

Learning Assembler with Delphi

author **Ian Hodger**

ianhodger@compuserve.com

copyright 1997

All rights reserved. No part of this document may be reproduced without prior permission of the author. Where the author has given a copy of this document, the recipient may reproduce this document for back up purposes only.

Whilst all care has been taken to test every line of code contained within this document, the reader is warned that all information herewithin is provided on an 'as is' basis. No Liability will be excepted for any errors found within the document nor for any consequences thereof.

Delphi, Object Pascal, and Turbo Assembler are trademarks of Borland International Inc. Windows, Windows 95 and DirectX are trademarks of Microsoft Corporation.

Abstract

The purpose of this article is to fill some of the gaps left by the original documentation provided with Borland's Delphi 2.0 Developer, however, at the time of writing, all of the code and concepts discussed herewithin are fully applicable to all variants of Delphi version 2.0 and later. The Principle area of discussion will be the use of assembler within Object Pascal. Nevertheless, other aspects of programming will be covered in passing, as required for the examples given. For simplicity, only a subset of Intel's instructions will be introduced, thus enabling a more generalized approach with few special cases. Furthermore, all the examples of assembler given will be contained in Pascal wrappers with portability in mind. Issues such as File access in assembler will not be covered, as this is more reliably achieved using standard Object Pascal. Indeed the document will seek to emphasize that one should only use assembler when necessary. In general, the document will take the following form. First an idea will be introduced, followed immediately by a relevant example, concluding with an explanation of greater depth. This format is abandoned where clarity demands.

Using Assembler in Borland's Delphi 2.0

Before we start, I should like to state the level of knowledge which I shall assume of the reader. Writing Object Pascal should be second nature. One should be familiar with Delphi's built in debugging facilities. Finally a general understanding of what is meant by terms such as instantiation, null pointer and memory allocation, is a must. If any of the above encourages feelings of doubt, please tread very carefully. Furthermore, only 32bit code will be discussed, and as such Delphi 2.0 is a necessity.

Why use Assembler ?

I have always found Object Pascal to produce fast and efficient code, add to this the Rapid Development Environment of Delphi, and the need to use assembler becomes questionable. In all of my work with Delphi, I have come across just two situations where I have felt one should consider the use of low level code.

- (1) Processing large quantities of data. Nb. I exclude from this any situation where a data query language is employed. For reasons of compatibility one should not tinker.
- (2) High speed display routines. Nb. I refer here to quick easy routines that sit well with pascal, not the esoteric C++ headers, external function libraries and hardware demands of DirectX.

I hope to introduce an example or two by the end of this article which meet the above criteria, and in doing so demonstrate not only how and when to use assembler, but also the seamless manner in which Delphi incorporates this code.

What is Assembler ?

I will assume that you have a rough idea of how a cpu goes about it's business. Basically we have a fancy calculator with a big memory. The memory is no more than an ordered sequence of binary digits arranged in blocks of eight bits, each forming a **byte**. Thus each byte can store an integer in the range of 0 to 255, and each byte's position in the sequence of memory gives it an unique **address** by which the cpu may change or recover it's value. The cpu also has a number of **registers** (you may like to think of these as global variables) with which to play. For example **eax,ebx,ecx** and **edx** are the general 32bit registers and throughout this article, these are the only ones we shall use. This means the largest number we can store in register **eax** is 2 to the power 32 minus 1, or 4294967295, for those of us brought up on 8bit machines, this is sheer luxury.

The cpu has the ability to manipulate the values of the registers, so to add 10 to the value of **eax**, one would issue the hexadecimal operation code

05/0a/00/00/00

this is machine code, there being a specific 'number' for each function the cpu can implement. To say that writing machine code is tedious would be an understatement, and as for debugging ! Assembler Language is just an easy way of remembering what machine code operations are available. The job of converting to machine code is done by an Assembler. Borland's Turbo Assembler is built in to Delphi, and is rather more pleasant to use.

If we look again at adding 10 to **eax**, the appropriate assembler instruction is

add eax,10 {a := a + 10}

Similarly, to subtract the value of **ebx** from **eax**

sub eax,ebx {a := a - b }

To save a value for a rainy day, we can move it to another register

mov eax,ecx {a := c }

or even better, save the value to a memory address

mov [1536],eax {store the value of **eax** at address 1536}

and of course to retrieve it

mov eax,[1536]

A quick aside on memory addresses, please bear in mind the size of the values you are moving about. The **mov [1536],eax** instruction affects not only memory address **1536**, but **1537,1538** and **1539** as well, because as you will recall **eax** is 32bits long, or rather 4 bytes. Memory is always addressed in bytes.

What does the Compiler do with all those variables ?

In every program you will have written, the compiler will have had to cope with a number of variables. Consider the pascal line

Count := 0;

To the compiler, this is just a value it has to remember. Consequently it sets aside a memory address to store this value, and to make sure it doesn't get confused later, calls this memory address 'Count'. This means that the code generated by the compiler for this line is something like this

mov eax,0

mov Count,eax

The compiler can't use a line such as

mov Count,0

because at least one parameter of the instruction must be a register.

If we were to consider the line

Count := Count + 1;

we would get something like this

```

mov eax,Count
add eax,1
mov Count,eax

```

For variables of types other than integer, matters can become more complex. So more of that later, and let's use what we've learnt so far.

Our first snippet of assembler.

Forgive the trivial nature of this example, but we've got to start somewhere. Consider the following lines of pascal code.

```

function Sum(X,Y:integer):integer;
begin
    Result := X+Y;
end;

```

Object Pascal provides the `asm .. end` block as a method of introducing assembler to our code. So we could rewrite the function Sum thus

```

function Sum(X,Y:integer):integer;
begin
    asm
        mov eax,X
        add eax,Y
        mov Result,eax
    end;
end;

```

This works, but there is little point. There is no speed gain and we've lost the readability of our code. There are, however, instances where our currently limited knowledge of assembler can be put to good use. Let's say we wish to convert explicit Red, Green, and Blue values into a colour of type TColor suitable for use in Delphi. The TColor type describes a 24bit True Colour stored in the format of an integer ie. four bytes, the highest of which is zero, thereafter in the order Red, Green, Blue.

```

function GetColour(Red,Green,Blue:integer):TColor;
begin
    asm
        mov ecx,0      {ecx will hold the value of TColor}
        mov eax,Red    {start with the Red component}
        and eax,255    {make sure Red is in range 0<=Red<=255}
        shl eax,16     {shift the Red value to the correct position}
        xor ecx,eax    {adjust value of TColor}
        mov eax,Green  {same again with Green component}
        and eax,255
        shl eax,8
        xor ecx,eax
        mov eax,Blue   {and again with Blue}
        and eax,255
        xox ecx,eax
        mov Result, ecx
    end;
end;

```

You'll have noticed that I've introduced a few binary operations. These operations are straightforward and are also defined in Object Pascal itself, but for clarity let's explicitly define all the operations introduced thus far and a few more.

Basic Assembler Operations

Lets first define a few symbols for the purpose of this document.

reg1, reg2... refer to any of the general 32bit registers **eax**, **ebx**, **ecx** and **edx**.

mem1, mem2... refers to any memory expression either explicit **[1536]**, implicit **[edx]** or pascal pointer **Count**. The brackets refer to the contents of the value, so **[edx]** is the memory location at address **edx**. The compiler knows a pascal pointer is a memory expression so brackets aren't required.

exp1, exp2... explicit values such as **10** in decimal, **0Ah** in hex.

dest refers to the parameter into which the result of the operation will be stored.

src refers to the source of any extra data required by the operation.

