Ruthie!

A Non-Verbal Game for Children Who Can't (Yet) Read

Table of Contents

Introduction

Ruthie Shipps is a little girl in suburban Washington, D.C. She's my niece, and she's brain-damaged. She has a "language processing deficit." A good analogy for a child with a language processing deficit is being a GW-BASIC interpreter trying to deal with QB (or VB) code. You can understand little bits of what's going on, but only if it's given to you in exactly the form you understand, with no special inflection or emphasis. If Ruthie understands "Sit down, Ruthie!" she won't necessary understand "Ruthie, sit down."

One afternoon Ruthie was at my mother's house, and discovered my mother's computer. Somehow or other she ended up with Windows Paint on the screen, and she was entranced. She spent hours in front of the screen, sweeping swashes of color across the picture area and giggling with delight. My mother called me with the news, and asked if I could find a game that would help Ruthie learn to use the mouse. Ideally the game would have these attributes:

* It would be completely non-verbal--Ruthie would be able to launch the game from the Program Manager, play the game, and quit without having to recognize letters or words.

* It would be able to run Maximized, so Ruthie couldn't accidentally move, resize, or launch other applications once her game was started.

* It would involve simple mouse skills. As Ruthie got better, perhaps the game controls would get smaller, and perhaps she could learn to drag controls around.

I'd just ordered Visual Basic, and thought this would be an excellent opportunity to do a little programming with my new toy...er, investment. Working from the general outlines my mother had dreamed up,

I set to work.

I initially figured this would be a one-form project, so I named the first game Ruthie.Frm. If you look at the Maintenance History in the RUTHIE.BAS file you'll see that this project went through quite a bit of development before I was ready to upload it to CompuServe. I've subsequently added two more games, and have met my mother's design criteria. Along the way I've added a few touches--but I'll explain that below.

People of all ages respond to success. When I worked for a computer software company I developed a training program for new users. Each session always ended with a "success point"--some activity or achievement that would let the trainees go to lunch or to the hotel satisfied that they had accomplished something good. It's particularly important to let children know that they've succeeded--attention spans are very short, and they'll lose interest quickly if they can't make the object (book, game, toy, computer game) do something Right Now. My company develops books for *Highlights for Children*, the largest children's magazine on the planet. "Celebrate your child's successes!" was a key theme of the *Highlights* founders, and it's a phrase you'll hear from the magazine's editor today. It's an important point.

I bring up success because when the child successfully achieves the desired task (clicking on the red square, or something else) the program displays a big picture with a "success message." If you play the game with a child, PLEASE exclaim the phrase on the picture. Don't just read it--SHOUT IT! "Hooray!" "Good Job!" "Yippee!" A child who can't read won't necessarily learn to read when she sees the words--but she'll know that you're very proud of her. And that will make the biggest difference in how your child does in her career in school.

This document serves several purposes: First, it simply documents the code--undocumented applications are worthless the week after they're written; Second, it gives you a glimpse of what I set out to do, and how I accomplished it; Third, it can serve as a model for documentation for the VBCT project--the CompuServe access project that's being developed in Visual Basic in Section 9 of the MSLANG forum.

This introduction gives you the overview--the idea, the goals, the sweeping generalizations. In the pages that follow we'll work through each of the forms, and each of the event procedures. The emphasis is on what's happening, and why. Novice users will find a neat trick or two (plagiarized wholesale from some of the gurus on MSLANG); experienced users will be able to see quickly that I have much to learn about programing.

Several key elements in RUTHIE were contributed by members of the MSLANG forum on CompuServe. Keith Funk responded to a plea for help

with a lot of helpful code, and I learned a clever way to shuffle the elements of the control array in GAME3.FRM from somebody else--but I deleted the message before I jotted down his name. You should assume that all flashes of brilliance come from the MSLANG gurus. The booboos, bumbling, and blunders are mine, all mine.

