This is a legacy piece, published in AIExpert magazine. I include this because I think Prolog provides many advantages for graphics programmers.

Calling dynamic link library functions from Visual Prolog

Visual Prolog, a new 'visual programming' product from Prolog Development Center is an excellent programming environment that can be employed in many contexts. Most programmers would likely acknowledge Prolog as an exemplary prototyping tool, while others would go even farther, to agree with me that it is more than adequate as an all-round programming environment. In fact, I have found it so generally useful that it is has become my tool of choice for most programming projects. I find (of course, these matters are largely matters of one's conceptual dispositions) that a Prolog programme has an impressively clear structure—I rarely have to re-read lines of code, even when they are written by other programmers, to figure out exactly what they are doing. I find that Prolog's logical structure (and especially the way it segregates the various conditions that make a predicate true into different clauses) makes Prolog a very good tool to think with and, consequently, that the distance separating the problem specification and the working code is much shorter in Prolog than any of the other languages I work with.

Something that no reasonable programmer would be willing to forsake for all the advantages of a rapid prototyping language, however, is the ability to call upon the many useful dynamic link libraries that are available. These libraries are usually written in 'C' and for 'C' programmers. The challenge of calling 'C' functions from Prolog can seem daunting initially—especially because 'C' libraries so often pass values by reference, and the means for accomplishing in Prolog what 'C' programmers achieve by using pointers to sets of values is not immediately obvious and, even when the general principles of doing so have become clear, one still confronts many pitfuls (not all of which have been documented.) I propose to explain how to call the `C' functions contained in a dynamic link library and to warn of the many pitfalls that confront the unwary programmer as he or she attempts to do this. With an adequate knowledge of the tools required, the ordinary degree of vigilance required by any programming, and special knowledge of a few undocumented pitfalls, the task really is not difficult. I write primarily for 'C' programmers who might be willing to give Visual Prolog a try; however, since I treat some undocumented pitfalls (and since the general task of calling on dynamic link libraries is so important), the information presented here should have some value to seasoned Prolog programmers.

As an example for illustrating how to call the 'C' functions included in a dynamic link library from Visual Prolog I have chosen Visualib, an comprehensive graphics and rendering library from Visual Tech, Co. (P.O. Pox 901-413, Kansas City, Mo. 64190; tel (219) 489-0235; fax (816) 746-6618.) I have chosen the Visualib library, partly because of the library's quality, partly because it is available on many bulletin-boards in a shareware version, and partly because graphics applications are so enjoyable yet are the sort of application to which Prolog is rumoured to be unsuited. I hope to show that, to the contrary, Prolog provides powerful facilities for modelling graphics objects, and that this strength in modelling problem domains provides a programmer actually interested in producing graphics effects with important conceptual advantages. As importantly, the problems that one encounters with this library are typical of, but

somewhat more demanding than, the problems that we commonly encounter in creating an interface between Visual Prolog and dynamic link library comprising 'C' functions written in 'C'—if we can handle this one, we can handle just about any library.

For the most part, one calls the 'C' functions contained in a dynamic link library pretty much as one invokes Visual Prolog's standard predicates. That Visual Prolog can call on a library so simply shouldn't be surprising: Visual Prolog is written in 'C', and a major component of the package you purchase when you buy Visual Prolog is a library (though not a dynamic link library) named '**prolog.lib**,' which is really a set of 'C' functions that the Prolog compiler links into your '.exe' when your programme code calls them. However, two small differences arise when calling on a function contained in a dynamic link library rather than in 'prolog.lib'. First, you have to tell your compiler not to give up if it doesn't find the code for a few procedures when it is building the '**.ob**j' files —that the code will come along later when you link the '.**obj**' files together with the graphics library to produce the final executable file. You do this in much the same way you do it 'C'—you tell the compiler that the code for certain functions (in Prolog we call them predicates) comes from an external file. You don't do this by labelling them "**extern**," however—rather, you flag the predicates as being "**global**."