For each operation at least one of **dest** and **src** must be a register or an implicit memory address, ie a memory address dependant on the value of a specified register.

And now to the operations

add **dest,src**
 reg1,reg2
 reg1,mem1
 reg1,exp1
 mem1,reg1

performs integer addition on dest and src, storing the resultling value in dest.

and **dest,src**
 reg1,reg2
 reg1,mem1
 reg1,exp1
 mem1,reg1

performs the logical binary operation 'and' on dest and src, storing result in dest.

dec **dest**
 reg1
 mem1

subtracts one from the value of dest.

inc **dest**
 reg1
 mem1

adds one to the value of dest.

mov **dest,src**
 reg1,reg2
 reg1,mem1
 reg1,exp1

mem1,reg1

copy the value of src and store in dest.

**not dest
 reg1
 mem1**

inverts the binary value of dest and stores the result in dest.

**or dest,src
 reg1,reg2
 reg1,mem1
 reg1,exp1
 mem1,reg1**

performs an inclusive 'or' operation on dest and src storing result in dest

**shl dest,src
 reg1,exp1**

shifts the binary digits of dest, src places to the left and stores the result in dest. This is a quick way to multiply by a power of two.

**shr dest,src
 reg1,exp1**

shifts dest, src places to the right. Quick division by a power of two.

**sub dest,src
 reg1,reg2
 reg1,mem1
 reg1,exp1
 mem1,reg1**

subtracts the value of src from dest, storing the result in dest.

**xor dest,src
 reg1,reg2
 reg1,mem1
 reg1,exp1
 mem1,reg1**

performs an exclusive 'or' operation on dest and src storing the result in dest.

Special cases, multiply and divide

The operations for multiplication and division operate on the 64bit value formed by combining the values of **eax** and **edx** where the latter is the most significant value. For example if **eax** = **AAAAAAAh** and **edx** = **BBBBBBBh** then the 64bit value would be **BBBBBBBAAAAAAAh**.

**div src
 reg1
 mem1**

performs an unsigned divide on the 64bit value discussed above, the divisor being src. The quotient being stored in **eax**, the remainder in **edx**, and src is unaffected. If quotient is too large to be stored in **eax** or the value of src is zero, the cpu will generate an error and stop program execution.

**idiv src
 reg1
 mem1**

same as **div** but performs a signed divide.

```
mul    src
       reg1
       mem1
```

performs an unsigned multiplication of **eax** and **src**. The resulting 64bit value is stored in **eax** and **edx** as above.

```
imul   src
       reg1
       mem1
```

same as **mul** but signed.

Labels and Jumps.

This section is really no more than a precursor to the section on flow control. A label is just an address in memory, and a jump just tells the cpu to look to a different address for the next line to execute. This is best illustrated by example

```
...
jmp   @label1      { jump to label1 }
...                               { these }
...                               { next }
@label2:                               { instructions }
...                               { are }
jmp   @label2      { never }
...                               { used }
@label1:
...                               { now we start execution again }
```

Note that the declaration of a delphi label starts with '@' and ends with ':'. The reference of a label does not include the ':'.

Controlling the flow.

If we are going to produce really useful code, at some point we shall need to implement conditional statements such as **while..do** and **repeat..until**. Just one instruction combines the conditional possibilities available

```
cmp    dest,src
       reg1,reg2
       reg1,mem1
       reg1,exp1
```

and sets the result flags accordingly. What the flags are, and how they are set, will not be covered here. All you need know is the result of the following combinations of instructions.

```
cmp    dest,src
je    @label        if dest = src then jump to @label.
```

```
cmp    dest,src
jle   @label        if dest <= src then jump to @label.
```

```
cmp    dest,src
jne   @label        if dest <> src then jump to @label.
```

```
cmp    dest,src
```

jg	@label	if dest > src then jump to @label.
cmp	dest,src	
jge	@label	if dest => src then jump to @label.
cmp	dest,src	
jl	@label	if dest < src then jump to @label.
cmp	dest,src	
jz	@label	if dest = src then jump to @label.
cmp	dest,src	
jnz	@label	if dest <> src then jump to @label.

Whilst the jump operations are equivalent to a goto statement, the **call** and its associated **ret** operation form the basis of procedural programming. It is common to see assembler routines of the following form

```

...
    jmp @mainroutinestart
@localprocedure:    {start of local procedure}
    {some code}
    ret             {return from procedure}
@mainroutinestart:
    {some code}
    call @localprocedure
    {some code}
...

```

Furthermore, conditional calls can be constructed thus

```

...
    cmp    eax,12
    je     @skipcall    {if eax = 12 jump over call to procedure}
    call   @localproc   {if eax <> 12 call procedure}
@skipcall:
...

```

Using a combination of compare's, conditional jump's and procedural call's all of the elements of structured programming maybe implemented.

A proper example.

Until now we have achieved little substantial. However, all of the basic instructions have been introduced, so let's use them.

Suppose we have to display the output of some function dependant upon two variables. You might imagine this as a three-dimensional map, where the coordinates [x,y] correspond to a height h. When we plot the point [x,y] on the screen we need to give the impression of depth. This can be achieved by using colours of differing intensity, in our example, blue below sea level and green above. What is needed is a function that will convert a given height into the appropriate depth of color for a given sea level.

First we should plan the ranges of the variables involved. It is reasonable to use the integer type to store the height and sea level, given the range of a 32-bit value. The true colour range available, limits us to a maximum color depth for blue and green to 256, so we will have to scale

the results accordingly. If we assume a maximum height above, and depth below sea level to be 65536 feet, we can use **shl** and **shr** for some fast scaling, and we will get a change in colour depth every 256 feet. The function will of course, have to return a **TColor** type to be compatible with Delphi.

```
function ColorMap(Height,Sea:integer):TColor;
begin
    asm
        mov eax,Height
        cmp eax,Sea
        jg @Above          { jump to section dealing with above sea level }

                                { Height is beneath Sea Level }
        mov ecx,Sea
        sub ecx,eax        { ecx is depth beneath sea }
        shr ecx,8          { divide depth by 256 }
        cmp ecx, 256
        jl @NotMaxDepth    { ecx should not exceed 255 }
        mov ecx,255
        @NotMaxDepth:
                                { ecx now holds color }
        jmp @ColorDone

        @Above:
        sub eax,Sea        { eax is height above sea level }
        shr eax,8          { divide height by 256 }
        cmp eax,256
        jl @NotMaxHeight   { eax should not exceed 255 }
        mov eax,255
        @NotMaxHeight:
                                { eax now holds green color depth }
        shl eax,8          { eax now holds color }
        mov ecx,eax        { ecx now holds color for compatibility with
                                beneath sea level routine}
        @ColorDone:
        mov Result,ecx
    end;
end;
```

As it happens, the above routine can be written with Delphi's **assembler** directive. This method of writing assembler removes a lot of the protection provided by the compiler, but as compensation, speed improves.