Finally, this isn't commercial software. This isn't going to turn up in the Games section of Windows 3.1, or in the next edition of Microsoft's Entertainment Pack. I wrote it as a learning experience, and to entertain (and encourage) a little five-year-old girl. If you learn something from it, I'd appreciate hearing from you--it's nice to know. If you find the code so poor that it's funny, at least let me know that you had a good time. And if you have a four-year-old handy, and play the game, please consider sending $5 to Ruthie's school. The address is shown on FORM2.FRM (About Ruthie). We've started a fund to buy books for their library. If you send them a contribution (which is tax-deductible, by the way) they'll be able to buy more books for handicapped children. Being in the publishing business, I'm all for libraries buying more books <grin!>.

Have fun!

John Murdoch, Wind Gap, Pennsylvania, July 1991
CompuServe 71507,1212

Global Declarations (RUTHIE.BAS)

It's important to keep track of the history of an application. The Maintenance History form is a good way to keep track of what has happened when, and who did it. In this instance the "who" is a little pointless--I'm the only programmer at Murdoch Books. But it's the style we used when I worked for a software company, and old habits die hard.

There aren't many global constants to keep track of. The Boolean constants help make things a little more readable. The HEADER constant lets me make a quick change if Bill Gates calls up wanting to license the game for Windows 3.1 <dream on>. HOWLONG defines a global constant for "how long" the sucess messages and sad faces are displayed.

OPENINGF.FRM (The Opening Form)

The opening form simply tells the user what the game does, and provides the usual information. There are two little twists:

* The picture in the upper left corner is a "version icon." During the initial schmoozing stage of the VBCT project we discussed using different pictures to identify specific versions of the finished product. The most seasoned heads seemed to poohpooh the idea. But I like it, and this application needs to be non-verbal.

* The second trick (which is repeated through the application) is that clicking on the picture has the same effect as clicking on the "Start" button. The child simply has to learn that clicking on the little picture--any little picture--will bring a game up.

And by the way, the little copyright symbol--"©"--is ANSI character 169. You can enter it on your next app (and you should) by holding down the ALT key and typing "0169" on the numeric keypad on your computer.

About Ruthie... (OPEN2.FRM)

The form is always loaded as Modal--meaning that so long as you see it, you can't access any other RUTHIE form. You'll also note that the Minimize and Maximize buttons have been removed. Clicking on the Continue button (or on the photo) will hide the form, and return you to wherever you called this from. (The form can be called from the Opening Form, or from any of the games.)

The picture is a BMP. It began as a TIFF file in Astral Picture Publisher. It was output as a PCX file and imported into Windows Paint. From there it was saved as a BMP file. Along the way it grew from a 4K TIFF into a 93K BMP. Yup--93K of RAM for that tiny little picture.

Mark Novisoff (of MicroHelp, a leading vendor of Basic add-on tools) mentioned that BMP files compress quite well with PKZIP. Even so, I'd think twice about using photos in VB apps--at least until Microsoft is able to directly support TIFF files.

Game 1 (RUTHIE.FRM)

When the game is loaded, the Form_Load event calls a subroutine named Initialize. Initialize defines a custom scale for the form, and places each of the visible elements.

Why go to the hassle, instead of simply defining their properties at design time? Several reasons. First, different users have different monitor sizes. What works on my monitor (a NEC 4D) won't necessarily work on my mother's 14" Super VGA, and won't begin to fit on my wife's 12" EGA monitor. So the Form_Load procedure defines the form size (in terms of the screen dimensions) and then tells Initialize to set up the visible form.

If you're using pictures on your form--especially if you're using big pictures, as I am--you may initially find yourself moving controls around trying to find the little jasper you want that's invariably down at the bottom of the heap. If you have the app size and position all the controls at runtime, you can shrink your pictures to a manageable size and store 'em off to the side, out of the way.

Actions:
The "actions" (Stop, GoToGame2) are stored on a picture box to make the initial screen setup easier. Because they're icons they never change in size--it's easier to put them both on a little "tray" and move that--it's one set of code instead of two (or three in Game2).

Background:
In Game 1 and 2 it's called Background. In Game 3 it's the Success Picture. When the user successfully clicks on the red square (Picture 1) she sees the Background displayed.

The "success picture" for each game is identical to it's version icon. When the version changes, the success picture will change as well.