There are a couple of restrictions on declaring global predicates. First, you must declare all the global predicates before declaring any non-global predicates and, second, every module in the project must call include identical information about the global predicates and the "**global domains**" (a Prolog **domain** is roughly equivalent to a 'C' **struct**) that the project uses—that is, every module must include all the information about all the global domains and all the global predicates that the entire project uses. While these restrictions could create some inconvenience for users of previous versions of PDC Prolog, Visual Prolog's new Visual Programming Environment makes the bookkeeping involved in declaring global domains and global predicates very simple: whenever you create a new module, Visual Prolog sets up a "predicates declaration file" (which serves similar functions as a 'C' header file does, and, according to a longstanding convention, has a ".**pre**" file extension) and, optionally, a "domains declarations file" (which according to convention has ".**dom**" as its filename extension.) When it creates these files, the programming environment also inserts the "**include**" statements for them in the project's "**<projname>.inc**" file (a file containing all the "**include**" statements the project requires), and inserts the "**include**" statement that invokes this "<projname>.**inc**" file at the beginning of every module it creates. So long as one enters a declaration for a domain into a "**.dom**" file when one first uses it, incorporates declarations for all global predicates for which the clauses (the source code specifications of the conditions that make a predicate true) in a certain module into that module's predicates declaration file, and declares non-global predicates within the "**.pro**" file itself (the source code file, roughly equivalent to a 'C' "**.c**" file), one encounters no difficulties whatsoever.

But this takes place once the project has been created, while the first of several minor pitfalls involved in building a project like ours (that relies on the MS-Windows API) occurs at the very beginning, as one is creating the project with Visual Prolog's Application Expert. The Application Expert's dialog box includes an edit box, which must be filled in, for stipulating the language in which the project's main file is written; it is best, when creating a dynamic link library, to respond by choosing your favourite 'C' compiler, and not Visual Prolog. The Application Expert also enables one to choose between using the 'winbind' method (which relies on calling functions from MS-Windows API) or the Visual Programming Interface (VPI) method (which uses extraordinarily portable functions for such tasks as putting up a message box or drawing a ellipse, that allow code to migrate from sixteen bit Microsoft Windows projects to

thirty-two bit MS-Windows projects to IBM's OS/2 projects, and automatically generates the code to create windows and resources and, on demand, creates the framework for clauses to handle various Windows messages.) Finally, the application expert aallows one to create either a 'dll' project or an 'exe' project.

Because extraordinary time-savings result from using the Visual Programming Interface, one is inclined to always choose it (as I did when first began this project.) Because, further, it makes good sense to embed the wrapper code for Visualib functions in another dynamic link library, one is inclined to choose to create our project as a "dll" (for this allows one simply to call upon the **gluevlib.dll**, and that library, through the **IMPORTS** section of the **gluevlib.def** file, takes care of calls to **visualib.dll**.) The most appealing combination of choices, then, is to create this project as dynamic link library project that uses the Visual Programming Interface. However, when I attempted to build such a project using a beta version of Visual Prolog, I discovered that the Visual Programming Interface really doesn't support dynamic link library projects. At first I tried to find a work-around, and when these efforts failed, I contacted the Prolog Development Center. They initially expressed surprise that the VPI would not support such a project. On looking into the matter, they confirmed that the Visual Programming Interface does not provide comprehensive support for "dll" projects, and told me they would try to eliminate the restriction before the final beta version. Despite what seem to have been good efforts on their part, they have not, and the release version of Visual Prolog has now appeared with a note in the manual that states that the VPI provides only limited support for dynamic link library projects (though no message box warning appears if one chooses this combination of options.) The Prolog Development Center plans to overcome this limitation if future releases. In the meantime, projects that result in a dynamic link library should be built using the 'winbind' method.