Here is the above routine using the assembler directive.

```
function ColorMap(Height,Sea:integer):TColor;assembler;
asm
    cmp eax,edx
    jg @Above
    sub edx,eax
    shr edx,8
    cmp edx,256
    jl @NotMaxDepth
    mov edx,255
```

```

@NotMaxDepth:
    mov eax,edx
    jmp @ColorDone
@Above:
    sub eax,edx
    shr eax,8
    cmp eax,256
    jl @NotMaxHeight
    mov eax,255
@NotMaxHeight:
    shl eax,8
@ColorDone:
end;

```

At first sight it is a little difficult to see what's going on. The registers are set with certain values before entering the function or procedure. How these are set depends on how the function or procedure was defined. There are two possibilities.

Stand alone, or explicitly defined procedures and functions

On entry,

eax holds the value of the first parameter of the function or procedure if such exists.

ebx holds the address of the data block of the function or procedure. You must be careful when using ebx, for it must hold its initial value whenever you refer to a function or procedure's parameters or data in your assembler code block. Furthermore ebx must hold its initial value when exiting. The Delphi manual actually says don't touch.

ecx holds the value of the third parameter.

edx holds the second parameter value.

On exit,

eax holds the result of the function, or in the case of a procedure, convention states it holds the value of any relevant error code you may define.

ebx must hold its initial value. Failure to ensure this will cause a system crash.

Object method, procedures and functions

On entry,

eax holds the address of the parent object's data block. You don't need to maintain this value, however it is needed whenever you wish to access or change the values of the parent object's fields.

ebx is the same as above.

ecx holds the second parameter value.

edx holds the value of the first parameter.

On exit, the register values are as for a stand alone procedure or function.

With the above information, you should now be able to work your way through the ColorMap function. On the face of it, we seem to have reduced the number of lines of assembler from 18 to 15, which is not much of a saving, and the code is not as readable. However this is not the whole story. The compiler generates a fail-safe entry and exit code block for any function or procedure defined with the usual **begin..end** block. By using the **assembler** directive, the compiler employs only minimal entry and exit code. In the case of the ColorMap function this means it's code size has roughly halved, as has its execution time. These are the levels of performance gain that make writing assembler worthwhile.

Implementing local variables

It is clear, that with just four registers, implementing any half serious algorithm will not be a trivial matter. If a routine is fast, this is usually because every constant used has been calculated before entering any loops. In Object Pascal, this is where local variables are used. For a

procedure or function defined with the assembler directive, the same declaration format is used. For example, the following is valid,

```
function Test:integer:assembler;  
var    first,second,third:integer;  
asm  
        {some code, remembering that ebx must be it's initial value}  
end;
```

local constants, and constant variables may also be defined just as one would in Object Pascal. The compiler just reserves a block of memory, whose address is stored in ebx on entry. Thereafter the local variable names refer to an offset value from the base address of the data block. This allows the compiler to use the index addressing provided by the cpu. An index address is of the form **[reg1+reg2]** or **[reg1+exp1]** for example **[ebx+edx]**. You will find indexing the easiest way of addressing data such as strings and arrays but more of this later. In the case of a function definition, a reference to a local variable is implemented as an indexed address irrespective of the operation, consequently the operation **mov eax,second** is actually **mov eax,[ebx+4]** where 4 is the offset to the address of the value of the second variable. You could write either, but the former offers greater clarity. I hope you are now starting to appreciate the importance of maintaining the value of ebx.

There is also a quicker method of temporary value storage, and this brings us to the stack.

The Stack

The stack is just what it says. Think of a stack of books on a table, we can place another book on the stack or take a book off the stack. To get at the third book down, we must first remove the top two. This is what we do with register values, and there are two appropriate instructions.

push reg1 place the value of reg1 on top of the stack.

pop reg1 remove the top value of the stack, and place this in reg1.

Windows uses the stack to pass parameters to the api functions, and as such great care must be exercised when using the stack. Each **push** must have an associated **pop** in all the code you write, get this wrong and once again the system will crash.

We will use the stack, indeed it is necessary when implementing recursive functions, but generally we shall only use the stack for temporary storage. For example when using the **mul** and **div** operations there are register value changes all over the place,

```
...  
push edx      {save edx}  
mul ecx       {eax := eax*ecx, edx holds any overflow}  
pop edx       {dump any overflow value and restore edx}  
...
```

An aside on recursion. Avoid it at all costs. Recursive functions may look elegant and appear to use very little code to achieve a great deal, but they are the signature of the inexperienced. A recursive function will commonly overload the stack because at design time you cannot be certain how many recursions are required and as such the memory demands are unknown. How do you explain to an end-user that your 100K program requires 8mb to run, when an equivalent iterative routine may produce a 500K program needing just 1mb. Furthermore, discovering the cause of a stack failure is problematic, making debugging a painful process. The only time I can think of, where recursion is justified, is in the implementation of artificial intelligence algorithms. It might also be noted that generally, I have found iteration to be quicker.

Beyond integers

Until now we have discussed only 32bit integer values. This would seem to impose a limitation on any code we may wish to write, but this is not so. When Object Pascal passes a string as a parameter, for example, it is a 32bit integer value that is used. In this particular case the value passed is the memory address at which the contents of the string are stored. This value is what is referred to as a **pointer** and is the approach used for any data type, whether string, array or user-defined object. Indeed, even floating point values are referred to in this way, as they are considered to be strings.

What's in a pointer

In a normal pascal procedural definition, the values of the parameters are copied to the data area of the procedure. If the value is an integer the value is passed explicitly whenever the parameter is referred to. In the case of other data structures, the pointer passed, points to the appropriate part of the data area. In neither scenario, is the value of any variable passed as a parameter changed.

Object Pascal allows one to define parameters with the **var** directive, whence pointers to the actual variables are passed. Consequently, changes to actual variable values may be made. The same result is achieved by defining parameters to be of pointer type, each pointer pointing to the appropriate data structure. This latter approach has the advantage of coping with dynamically instantiated data, the price to pay being that care with null pointers and memory allocation should be taken.

A real project - Matrices

What follows is the definition of a new structure TMatrix. Together with explanations of the design decisions taken, you should find it a useful reference.

A 2x2 matrix has been chosen because it is adequate to illustrate the ideas involved, and can be easily expanded to a 4x4 which is far more useful, especially in graphics where homogeneous coordinates are used.

First we must decide what this class is to be used for. 2x2 matrices are most commonly employed in describing basic geometric transformations in a 2-dimensional vector space. For simplicity we shall assume the use of a cartesian coordinate system. Given this, two data structure types are required, one for a vector or point, the other for the matrix. It will also be assumed that dynamic allocation of both vectors and matrices will be required.

Now we have some ground rules, let's consider just one more design point. To implement a vector space with only integer values, transformations will be limited to a subset of translations, reflections and skews. Using real numbers is the ideal, but quite honestly it's painful and there is a speed penalty. What we must realize, is that in a digital computer a 'real' number is really an approximation made using a quotient. Having taken this on board, and given that we are unlikely to use a plotting range in excess of -32767 to +32767 (does your monitor have a resolution in excess of 1600x1280 ?) , we shall store each coordinate as 65536 times its actual value. This gives us a quotient with which we can approximate quite accurately the range of real values we shall use. Moreover we can implement this using the standard 32bit integer type, thus enhancing performance. Recovering the 'integer' part of a value is achieved by dividing it's value by 65536 which is easily done using **shr**, similarly setting a value requires multiplying by 65536 for which we use **shl**. If these ideas are new to you, you've just learnt one of the key approaches in code optimization.