Menus:
"If this is a non-verbal app, why do you have menus?" For the parents. If a child wants to play Game 3, it's a drag to go to Game 2, and then to Game 3. The menus let you move a little more quickly. Using menus also lets me give the user access to instructions (another help for parents) and to the About Ruthie... form. Further, Windows users expect FileExit--novice users (like my mother) get lost if they can't find it. The Stop icon isn't Windows-standard, and Control Close is a kludgy way

to quit.

Game Menu:
Call me flippant. Rather than just ignoring the user who clicks on Game 1, we give him a little smart-aleck comment. A little something for parents who take a peek at Junior's game late at night.

GoToGame2 and GameGame2, GameGame3:
These controls just hide this form, and load the next one. Yes, they're redundant. Should I "simplify" the code by having GoToGame2 call the GameGame2_Click event procedure? In theory, yes. But in practice doing so doesn't really save anything--both instructions are one line, and doing it directly saves a subroutine call. In addition it's easier for you to understand what I'm doing--you don't have to shuffle off to another subroutine to find out. If it's a big hunk of code (Initialize) then I'll call the sub. But for a one-liner, I'd rather do it directly.

InstructionsGame1 (INSTGAME1.FRM):
The form shows as modal. When it is unloaded (by clicking on Continue, or on the picture) it hides the form. Why not Unload the form instead? I could, I suppose. But leaving it up makes it quicker to refer back to it, and I don't think freeing up system resources is a big deal with this app. Nobody's going to be recalculating an Excel worksheet in the background while they let Junior play Ruthie (if someone does, he gets what he deserves!).

Live Area:
Game 1 and Game 2 have a "Live Area". The Live Area defines all the places the target picture (Picture1) can move to. Using a Live Area prevents the possibility of having the picture randomly move over the top of something like the Stop icon. We want the Live Area to be transparent to the user--clicking on the Live Area gets you exactly the same result as clicking on the Background.

Picture1:
The red square. Picture1 is "contained" in the Live Area. When it is clicked the Background is turned on. Then we want to determine the available space within the Live Area, and have the picture randomly move there.

* The Visual Basic Programmer's Reference doesn't do a very good job of explaining Scale, ScaleWidth, and Width, and the differences between them. It's very important to know if you're going to be placing (and expecially moving) controls around. See the "Useful Things I Learned Along the Way" section below for more.

* You may have noticed, earlier, that the last line in the Initialize subroutine was "Randomize". If you ask VB for a random number, you'll get one. If you ask for a series of them, you'll

> get a series. If you then ask for a random number the next time you load the app, you'll get exactly the same series. For my purposes this was not a feature. "Randomize" recreates the random number series every time the form loads (or is resized)--so the picture will always move in a different pattern. (Kids with video game experience are amazingly quick at picking up recurring "random" patterns--it's the key to success for most video games.)

Timers:
The Timers turn off either the background or the sad face. A nagging little problem I discovered early on was that when I moved Picture1 so that half of it was visible underneath the Background, part of it would not redraw when Timer1 turned Background.Visible = FALSE. That's why I executed Picture1.Refresh at the end of Timer1.

The Visual Basic Programmer's Reference describes this sort of problem on page 176, in the context of persistent graphics and the AutoRedraw property. The manual points out that the AutoRedraw property costs memory (they give a pretty big hint that you shouldn't use it unless absolutely necessary--they clearly think that a little extra work for the processor is better than using a little more memory). The manual suggests using the Paint event to trigger a Refresh or some similar action (they're talking about drawn graphics). I'm just adding it to the Timer, rather than adding another event procedure.

Game 2 (GAME2.FRM)

As with Game 1, the Form_Load procedure defines the form area, and calls the Initialize subroutine to set the scale and place the controls on the form.

> Note: the Initialize routine for Games 1 through 3 are not the same. Also, for some dumb reason, I called the Game 3 procedure InitGame3. I suppose I should have called them InitGame1, InitGame2, etc. A journeyman programmer will tell you that there's a lot of wisdom in defining your variable names, styles, and subroutine names in advance. I winged it, and it shows.

As I mentioned above, I'm placing all the controls in the code so that the form can be resized. Game 2 is the most "stretchy" of all three. If you resize the game, everything, including the "thermometer" changes. Try it! But there's a cost to this technique--try changing the proportions of the game. Make it as wide as your monitor, and half the height. Now zip through eight successful hits on the target picture, and view the success message. Looks like something from one of those distorted sideshow mirrors, doesn't it? For this application I don't mind--but it's something you should be aware of.