One confronts another possible pitfall at this point. Suppose you choose to create a "**.dll**" using the 'winbind' method. Then you notice that the main Application Expert dialog box includes a button labelled "**code generator**," and you click on it to make sure that you are aware of what the Application Expert is doing. Among other things, the code generator dialog box that pops up when you push the "code generator button" asks about the directory into which you want to put your resources—and tells you that, unless instructed to do otherwise, it will put the resources in a "**RES**" directory the Application Expert generally creates under the project's main directory. If you click on the "code generator" button and then accept the defaults instead of cancel (a not implausible series of steps) and then return to the main Application Expert dialog box, and press the "create" button (to create the project), you find you get a '**7005 error**', which (by starting the Visual Prolog help file and searching for "**7005_error**") you discover means "access denied"—access to some file has been denied you. The problem seems to be that projects created using the 'winbind' method do not set up a "**RES**" directory under the project's main directory. Visual Prolog counts on this directory to store the resources used by VPI applications, and does not create it when it creates '**winbind**' projects (for which the Application Expert does not automatically generate a set of resources, such as bitmaps for toolbars, as it does when the VPI method is used.) Because the "**RES**" directory does not exist, the Application Expert cannot create a "**<projname>.res**" file in the directory and issues the '**7005 error**' message.

What is worse, if you were to procede at this point, you would find, if you were to check, that that project's include file, a file with the name does not exist. The Visual Development Environment expects to insert information about the include files for all the modules it creates into "**<projname>.inc**". When you load an already existing module into your Project Window or ask it to create a new module for you, the usual dialog boxes still appear, that ask what you want

to do with the "**include**" statements for it. If you accept the defaults and tell the project manager to put them in the project's "**.inc**" file, you will find, when you come to compile, that you lack the "**include**" statements you require. Apparently the project manager went to put the "**include**" statements into "**<projname>.inc**" and, since the file did not exist, simply threw the statements away (although the programme manager did nothing to alert you to the fact.)

The problems can compound still further. Suppose you figure out that you lack "**include**" statements for the modules you created and so you create "**<projname>.inc**" and its contents by hand. Suppose, further, that realize that project has an uncharacteristic directory structure—that it lacks a "**RES**" directory—and so you create the directory on you own, or, alternately, you pull up the "code generator" dialog box and tell the Application Expert to put the resources in the main directory or the "**OBJ**" directory. You try once again to create the project (by pressing the Application Experts "create" button.) At this point, you find that you get a '**1041 error**' message. The Visual Prolog help file tells you that a '**1041 error**' means that you attempted to "assert a second instance of a fact declared as determ." Apparently, every successive attempt on the Application Expert's part to create a project under a given name involves its endeavouring to re-assert a "deterministic fact" (a fact that can only be satisfied by a single condition.) These attempts at asserting a deterministic fact more than once results in your failing again to create the project.

Frustration at this point can lead to even more serious problems. If you try to repeat you steps, you find yourself giving the Application Expert the go-ahead and overwrite files it created when it first created the project—after all, you think, you haven't yet done anything with files, and since your previous efforts at creating the project failed, you would probably be better off remaking everything that Application Expert created on its own. If you go along telling the Application Expert that it is okay to overwrite the files it created, Murphy's Law guarantees that you end up overwriting one of your own files (as I did just after finishing an earlier draft of this article.) While the information I have provided should protect you against such adversity, I don't feel that even this knowledge provides sufficient protection (as my recent mishap points up.) I recommend the following cautions concerning the Visual Prolog Application Expert: 1) Never create a project in an existing directory. 2) Never continue with a project that the Application Expert failed to create without issuing a warning message. And, most importantly, 3) never, never undertake to create a project without first backing up any files that think you might use in the project.