Important note, when assigning and recovering values in Delphi, we must use multiplication and division, otherwise negative values get rather upset.

The type definitions are consequently straightforward.

type

{remember each stored value is 65536 times its actual value}

PVector = ^TVector;

{pointer type for dynamic instantiation}

TVector = record
X:integer;
Y:integer;
end;

{the vector is of the form X }
{ Y }

PMatrix = ^TMatrix;

TMatrix = record
A:integer;
B:integer;
C:integer;
D:integer;
end;

{the matrix is of the form A,B }
{ C,D }

{to avoid problems with memory allocation each of the following}
{functions will adjust the actual value of the first parameter, the }
{second parameter will be considered a source, as in assembler }
{and as such remains unaffected}

function VectorAdd(Vdest,Vsrc:PVector):boolean;

 {Vdest := Vdest + Vsrc}

function VectorSub(Vdest,Vsrc:PVector):boolean;

 {Vdest := Vdest - Vsrc}

function VectorScale(Vdest:PVector;K:integer):boolean;

 {Vdest := K * Vdest, Nb. K is 65536 times value again}

function VectorAssign(Vdest,Vsrc:PVector):boolean;

 {Vdest := Vsrc}

function MatrixMul(Mdest,Msrc:PMatrix):boolean;

 {Mdest := Msrc * Mdest, Nb. the order is important}

function MatrixIdent(Mdest:PMatrix):boolean;

 {set Mdest to the identity matrix}

```

function MatrixScale(Mdest:PMatrix;K:integer):boolean;

    {Mdest := K * Mdest, Nb K is 65536 times value again}

function MatrixAssign(Mdest,Msrc:PMatrix):boolean;

    {Mdest := Msrc}

function Transform(Vdest:PVector;Msrc:PMatrix):boolean;

    {Vdest := Msrc * Vdest}

```

Before going through the implementation of these functions, we have two more points to discuss. Firstly, error trapping. If any of these functions is sent a null or invalid pointer, a memory exception error will occur unless some form of checking is used. Checking for a null pointer is easy, but discovering whether a pointer is valid, this is nigh impossible. This aside, we should not lose sight of why we are using assembler. Speed is the essence. My own view is that for such routines as these, error checking is not required, as the cause of the error will be self evident in debugging the main delphi program. However, a simple null pointer check will be implemented in the first function as a simple example. Further error checking routines will be implemented in the next project, where identifying the causes of errors will not be so easy.

Secondly, as the reason for using assembler is speed, we should consider how this is best achieved. It is clear that given a generic NxN matrix, a generic algorithm capable of calculating the desired result for any element, nested in two loops, is the natural approach, and would produce a very memory efficient routine. If we were to explicitly write optimized code for each element of the matrix and tag these end to end we would avoid the time taken to execute any loop control code. Such an algorithm would only work for a given matrix size, and with even moderately sized matrices, would consume a far greater chunk memory. It would, however, be quicker. Given these two approaches, we must decide which route to take, and clearly with a 2x2 matrix the latter is preferable. It is also worth considering just how little memory is used by a line of assembler, typically 8 or 12 bytes, how many lines of assembler you are going to write for a routine, more than 1000 ?, and then look at how much memory your machine has, 16Mb+ perhaps ? You will find that in general, using more memory produces faster routines, only experience will tell you where to draw the line.

The implementation of the Vector and Matrix functions.

```

function VectorAdd(Vdest,Vsrc:PVector):boolean;assembler;
asm
    {as this is a stand alone function
    eax holds the pointer Vdest and
    edx holds the pointer Vsrc
    remember ebx must be saved}

    {the error checking I promised}
    jmp @errorcheck
@seterrorcode:
    mov eax,0
    jmp @adddone
    {eax holds return value, 0 is equivalent to False}
@errorcheck:
    cmp eax,0
    je @seterrorcode
    cmp edx,0
    je @seterrorcode

```

```

{error checking done, now start routine}
{Nb. addition and subtraction are unaffected by our
multiplying the values by 65536}
mov ecx,[eax]
{eax holds pointer to first value of Vdest, so ecx = Vdest.X}
add ecx,[edx]
{ecx = Vdest.X + Vsrc.X}
mov [eax],ecx
{change Vdest.X}
add edx,4
add eax,4
{as the X element of a Vector record is an integer it consumes
four bytes of memory, thus by adding four to eax and edx, both
now point to the Y element of the Vector record}
mov ecx,[eax]
add ecx,[edx]
mov [eax],ecx
{update Y element of Vdest}
mov eax,1
{set return value to True}
@adddone:
end;

function VectorSub(Vdest,Vsrc:PVector):boolean;assembler;
asm
    {no error checking, speed gain 30+%}
    mov ecx,[eax]
    sub ecx,[edx]
    mov [eax],ecx
    add eax,4
    add edx,4
    mov ecx,[eax]
    sub ecx,[edx]
    mov [eax],ecx
end;

function VectorScale(Vdest:PVector;K:integer):boolean;assembler;
asm
    {I'll go through this step by step, and try not to loose you}
    mov ecx,edx
    {we're going to use the mul operation, which overwrites the
value of edx, since edx holds the value of K, we move the
value to ecx}
    push eax
    {mul also changes eax, and we need to store the pointer
Vdest}
    mov eax,[eax]
    {before we multiply let's look at what is stored where.
     $eax = 65536 * Vdest.X$ 
     $ecx = 65536 * K$ 
    and the pointer Vdest is on the stack}
    imul ecx
    {the 64bit result of the mul operation is
     $65536 * 65536 * Vdest.X * K$ 
    and is stored in registers edx and eax

```

```

do the arithmetic and you'll find
edx = K * Vdest.X
eax = the remainder of the quotient}
shl edx,16
{we actually want 65536 * K * Vdest.X
so multiply edx by 65536, the shl operation does this}
shr eax,16
{to maintain accuracy we must add the remainder of the
quotient to edx. As eax holds the 32bit remainder and we
only want a 16bit remainder for edx. We divide eax by
65536 using shr}
xor edx,eax
{we now combine edx and eax using the binary operation
xor, we could use add, but xor is generally quicker}
pop eax
{restore the value of eax, Vdest}
{At this point it's worth looking at the register values
eax = Vdest
ebx = unchanged
ecx = 65536 * K
edx = 65536 * K * Vdest.X
and the stack is clear}
mov [eax],edx
{update value of Vdest.X}
add eax,4
{move pointer from Vdest.X to Vdest.Y and do it all again}
push eax
mov eax,[eax]
imul ecx
shl edx,16
shr eax,16
xor edx,eax
pop eax
mov [eax],edx
end;

```

```

function VectorAssign(Vdest,Vsrc:PVector):boolean;assembler;
asm
    {by now you should be able to follow this}
    mov ecx,[edx]
    mov [eax],ecx
    add edx,4
    add eax,4
    mov ecx,[edx]
    mov [eax],ecx
end;

```

```

function MatrixMul(Mdest,Msrc:PMatrix):boolean;assembler;
var nA,nB,nC,nD,dest,src:integer;
{there are too many intermediary values to keep track of, which makes
using the stack confusing, so we'll just define some local values for the
new element values of Mdest and the pointer values of Mdest and Msrc}
asm
    mov dest,eax