Let's discuss the idea of custom scales for a minute. Note that in Game 1 I used a custom scale of (75, 100); in Game 2 I'm using a scale of (75, 50). (You'll recall that the first element in the scale is the x-scale, the second element is the y-scale.) In both games (and in Game 3) I want the scale units to be roughly equivalent, so that a 10 by 10 box is a square. The Programmer's Reference gives you an example of a custom scale that's (100, 100). That's nice if you have a square form, but. Use a custom scale to make a clear break in your mind from your monitor--the user's monitor won't be the same. If you use twips or pixels for form placement, you're just looking to irritate the user (we'll use twips later on--stay tuned).

Initialize:
The Initialize routine is similar in purpose to Ruthie_Initialize, so I won't go over everything here. I do just want to mention that I force a Refresh of all the controls down at the bottom of the routine. I call Initialize when the form is loaded, and when it is resized. I found in testing that if I resized the form by pulling on a side or corner border, the controls moved around and redrew without a problem. But if I maximized the form, some of the controls (in a seemingly random fashion) looked funky. A bug? Maybe. But this cures the problem.

Background:
As with the other games, clicking on the Background (the live area) or the form gets a Sad Face. Note that I called it a Frown Face here--see my note above about the perils of winging one's variable-naming conventions.

Form:
The Load procedure defines the startup screen size and shape, and calls Initialize. Why didn't I include the Picture1.Picture assignment in Initialize? Because I wanted the current picture to remain the same in the event of a Resize. I may at some point in the future include a random selection of pictures--we'll see.

The Form_Resize event
Here is where using twips for the scale makes a difference. Game2 drove me nuts trying to deal with the problem of resizing the form, when your "resizing" action was to minimize the form. I kept getting an error message at compile time, for stack overflow. Apparently I was making an invalid assignment, but I'm not sure what's going on. (This is something I'm still trying to pin down. For now I've eliminated minimize buttons from all the Ruthie forms.)

Menu Commands:
These are essentially the same as the menu commands for Game 1. Note that we give a slightly different smart-aleck message if the user tries to "go to" Game 2. Think of it as "context-sensitive flippancy." No extra charge.

Picture1:
If Picture1 is successfully clicked, we want the "mercury" in the thermometer to increase. In actual fact the thermometer is Picture2, and the mercury is Picture3. Picture3 is contained by Picture2.

First off, we define the increment by which the mercury increases as one-eighth the height of Picture2. We increase the height of Picture3 that amount, and then move Picture3 up. (If we reversed the order, the mercury would look weird--it would move up, and fill in below. If you increase something so that it projects out of its container nothing happens--you just can't see it. In this case that's a good thing.)

If the new Picture3.Height is almost equal to the height of Picture2, then we're ready to show the sucess message and move the control. "But why," you ask, "are you asking if Picture3 is almost the height of Picture2? Shouldn't it be exact?" There's a small rounding error when we divide the size of Picture2. I might be able to solve it with integer division ("\" instead of "/") and some clever code to round up (100 \ 8 would equal 12, for instance), but this seemed more direct.

Once the success message comes on, we move to the next picture. In this game we don't randomly pick a new picture--instead we just rotate through the four possibilities. Here's how:

```
      Counter = (Counter + 1) Mod 4             'Rotates the counter
```

This is a one line way of saying:

```
Counter = Counter + 1
If Counter >= 4 Then
      Counter = 0
Else
End If
```

It's a little less obvious, but it takes less time. Work it through in your mind, and you'll see that it works:

      If Counter = 0, the new Counter is 1;
      If Counter = 1, the new Counter is 2;
      If Counter = 2, the new Counter is 3;
      If Counter = 3, the new Counter (4 mod 4) is 0.

It's nice to call this "elegant" programming. Just be careful that your programs aren't so "elegant" as to be unreadable.

Once the picture has been changed we reset the thermometer to its original size and set the timer.

Timers:
Timer1 turns off the "FrownFace" after an interval of HOWLONG--Timer2 turns off the success picture after HOWLONG. I could write this more efficiently with one Timer:

```
Sub Timer1_Timer()
      FrownFace.Visible = FALSE
      Yippee.Visible = FALSE
End Sub
```

There's little extra work for the program to turn off a picture that's already off....

