rnd*10)  ! random change of direction

       IF new_direction<4 then
          LET direction(b)=direction(b)-1
       ELSEIF new_direction>6 then
          LET direction(b)=direction(b)+1
       END IF
    END IF
```

```
      REM: check new directions do not fall outside limits
      IF direction(b)>8 then
         LET direction(b)=direction(b)-8
      ELSEIF direction(b)<1 then
         LET direction(b)=direction(b)+8
      END IF

END SUB
```

You can see that this routine first of all checks to see if the branch direction complies with Rule 5 and is not about to grow outside the window or below the trunk height. If the growth is within the limits then the routine uses a random number newdirection between 0 - 10 to decide which way the stem will grow next. If the number is less than 4 then it will go right by 45 degrees and if it is greater than 6 then it will go left by 45 degrees if the number lies between then the branch will just keep going on its present course. Finally we have a double check to make sure we have not produced a direction which is outside the limits. In other words if the direction is less than 1 or greater than 8.

10. Repeat the loop

Because we took the earlier precaution of adding loops and end if statements all along the way, now that we are at the end of the program we need do no more.

Once the user has elected to quit the program it is good practice to clear up our mess and return everything just as it was. Version 5.x users will know that mopping up is done with:

```
CALL TC_cleanup
```

Standard TrueBasic users working with a PC should return the normal colors with:

```
SET COLOR MIX (0) 0,0,0 ! this makes the screen black
SET COLOR MIX (15) 1,1,1 ! this is now white again
```

All the sub-routines are located at this point in the program followed by:

```
END
```

Now it is time to check out the program listing to see what it all looks like when everything is put together. Then run the program to see how the vine forms.

John Arscott
26 October 1998
engine@clara.net