```

mov src,ecx

{save dest and src}

mov eax,[eax].TMatrix.A

mov ecx,[ecx].TMatrix.A

{a little earlier I mentioned index addressing, these two lines ask the compiler to calculate the offset required to address the 'A' field of a TMatrix record and use the appropriate index address. For example

$[eax].TMatrix.B = [eax].TVector.Y = [eax+4]$

or rather the offset to both 'B' and 'Y' is 4 bytes

Otherwise nothing else new in this function because I am going to assume you know how to multiply two matrices}

imul ecx

shl edx,16

shr eax,16

xor edx,eax

mov nA,edx

mov eax,dest

mov ecx,src

mov eax,[eax].TMatrix.C

mov ecx,[ecx].TMatrix.B

imul ecx

shl edx,16

shr eax,16

xor edx,eax

add edx,nA

mov nA,edx

{new A calculated now start on new B}

mov eax,dest

mov ecx,src

mov eax,[eax].TMatrix.B

mov ecx,[ecx].TMatrix.A

imul ecx

shl edx,16

shr eax,16

xor edx,eax

mov nB,edx

mov eax,dest

mov ecx,src

mov eax,[eax].TMatrix.D

mov ecx,[ecx].TMatrix.B

imul ecx

shl edx,16

shr eax,16

xor edx,eax

add edx,nB

mov nB,edx

{new C}

mov eax,dest

mov ecx,src

mov eax,[eax]

mov ecx,[ecx].TMatrix.C

imul ecx

shl edx,16

```

    shr eax,16
    xor edx,eax
    mov nC,edx
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TMatrix.C
    mov ecx,[ecx].TMatrix.D
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    add edx,nC
    mov nC,edx
    {and finally D}
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TMatrix.B
    mov ecx,[ecx].TMatrix.C
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    mov nD,edx
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TMatrix.D
    mov ecx,[ecx].TMatrix.D
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    add edx,nD
    {finish by updating Mdest, Nb. as edx holds final value of D
    there is little point in updating the local variable nD}
    mov eax,dest
    mov [eax].TMatrix.D,edx
    mov edx,nA
    mov [eax].TMatrix.A,edx
    mov edx,nB
    mov [eax].TMatrix.B,edx
    mov edx,nC
    mov [eax].TMatrix.C,edx
end;

```

```

function MatrixIdent(Mdest:PMatrix):boolean;assembler;
asm

```

```

    {no comments needed}
    mov [eax].TMatrix.A,65536
    mov [eax].TMatrix.B,0
    mov [eax].TMatrix.C,0
    mov [eax].TMatrix.D,65536
end;

```

```

function MatrixScale(Mdest:PMatrix;K:integer):boolean;assembler;

```

asm

{same as VectorScale but twice as long}

```
mov ecx,edx
push eax
mov eax,[eax].TMatrix.A
imul ecx
shl edx,16
shr eax,16
xor edx,eax
pop eax
mov [eax].TMatrix.A,edx
push eax
mov eax,[eax].TMatrix.B
imul ecx
shl edx,16
shr eax,16
xor edx,eax
pop eax
mov [eax].TMatrix.B,edx
push eax
mov eax,[eax].TMatrix.C
imul ecx
shl edx,16
shr eax,16
xor edx,eax
pop eax
mov [eax].TMatrix.C,edx
push eax
mov eax,[eax].TMatrix.D
imul ecx
shl edx,16
shr eax,16
xor edx,eax
pop eax
mov [eax].TMatrix.D,edx
```

end;

function MatrixAssign(Mdest,Msrc:PMatrix):boolean;assembler;

asm

```
{no comments needed}
mov ecx,[edx].TMatrix.A
mov [eax].TMatrix.A,ecx
mov ecx,[edx].TMatrix.B
mov [eax].TMatrix.B,ecx
mov ecx,[edx].TMatrix.C
mov [eax].TMatrix.C,ecx
mov ecx,[edx].TMatrix.D
mov [eax].TMatrix.D,ecx
```

end;

function Transform(Vdest:PVector;Msrc:PMatrix):boolean;assembler;

var nX,nY,dest,src:integer;

asm

{MatrixMul cut in half really}

```

    mov dest,eax
    mov src,edx
    mov eax,[eax].TVector.X
    mov ecx,[edx].TMatrix.A
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    mov nX,edx
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TVector.Y
    mov ecx,[ecx].TMatrix.B
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    add edx,nX
    mov nX,edx
    {new X done}
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TVector.X
    mov ecx,[ecx].TMatrix.C
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    mov nY,edx
    mov eax,dest
    mov ecx,src
    mov eax,[eax].TVector.Y
    mov ecx,[ecx].TMatrix.D
    imul ecx
    shl edx,16
    shr eax,16
    xor edx,eax
    add edx,nY
    {new Y done update Vdest}
    mov eax,dest
    mov [eax].TVector.Y,edx
    mov edx,nX
    mov [eax].TVector.X,edx
end;

```

Using the Vector and Matrix functions is very straightforward with few points to remember. Firstly, the functions assume that a pointer is to be passed as the parameter, but you won't forget this as the compiler will remind you. This means, that to create vector and matrix variables, you must first define your variable with the appropriate PVector or PMatrix type in the required var section as in the example below.

```

procedure Example;
var aVector:PVector;aMatrix:PMatrix;
begin

```

```

new(aVector);
aVector^.X := 65536*32;
aVector^.Y := 65536*21;
new(aMatrix);
with aMatrix^ do
begin
    A := round(65536 * 0.5);
    B := round(65536 * cos(pi /3));
    C := 3 * 65536;
    D := 65536;
end;
Transform(aVector,aMatrix);
dispose(aVector);
dispose(aMatrix);
end;

```

Pascal's new() function is used to allocate memory for the new variable, this must be done otherwise the pointer will remain null with the consequences discussed earlier. Furthermore you must tidy up after yourself, that is to say, every new() must have an associated dispose() to free the memory you have used. Otherwise dynamically allocated variables are just like any other, except one must use the '^' to access the variable's fields.

It is likely you will use the PVector and PMatrix types with TList's as these deal explicitly with pointers and their manipulation.

You may be asking yourself why the Vector and Matrix were defined as a record rather than as an object as one would expect with Object Pascal. The reason is that each object carries with it the overhead of a virtual method table (see Delphi documentation) thus increasing the memory requirement for data storage. Records store just the data and when one considers that a half serious vector graphics application may deal with 500,000 vectors or more saving just one byte in each record soon makes a difference.

As an exercise you might like to write a function to find the inverse of a Matrix. For those of you with aspirations of writing a 3D graphics engine, you now have all the knowledge of assembler you will need to rewrite the vector and matrix definitions. All that is missing is an understanding of homogeneous equations, a good book on modern algebra will help here, and a method of fast screen access. I shall deal with the latter next.

Project 2 - fast screen plotting

Have you ever tried to write a bitmap painting program with that fancy brush in which you've worked out how to blend and fade colors to spectacular effect, only to be let down by the speed at which Delphi's canvas pixel array operates ? You have, well read on.

First, before we go any further please take time to fire up Delphi and make sure that 'aligned record fields' is switched on in the compiler section of project options. It is ? Good, virtually every api function we shall use in this section demands that all data is double word aligned. That last two minutes has just saved you hours of debugging time.