Game 3 (GAME3.FRM)

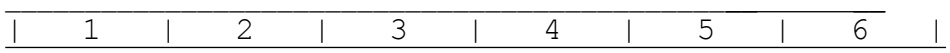
The Game:
This game is significantly more complicated that Game 1 or Game 2. Instead of just finding the target and clicking on it, the user has to find a picture, find it's matching pair, and drag the "source" picture over the "target" picture. To further complicate things, the pictures are icons--they're not Window Metafiles (which most of the other pictures are). (We'll discuss why these pictures are icons in "Little Things I Learned..." below.) Consequently these pictures are smaller. The user's parent should plan on doing a little coaching with this game. The opportunities for failure are greater.

When the user clicks down on the source picture, the DragIcon property shows a similar icon--but not the identical one. In testing so far, no child has been confused by this--they've all understood which picture in the target array they were looking for. It's a little feature--that something extra that adds a little pizzazz.

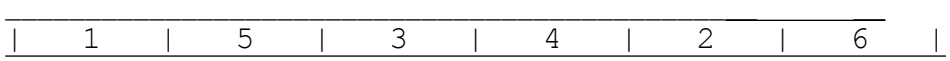
If the user drops the icon too soon, or on the wrong picture, a SadFace appears. If the user clicks on the form, but not on the source picture, the sad face appears. I considered using different sad faces to trap for different events (a control array named SadFace(), coupled with a matching subroutine called SadFaceVisible--each different event would just pass a value for the index). I chose to hold off on the idea for now--I'd decided to upload this to CompuServe, and I was getting wary of making the ZIP file too big for many folks' download budgets.

InitGame3:
We begin by declaring an array into which we'll assign the numbers 0 to 5. Then we'll pick two random numbers from 0 to 6, and swap the values in those array positions. An example:

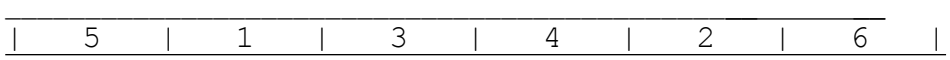
When we begin, the array looks like this:



We'll randomly select two array positions, say 2 and 5, and swap their contents. Now the array looks like this:



Now let's suppose that the next pair to be swapped is pairs 1 and 2. When that's done, the array will look like this:



Do that 28 more times, and you've got a pretty thoroughly shuffled array.

Once we've shuffled the array, we'll use that array to assign pictures to the Target() picture array.

Note: My Storage() array has 12 pictures, not 6. That's why the assignment multiplies the array number by 2. The Storage() array is sorted with the DragIcon immediately following a given picture.

Now that we've loaded the Target() array, we need to identify the SourcePicture--we do that by randomly selecting one of the Storage() pictures. We then define the DragIcon as whatever the following picture is.

The rest of InitGame3 simply defines the location and placement of the elements on the screen. Note that I didn't call the game background Background this time (another example of poor planning)--I called it Picture1. This is particularly dangerous because Picture1 in Games 1 and 2 is an entirely different control, akin to SourcePicture in this game.

Form:

You may recall that earlier I preached on the virtues of custom scales, as opposed to using twips. (I don't think the Microsoft documentation people ever grasped Scale--the Programmer's Reference clearly steers you away from custom scales in favor of staying with twips. They don't even give you any examples of why you'd want to use them.) Here's where twips come to the fore.

You will probably remember that icons are a fixed size--if you've played with IconWorks, IconDraw, IconMagic, or any of the other shareware/freeware/bundleware icon makers, you'll have realized that

nobody's ever given you the option to resize the puppy. If you resize an app that's chock full of metafiles, the metafiles just get squeezed or stretched. If you resize an app with a lot of icons, some of those icons might just disappear. This game would be a little tough for a five-year-old to play if the Target() array wasn't visible, so I used the Form_Load and Form_Resize procedures to make sure nobody made the form too small.

In the Form_Resize procedure I test to see if the size has been reduced either vertically or horizontally. Depending on what happens (and it will catch both circumstances if the user has used a corner border to shrink the form) the user will have a message pop up correcting him, and resetting the app size.

SourcePicture:
The SourcePicture.DragMode property is set to Manual. I could have set it to automatic, but then I can't take advantage of MouseDown events. At present I don't--but in the future I'd like to add VBTools's nifty music functions to play a little "ta dah!" jingle when the picture is clicked on (I'll include a Preferences dialog box so parents can choose to turn the sound off<!>).

When the MouseUp event occurs dragging stops (that is, we "drop" the SourcePicture.DragIcon). I don't move the control's final position--I just want the user to drag it over the correct control, or start over.

Timers:
Only Timer1 is used at present. I originally used two timers, but found that (as I described at the end of Game 2) I could use one timer to achieve both results. I haven't killed off Timer2 yet, and I won't until I'm sure that I'm not calling it from someplace else.

Useful Things I Learned Along the Way

Consistency and Planning:
One of the beauties of Visual Basic is the way that it implements the Windows form. The "form" is in one window, while the code for the form is in another. You can do lots of interesting and creative things.

Traditional "structured" programming doesn't work that way. A well-written program starts at the top and works its way down through the code. Numerous GOTO loops will cause journeyman programmers to snicker at you--if you really use too many, they refer to your work as "spaghetti code." (Back when I worked for a software company--I was primarily a management consultant, but tried to do a little coding now and again to be helpful--I got a real friendly invitation from the programming staff the next time I was visiting the home office. We all hiked a few blocks from the office to a trendy new nouveau-Italian place, where the Chief Programmer ordered an extra-large plate of spaghetti for me. The

restarant used paper tablecloths, so the programmers all pulled out crayons and diagrammed my latest programming effort in about a dozen colors...)

If poor programming in a structured environment produces spaghetti, poor programming in a Visual environment will produce lasagna. You can quickly prototype an application. But if you aren't careful, you'll kick yourself every time you go back into your code, as you deal with poorly named variables, inconsistencies between forms, etc.

The Databus programming language (a minicomputer language from long ago) requires you to declare all your variables at the beginning of the program. It's a good practice. In Visual Basic, where you have different code for different forms, I'd strongly suggest using the (general declarations) section to declare variables. I haven't done it--yet--but getting the variable and subroutine names straight in this app will be the next project.

Documenting a Project:

As I wrote above, a project that hasn't been documented is worthless the week after it has been written. You don't need to be quite as chatty as I've been here. But, if you're creating an application for a client, and you won't necessarily be there when the app is updated, you should be as exhaustive as you can be. A lot of "propeller-heads" don't like to write, so they find documenting code to be a real drag. But a lot of those programming wizards are also extraordinarily good teachers. If you just write out what you'd say to a novice programmer, you'll produce the best possible documentation your program could ever have.

You should produce a document much like this one: why the app was written, what it does, how it does it. You should also include lots of comments within the body of your code. The "thousand points of light" style of Visual Basic can make keeping track of programs pretty tough. If you're calling the same subroutines from different places, be sure to indicate in the subroutine code all the names of the calling routines. If you don't, you'll eventually cripple your application by "cleaning out" a chunk of code that you need to trap for a non-standard 9600 bpi modem, or somesuch.

Let me also give you a plug for providing a Windows Help file. You have to call Microsoft to get the Help compiler for VB, and it helps a bunch if you use Word for Windows. I haven't yet popped for the compiler (I want to really get some use from VB before I go spend another fifty bucks on another piece of software), but I will soon. Windows users expect Help files. A big part of making an environment like Windows work is playing by the rules, and that means providing Help files.

Pictures and Icons:

This application won't log you on to CompuServe. It will not sort a 10,000 record customer file. But it does do quite a bit with moving and

resizing controls, and it uses a lot of pictures.

Moving, Sizing, and Resizing:

If you've used computers for more than a year, chances are that you've used more than one computer. I used to work for a publishing company that had three PC-XT clones, an AT clone, a genuine IBM PC AT (with a 6 Mhz clock speed--golly!), and a Macintosh Plus. All in all, we had five different keyboards and six different monitors. So what? So assume that your application will be run on a different size monitor that what you're using.