What makes Windows so good a programming platform, is that it acts as an interface between the programmer and the hardware. You don't need to worry about what kind of graphics or sound card is actually there you just ask windows to play a wav file or display a bitmap and it does it. No more sound card crashes because you chose the wrong interrupt.

There is of course a price, to pay. That price is as ever, speed. With DirectX there are now api routines for direct screen access, however its performance is highly dependant upon the hardware fitted.

Delphi's canvas is designed to hide the complexities in using the gdi to draw a square on the screen, leaving well alone the rather more esoteric nature of directX. This simplicity is once again at the price of speed.

I decided some time ago that I needed a method of constructing a bitmap in memory which I could then bitblt on to a canvas. I would be able to build the bitmap using assembler routines for

speed and bitblt'ing is quick. Indeed what I primarily needed at the time, was to be able to plot full 24bit color values on to a bitmap at rate of about one million pixels per second, this being for a graphical display routine.

I soon discovered that attempting to unravel the torturous way in which Delphi deals with bitmaps would prove unfruitful. As such the windows gdi functions would have to provide the answer. Windows 95 provides Device Independent Bitmaps, as it's way of allowing the direct memory addressing of bitmap pixels, together with several routines to service their use. The key function in this respect, is **CreateDIBSection** which not only creates the appropriate structure and handle, but also provides the address of the **DIB's** pixel array. Given the address and size of bitmap in memory, writing routines to manipulate the image becomes almost trivial. Functions such as **StretchDIBits** can then be used to map the resulting bitmap on to any canvas.

At this point you maybe feeling misled, after all this project is entitled 'fast screen plotting', and the method described does not seem direct. Look at the DIB that we are going to create, as a working area in which we shall construct an image before displaying it. Such an area of memory is often referred to as a frame buffer, and is commonly used where generating a new image from scratch is quicker than manipulating an existing image. Such is the case for real time display systems, 3D views for example. However this does not preclude us from using this approach to manipulate existing pictures.

Encapsulating this frame buffer in a class definition seems appropriate giving us the following basic declaration.

```
TFrame      = class
  private
    Bitmap : HBitmap;
    lpvbits : Pointer;
    Bitmapinfo : PBitmapinfo;
    {Bitmap will hold the windows handle of the DIB, lpvbits will be a pointer to the DIB's
    pixel array, and Bitmapinfo is a pointer to an instance of the Windows Bitmapinfo
    structure in which we shall describe the format of the DIB, ie. width, height and color
    depth}
    FWidth : integer;
    FHeight : integer;
    FSize : integer;
    FLineLen : integer;
    {just fields to store values of the DIB's width, height, size in bytes, and horizontal size
    in bytes}
  public
    constructor Create(ACanvas:TCanvas;Width,Height:integer);
    destructor Destroy;override;
    {the size of the DIB to be created will be determined by the width and height
    parameters, the canvas parameter determines the palette to be used but this is best
    explained in 'Create's' implementation. The destructor must free up any memory
    allocated to the DIB. The memory usage is likely to be considerable.}
    function Draw(ACanvas:TCanvas;X,Y:integer):integer;
    procedure Plot(X,Y,Color:integer);
    {Draw provides a method of painting the DIB on to a Canvas. X and Y giving the
    canvas coordinates of the top left corner of the DIB after painting. Plot allows one to
    set the color of a given pixel in the DIB.}
  end;
```

Before looking at the implementations we should consider how we are to define the DIB. I am not referring here to the windows api functions but rather the color depth of the DIB as this will dramatically affect our code. The windows routines are generic and can cope with any color depth from monochrome to true color. Consequently there is a speed penalty whilst format checking is carried out. By choosing a single color format, our routines can be optimized. In this

case I have chosen 24bit true color as this met my needs at the time of writing and avoided the need for palette definitions. However for ultimate speed, 8bit color using the system palette is the way to go.

If we now look at the implementation of each object method in turn.

```
constructor TFrame.Create(ACanvas:TCanvas;Width,Height:integer);
var LineAdj:integer;
begin
```

```
    FWidth := Width;
    FHeight := Height;
    FLineLen := 3*Width;
    LineAdj := FLineLen and 3;
    LineAdj := 4 - LineAdj;
    LineAdj := LineAdj and 3;
    FLineLen := FLineLen + LineAdj;
    FSize := FLineLen * FHeight;
```

{Storing the values of width and height in the appropriate fields is straightforward enough, the tricky bit is calculating the size of the DIB. Each horizontal scan line of the DIB must be double word aligned, that is to say, each scan line must start at an address which is a multiple of four bytes. Windows demands this is true and will fail to create the DIB if it is not. Why this demand is made is a matter of cpu architecture and optimizing performance. This is why I asked you to check that 'aligned record fields' is switched on in the compiler. To calculate the memory required to store one horizontal scan line we multiply the width by three and then work out how many bytes we must tag on the end to make this value divisible by four. Summing these values gives us FLineLen the number of bytes required to store a single horizontal line. The total memory used by the DIB being the product of FLineLen and the number of Horizontal lines FHeight.}

```
    New(Bitmapinfo);
    with Bitmapinfo^.bmiHeader do
    begin
        bsize := 40;                {size of the bmiHeader structure}
        biWidth := Width;
        biHeight := Height;
        biPlanes := 1;              {must always be one}
        biBitCount := 24;            {24bits required to store each pixel}
        biCompression := BI_RGB;    {image uncompressed, no palette}
        biSizeImage := FSize;        {size of image pixel array}
        biXPelsPerMeter := 0;        {info for scaling when printing etc.}
        biYPelsPerMeter := 0;
        biClrUsed := 0;              {number of colors in palatte}
        biClrImportant := 0;        {number of important colors in palette}
    end;
```

{The PBitmapinfo type is defined in Delphi's Graphics unit and encapsulates the Windows Bitmapinfo structure itself containing two record structures, bmiHeader and bmiColors. The latter defines a palette, but as we are using explicit 24bit true color values, a palette is not required. Consequently bmiColors remains null. The bmiHeader structure defines the size and color usage as above.}

```
    Bitmap := CreateDIBSection(ACanvas.Handle,Bitmapinfo^,
        DIB_RGB_COLORS,lpvbits,nil,0);
```

{If we look at the parameters in order, ACanvas.Handle is the handle of a valid device context and is used to define the logical palette of the DIB if the color usage is defined as DIB_PAL_COLORS, it isn't so the handle passed doesn't matter except it must be a valid device context. Bitmapinfo^ passes the size, format and color data in the required structure. DIB_RGB_COLORS defines the color usage, in this case explicit RGB values. lpvbits is a pointer whose value will be changed so that it points to the pixel array of the DIB. The last two

parameters tell windows how the memory required by the DIB is to be allocated, in this case the values tell windows to allocate the memory itself. It is possible to handle the memory allocation yourself, but why bother. The function returns a valid handle in Bitmap if successful.}

end;

The destructor is clearly defined as follows. We just get windows to reclaim the memory allocated to the DIB and we finish by disposing of the Bitmapinfo record.

```
destructor TFrame.Destroy;
begin
    DeleteObject(Bitmap);
    Dispose(Bitmapinfo);
end;
```