When you write your application, you will have to keep track of three different kinds of measurements (I'll just talk about height--the same will be true for Width as well): Screen.Height, Height, and ScaleHeight.

Screen.Height is obvious. It's the height of the screen. But the trick is that you never know what that actual screen height will be. So you will always have to define form sizes in terms of percentages of the Screen.Height. Many applications simply center the form on the monitor:

```
Top = (Screen.Height - Height) / 2
Left = (Screen.Width - Width) / 2
```

Include this in the Form_Load event and the form will be centered. If you want to get a little fancier, move the form a little to one side or another:

```
Top = (Screen.Height - Height) * .4
Left = (Screen.Width - Width) * .6
```

That will make the form set a little higher on the monitor, and a little closer to the right-hand side.

If you want to move an object, you'll have to define it's Left and Top positions (X and Y). You can define X and Y in a variety of ways. In Ruthie I defined them by random movements (see Game 1 and 2). You could also define X and Y with the CurrentX and CurrentY positions returned by a MouseUp event (where something was dropped). One thing I'm trying is adding a little animation to the process. I'm fooling around with a little "counter motion" when I move a control: If I'm moving it two inches to the left, I'll first move it 2/10ths of an inch to the right, and then zip to the left. I'm still working on getting the effect to work reliably with a random-motion game, but it's really slick when it works right.

How about moving controls? If you let the user drag a control (a picture box, for instance) all that shows when he drags is the outline of the control. Yuck. That's why Game 3 uses icons

instead of metafile pictures. If I used metafiles I wouldn't be able to use the nifty little DragIcons. Keith Funk spent a lot of time trying to move picture controls manually, redrawing the object when the MouseMove event took place (see page 200 of the Programmer's Reference). He wasn't satisfied with the results, and I didn't see how I could improve matters. I gave up on picture controls and used icons instead.

BMPs vs. WMFs:
Don't use BMPs in your applications. The little picture of Ruthie Shipps on the "About Ruthie" screen requires 93K of RAM. That's a little less than a third of the entire .EXE file. That 93K picture began life as a 4K TIFF file, and it was a lot nicer image to boot....

You'll see the same problem with any type of bitmap file. Visual Basic won't support EPS files, but it will support Windows Metafiles. I drew the WMFs in Ruthie with Corel Draw, and imported them. It's quite simple, and they don't take up much in memory.

Randomize and Random Series:
As I mentioned above, Visual Basic gives you random numbers in a sequence. You'll continue to get the same sequence every time you run the routine where you ask for random numbers--VB uses the same number to "seed" the random number generator each time.

This can be a good thing in some circumstances. In my circumstances I only needed the appearance of random motion--reseeding the random number generator every time I loaded or resized the form was fine. But if you're doing something that requires a truly random number each time, remember to use the Randomize instruction before you ask for a Rnd.

Scale and Dimensions:
The manual tells you that the ScaleHeight depends on the Scale of the "container" that holds a control. It then describes a few ways to set the Scale, but doesn't give you any sense of how ScaleHeight and Height relate. First off, remember that you can use a variety of different measurement scales in an application. I work in publishing, so I'm quite content to work with points. A twip is 1/20th of a point, so twips are nice as well. However, you already know my thoughts on the value of custom scales. In addition I might want to use inches or centimeters, especially if I'm going to be printing out a report. Suppose you're the computer--we tell you that the Height of the form is 3, but the height of the command button is 425. Which is bigger? The form I'm thinking of is 3" high, with a thin little command button on it that is a mere 3/10" high. There has to be a way to relate one control to another, even if they're using different scales. Enter ScaleHeight. Every control is contained by something else. In Game 3,

for instance, the six pictures in the Target() array are contained in another picture box named Targets; That in turn is contained on the Game3.Frm form. If you want to relate a control and it's container, you always think in terms of the Height of the control, and the ScaleHeight of the container. If you're resizing Picture1 in Game 2, for instance, you would tell the computer

```
Picture1.Height = LiveArea.ScaleHeight * .15
Picture1.Width = LiveArea.ScaleWidth * .15
```

LiveArea is the container, Picture1 is the control being resized. If you then wanted to resize LiveArea, you'd treat it as the control being monkeyed with, and Game2.Frm as the container:

```
LiveArea.Height = Game2.ScaleHeight * .75
LiveArea.Width = Game2.ScaleWidth * .75
```

User Interface Considerations:

The first version of this game was polished up, copied onto a floppy disk, and sent off to my mother. She called up to report that it "didn't work right." Why? Mother has a CompuAdd HiRez 14" monitor, with 800 by 600 resolution. I designed the application to look splendid on a NEC 4D, a 16" monitor with 1024 by 768 resolution. All the measurements were different on her monitor.

If you design screens in any character-based environment, you're pretty much dealing with an 80-column, 25-row format. If one user uses 8514/A and another CGA, so what? It's all text. But with Windows and graphical programming, it's a little bit more complicated.

The most important thing I learned was to assume the user has a different monitor than I do. That pretty quickly leads you into using custom scales, and placing and sizing your controls at run time.

That's a drag.

No it's not. Because there's one big, glaring, stupendous flaw in Visual Basic: **There's no way to document control properties**. Sooner or later somebody will come up with an add-on that will cycle through all the properties in your application (using the TabOrder, for instance) and printing all the properties. Till then, (and even then) placing and sizing the controls at runtime makes it much simpler to document how and where the controls are used.

Improvements I'd Like to Make:

Right off the bat, I want to add sound. Instead of showing fireworks in the success picture in Game 3, I want to make fireworks sounds. Children, especially non-verbal children, react to sound. (Some would argue that games with sound--Nintendo, for instance--help many children become

non-verbal.) I'd like to play simple jingles with successes, a little "uh oh" tune when a Sad Face appears, things like that.

I'd also like to incorporate some animation features when I move controls. I'd like to give a little "counter motion" before I moved a picture, or perhaps have the picture "bounce" off the borders of the frame like a pool ball before coming to rest in its final location. Steve Gibson, in his InfoWorld column on Visual Basic, wrote an app that made the control bounce off the far side of a form before it settled into place. I'd like to explore that.

I also want to add a couple of other games. I'd like to add a second "level" to Game 3, where after a certain number of successes the Target() pictures move randomly about the LiveArea. (I figure I'll just have them move randomly along their X axis--since they'll still be shuffled it will present the appearance of completely random movement.) Then I'd like to have the child progress to a game where a series of blank icons are displayed. Somehow or other one of the pictures will move--and the child has to click on that picture. If the picture is stopped, one event happens. If she catches it in motion, then something else happens. I'd also like to have the child learn to compare different forms of the same letter. A lower case "a" might match a capital "A". I'd use the MouseMove event to draw a line behind the mouse to connect one form with another. It would be, in effect, an electronic version of the matching games that elementary workbooks use.

In Conclusion

If you've stayed with me to the end, I appreciate your patience. I've learned a lot in the past few weeks with Visual Basic. It's a fascinating tool. For many of us, it will be the difference between success and failure at programming.

I have a very low threshold of "good enough." If I can't produce an illustration or a program that does what I want pretty quickly, I'm inclined to go do something else. I found that when I bought Corel Draw my productivity went way up--I could achieve the drawing I wanted within my "good enough" threshold, so all of a sudden I was producing substantially better illustrations.

I've had the same experience with Visual Basic. I've fooled around with C. I've thought about getting a more modern C compiler and buying a book/disk package I saw advertised--a poor man's SDK. But I couldn't see anything coming of it--my prior experiences with C (and Fortran, and APL) proved to me that I didn't have the patience or the tenacity to learn the language well enough to achieve much more than "Hello, World!" As I reminded myself, if all I'm going to accomplish is "Hello, World!", why do I need more than GW-BASIC?

Visual Basic isn't going to make me a full-time programmer. But it will let me write fun little games like Ruthie!, it will let me participate productively in projects like the VBCT effort, and it will let me develop useful little programs that we'll use here at my company. I'm fascinated at what I can do with DDE links among VB applications, Excel, and Word for Windows. It's so fascinating that I'm having trouble paying attention to work these days.

Enough already. I hope you've enjoyed Ruthie. If you have any questions or comments, please contact me on CompuServe at 71507,1212. If you have suggestions for more (or better) games for Ruthie, please send them. I'd love to do more with it.

Thanks for your time!