The Draw method just uses StretchDIBits. By changing the parameters various scaling and image manipulation effects can be achieved.

```
function TFrame.Draw(ACanvas:TCanvas;X,Y:integer):integer;
begin
    StretchDIBits(ACanvas.Handle,X,Y,FWidth,FHeight,0,0,FWidth,FHeight,
        Ipvbits,Bitmapinfo^,DIB_RGB_COLORS,SRCCOPY);
    Result := GetLastError;
end;
```

Now we can get back to assembler and define a routine to change the color of a pixel in the DIB.

```
procedure TFrame.Plot(X,Y,Color:integer);assembler;
asm
```

```
    push ebx
    mov ebx,[eax].Bitmap
    cmp ebx,0
    je @plotdone
    {if the value of Bitmap is zero then no memory has been allocated to the DIB. All we can
do is abort the plot.}
    pop ebx
    push ebx
    {recover value of ebx without affecting the stack}
    cmp edx,0
    jl @plotdone
    {if X coordinate is less then zero then abort}
    cmp edx,[eax].FWidth
    jge @plotdone
    {if X coordinate is greater then or equal to the DIB's width then abort}
    cmp ecx,0
    jl @plotdone
    cmp ecx,[eax].FHeight
    jge @plotdone
    {same checks on Y coordinate}
    {we need to calculate the memory offset of point X,Y in the DIB and then add the
memory address of the start of the DIB to find the actual address of the point. The offset is
FLineLen*Y+3*X}
    push eax
    push edx
```

```

    {eax = object base address, edx = X. since we are about to use the mul operation we
must save these values}
    mov eax,[eax].FLineLen
    {eax = FLineLen, ecx = Y, so we can now multiply}
    mul ecx
    {eax = FLineLen*Y, edx = 0}
    mov edx,eax
    {we need to recover the values of X and the object base address from the stack, so we
move the value of FLineLen*Y to edx before recovering eax's value}
    pop ecx
    pop eax
    {eax = object base address, edx = FLineLen*Y, ecx = X}
    add edx,ecx
    {edx = FLineLen*Y+X}
    shl ecx,1
    {ecx = 2*X}
    add edx,ecx
    {edx = FLineLen*Y+X+2*X = FLineLen*Y+3*X, which is what we want}
    add edx,[eax].lpvbits
    {add the memory address of the start of the DIB, and edx now holds the actual address
of the pixel X,Y}
    mov ecx,[edx]
    and ecx,0ff00000h
    {get the current value of the pixel, as we can only move four bytes around at a time and
the pixel color value is only three bytes long, the fourth and most significant byte is part of the
color value of the next pixel. Using the 'and' operation we isolate the value of this fourth byte and
store it in ecx}
    mov ebx,Color
    and ebx,0ffffffh
    {the value of ebx is currently on the stack, so this can be recovered in a moment. Having
loaded ebx with the color value to be 'plotted' we must ensure it is only three bytes long}
    xor ecx,ebx
    mov [edx],ecx
    {using 'xor' we combine the three byte color value in ebx with the fourth byte in ecx, and
in doing so avoid affecting the color of the next pixel. This combined value is then written over
the pixel address achieving the 'plot'}
@plotdone:
    pop ebx
    {before exiting we recover ebx's value}
end;

```

So, how do we use this frame buffer. The following example function illustrates the TFrame creation, the plotting of a basic fractal effect, the display of the resulting image on a form's canvas, and finally the destruction of the object.

```

function Example;
var   x,y:integer;Frame:TFrame;
begin
    Frame := TFrame.Create(Self.Canvas,400,300);
    {instantiate a TFrame object with a DIB 400 pixels wide 300 pixels high}
    for x := 0 to 399 do
        for y := 0 to 299 do
            Frame.Plot(x,y,x*y);
            {do some plotting, incidentally the coordinate (0,0) refers to the bottom left of the image,
not the top left as in a normal bitmap}

```

```

Frame.Draw(Self.Canvas,10,10);
    {now display the image, in this case the top left of the image will have coordinates 10,10
with respect to the Canvas chosen}
Frame.Free;
    {finally dispose of the memory allocated to the object. In a real application you are likely
to create a TFrame object at the start of processing, and only dispose of it just before exiting to
windows}
end;

```

The object as shown here is pretty much as I used it. The only additional function I have found useful has been GetPixel(X,Y) returning the color value of the DIB at (X,Y). This function is of course, just the Plot routine with a couple of changes. That I have not needed to draw geometric shapes, is probably a peculiarity of my own requirements. I suspect however, that your own circumstances may differ, so there now follow a few more routines for the TFrame object. Incidentally, as the DIB has a valid windows handle, the gdi functions can be used to draw basic shapes.

```

function TFrame.GetPixel(X,Y:integer):TColor;assembler;
asm

```

```

    push ebx
    mov ebx,[eax].Bitmap
    cmp ebx,0
    je @getpixeldone
    pop ebx
    push ebx
    cmp edx,0
    jl @getpixeldone
    cmp edx,[eax].FWidth
    jge @getpixeldone
    cmp ecx,0
    jl @getpixeldone
    cmp ecx,[eax].FHeight
    jge @getpixeldone
    {we need to calculate the memory offset of point X,Y in the DIB and then add the
memory address of the start of the DIB to find the actual address of the point. The offset is
FLineLen*Y+3*X this is the same as the Plot routine}
    push eax
    push edx
    mov eax,[eax].FLineLen
    mul ecx
    mov edx,eax
    pop ecx
    pop eax
    add edx,ecx
    shl ecx,1
    add edx,ecx
    add edx,[eax].lpvbits
    mov eax,[edx]
    and eax,0ffffffh
    {having got four bytes of data from the DIB, we dispose of the fourth, most significant
byte, leaving just the color value of point X,Y}
@getpixeldone:
    pop ebx
end;

```

The next routine fills the DIB with a given color.

```
procedure TFrame.Fill(FColor:integer);assembler;  
var X,Y,indexY,indexP,Color:integer;  
asm
```

```
    mov ecx,[eax].Bitmap  
    cmp ecx,0  
    je @filldone  
    {check DIB exists and exit if not}  
    mov ecx,[eax].FWidth  
    mov X,ecx  
    mov ecx,[eax].FHeight  
    mov Y,ecx  
    mov ecx,[eax].lpvbits  
    mov indexY,ecx  
    mov indexP,ecx  
    and edx,0ffffffh  
    mov Color,edx
```

{initialize variables X and Y act as counts, each horizontal line is considered in turn indexY holding the address of point (0,Y) for a given Y. There after each iteration adds three to this value storing the result in indexP, each successive value corresponding to the address of a point on the horizontal scan line. When the count reaches zero the line has been completed, and the next scan line is considered by adding FLineLen to indexY and resetting X and indexP. When Y equals zero the fill has been completed without resorting to multiplication}

```
@startfill:
```

```
    mov edx,indexP  
    mov ecx,[edx]  
    and ecx,0ff000000h  
    xor ecx,Color  
    mov [edx],ecx  
    add edx,3  
    mov indexP,edx  
    mov ecx,X  
    dec ecx  
    mov X,ecx  
    cmp ecx,0  
    jg @startfill  
    mov edx,indexY  
    add edx,[eax].FLineLen  
    mov indexY,edx  
    mov indexP,edx  
    mov ecx,[eax].FWidth  
    mov X,ecx  
    mov edx,Y  
    dec edx  
    mov Y,edx  
    cmp edx,0  
    jg @startfill
```

```
@filldone:
```

```
end;
```

A plot function utilizing our previously defined PVector pointer in place of explicit coordinates.

```

procedure TFrame.VPlot(V:PVector;Color:integer);assembler;
asm
    push ebx
    mov ebx,[eax].Bitmap
    cmp ebx,0
    je @vplotdone
    {if the value of Bitmap is zero then no memory has been allocated to the DIB. All we can
do is abort the plot.}
    pop ebx
    push ebx
    {recover value of ebx without affecting the stack}
    cmp edx,0
    je @vplotdone
    {if edx = V = 0 then the vector pointer passed is undefined, so exit}
    mov ecx,[edx].TVector.Y
    mov edx,[edx].TVector.X
    {now move the vector coordinate values into edx and ecx and the rest of the routine is
the same as Plot}
    cmp edx,0
    jl @vplotdone
    cmp edx,[eax].FWidth
    jge @vplotdone
    cmp ecx,0
    jl @vplotdone
    cmp ecx,[eax].FHeight
    jge @vplotdone
    push eax
    push edx
    mov eax,[eax].FLineLen
    mul ecx
    mov edx,eax
    pop ecx
    pop eax
    add edx,ecx
    shl ecx,1
    add edx,ecx
    add edx,[eax].lpvbits
    mov ecx,[edx]
    and ecx,0ff000000h
    mov ebx,Color
    and ebx,0ffffffh
    xor ecx,ebx
    mov [edx],ecx
@vplotdone:
    pop ebx
end;

```

Just for fun, here is a basic airbrush routine. It is not optimized, but does illustrate how I tend to test ideas, and uses many of the ideas discussed earlier. When used, the airbrush routine produces a circle of the desired color and radius, centred on X,Y, whose effect on the original image lessens as the perimeter of the circle is approached.

```

procedure TFrame.AirBrush(FX,FY,Radius,Color:integer);assembler;
var    X,Y,X0,Y0,X1,Y1,Xd,Yd,R2,D2,newColor:integer;

```

{the variables declared are all of the constant values which will be used

X,Y centre of airbrush plot

X0,Y0 bottom left coordinate of square to scan = X-Radius,Y-Radius

X1,Y1 top right coordinate of square to scan = X+Radius,Y+Radius

Xd,Yd current point being considered

R2 square of the Radius

D2 square of the distance of current point Xd,Yd from centre

newColor holds the color value for current point as it is being constructed}

asm

jmp @airstart

{define subroutines}

@airpointok:

{checks point Xd,Yd is valid, if valid edx = address, if not edx = 0}

push ecx

mov ecx,Yd

cmp ecx,0

jl @airpointerror

cmp ecx,[eax].FHeight

jge @airpointerror

push eax

mov eax,[eax].FLineLen

mul ecx

mov edx,eax

pop eax

mov ecx,Xd

cmp ecx,0

jl @airpointerror

cmp ecx,[eax].FWidth

jge @airpointerror

add edx,ecx

shl ecx,1

add edx,ecx

pop ecx

add edx,[eax].lpvbits

ret

@airpointerror:

pop ecx

mov edx,0

ret

@airblend:

{takes the intensity of R,G or B, 0 -> 255, ecx = current value, edx = new value and blends them according to current value of D2, the square of the distance from X,Y. returns value in ecx}

push eax

push edx

mov eax,D2

mul ecx

mov ecx,eax

pop edx

mov eax,R2

sub eax,D2

mul edx

add eax,ecx

xor edx,edx

```
mov ecx,R2
div ecx
mov ecx,eax
pop eax
ret
```

@airstart:

```
{initialize all variables}
mov X,edx
mov Y,ecx
sub edx,Radius
mov X0,edx
mov Xd,edx
add edx,Radius
add edx,Radius
mov X1,edx
sub ecx,Radius
mov Y0,ecx
mov Yd,edx
add ecx,Radius
add ecx,Radius
mov Y1,ecx
mov ecx,Radius
cmp ecx,0
jle @airdone
push eax
mov eax,Radius
imul eax
mov R2,eax
pop eax
```

@airloop:

```
{start of main loop}
mov ecx,Xd
push eax
sub ecx,X
mov eax,Yd
sub eax,Y
imul eax
mov D2,eax
pop eax
{D2, square of the distance of current Xd,Yd from centre now calculated and stored}
call @airpointok
cmp edx,0
je @airpointdone
{now know current point OK and have it's address in edx}
mov ecx,[edx]
push edx
push ecx
{get pixel color value and save pixel address and color on stack}
and ecx,0ff00000h
mov newColor,ecx
{grab fourth byte of color value and store in newColor}
pop ecx
push ecx
and ecx,0ff0000h
```

```

shr ecx,16
mov edx,Color
and edx,0ff0000h
shr edx,16
call @airblend

```

{recover color value but maintain stack status, isolate Red value and shift right so that Red intensity is in range 0->255 to keep subroutine @airblend happy. Do same with color value to be applied. Call @airblend to blend these color values according to status of R2 and D2, returning modified value in ecx}

```

shl ecx,16
{shift back to position of red intensity}
mov edx,newColor
xor edx,ecx
mov newColor,edx
{update newColor}
{now do this again for the Green values}
pop ecx
push ecx
and ecx,0ff00h
shr ecx,8
mov edx,Color
and edx,0ff00h
shr edx,8
call @airblend
shl ecx,8
mov edx,newColor
xor edx,ecx
mov newColor,edx
{and again for Blue}
pop ecx
and ecx,0ffh
mov edx,Color
and edx,0ffh
call @airblend
mov edx,newColor
xor ecx,edx
pop edx
mov [edx],ecx

```

{finally recover address of pixel, and update using newColor}

@airpointdone:

{and we end with the standard loop control checks}

```

mov ecx,Xd
inc ecx
mov Xd,ecx
cmp ecx,X1
jle @airloop
mov ecx,X0
mov Xd,ecx
mov edx,Yd
inc edx
mov Yd,edx
cmp edx,Y1
jle @airloop

```

@airdone:

end;

Implementing routines for drawing squares and circles should now be within your grasp. Triangles can be tricky though. Nevertheless you have in your hands all of the tools required.

Debugging your code

To conclude this article a few words about debugging seem in order. It is very easy to set up watches, program break's, and traverse Delphi programs a line at a time. The same is true, even when using assembler. All one needs to do, is add the four 32bit general registers eax, ebx, ecx and edx to one's watch list, and see the effect of each line of assembler.

When dealing with the stack try numbering each push, giving the same number to each corresponding pop. It is usually best to do this before running the code for the first time.

Where possible break down complex algorithms into small relatively simple sub-routines, and make as much use as possible of local variables. Both these courses of action will hinder your code's performance, but you are more likely to produce code that works.

and finally

Enough has been covered in this article, for you to explore the possibilities of assembler. However the majority of assembler instructions actually available to you, have been ignored. Should you wish to learn more, may I suggest Borland's Turbo Assembler, just for the manuals, and Wrox's Assembly Language Master Class, which is in my opinion the finest book of its type available. Neither of these products directly address the use of assembler in Delphi, nor in Windows 95, but both give a good grounding in assembler algorithm design.

By orwah ali essa .. Syria 2005-2006

www.orwah.net

E_mail : orwah@orwah.net