A further issue relating to the Application Expert is that "**winbind.lib**" must be included in every project built with the 'winbind' method; "**winbind.lib**" includes predicates which masquerade as 'C' macros for the Microsoft Windows API, as well as a predicates in which lists take the place of objects that the Windows API models with arrays (and which call all upon the the native Windows API functions indirectly.) The 'winbind' method relies primarily "**windows.pre**," "**windows.dom**" and "**windows.con**", files that contain, respectively, the predicate declarations, the domain declarations and the constants declarations for the Microsoft Windows API. The most efficient way of including these files in your project is to properly declare the paths to directories containing these files in the Visual Prolog Application Expert's "Directories" dialog box (by appending these paths to those which the Applications Expert provides automatically. Visual Prolog's dialog box for this purpose conforms to the convention of using semi-colons to separate various paths to the set of alternative include directories.) Otherwise, you can insert statements of the form **include "f:\\run\\winbind\\include\\windows.pre"** and **include "f:\\run\\winbind\\include\\windows.con"** in the project's "**<projname>.inc**" file (discussed below) and "**winbind.lib**" in the list of modules maintained by

the project's main dialog box, its Project Window—a process that is anyway sometimes more efficient that using the "Directories," dialog box if you running Visual Prolog from the CD-ROM on which it is delivered, as the "browse" machinery will not take you to the CD-ROM drive.

While the '**.h**' files that 'C' programmers write typically starts with several statements with the following form:

extern void rect(int,int,int,int);
extern int putpixel(int,int,int);

a comparable '**.pre**' file begins with the heading:

GLOBAL PREDICATES

Then, under this heading there appear several statements of the form

rect(integer,integer,integer,integer) - (i,i,i,i)
putpixel(integer,integer,integer,integer) - (i,i,i,o)

These predicate declarations highlight a differences between standard or "Clocksin and Mellish" Prologs and versions of Prolog from the Prolog Development Center—a difference that accounts for PDC Prolog's superior efficiency and speed. Unlike standard Prolog, PDC Prolog (and now Visual Prolog) is a typed language and, accordingly, the types for all the variables in a predicate's parameter list must be specified in the predicate declaration (which occurs in the "**PREDICATES**" or "**GLOBAL PREDICATES**" sections of the programme.) This use of type specifiers also allows PDC Prolog to do tight error-checking—PDC Prolog's error-checking is so tight (and the logical structure of Prolog programmes so clear) that it has been my experience, over years of programming with PDC Prolog, to rarely encounter run-time errors other than the most trivial sorts of errors associated with file availability. For the most part, if a project compiles, it runs without run-time errors (though whether it does what the programmer hoped it would is something that a language's clarity and efficiency can guarantee.)

The domains declaration file includes statements like the following:

GLOBAL DOMAINS
stringlist = string*
/* The asterisk means "an arbitrary number of the type; so a **stringlist** is an arbitrary number of **string**s. */

reallist = real*
list_of_reallists = reallist*
real_array = real*
integerlist = integer*
list_of_integerlists = integerlist*

VL_COORD = real
VL_COORD2D = vl_point2D(VL_COORD, VL_COORD)
VL_COORD2H = vl_point2H(VL_COORD, VL_COORD, VL_COORD)
VL_COORD3D = vl_point3D(VL_COORD, VL_COORD,VL_COORD)
VL_COORD3H = vl_point3H(VL_COORD, VL_COORD, VL_COORD,VL_COORD)
VL_COORDLIST2D = VL_COORD2D*
VL_COORDLIST2H = VL_COORD2H*
VL_COORDLIST3D = VL_COORD3D*
VL_COORDLIST3H = VL_COORD3H*
LIST_OF_VL_COORDLIST2D = VL_COORDLIST2D*

LIST_OF_VL_COORDLIST2H = VL_COORDLIST2H*
LIST_OF_VL_COORDLIST3D = VL_COORDLIST3D*
LIST_OF_VL_COORDLIST3H = VL_COORDLIST3H*

vl_vector = real
vl_vectorlist = real*

VL_MATRIX = vl_matrix(reallist)
VL_VECTOR1 = vl_vector(real)
VL_VECTOR2 = vl_vector2(real, real)
VL_VECTOR3 = vl_vector3(real, real, real)
VL_VECTOR4 = vl_vector4(real, real, real, real)
VL_MATRIX2 = vl_matrix2(real, real, real, real)
VL_MATRIX3 = vl_matrix3(real, real, real, real, real, real, real, real, real)
VL_MATRIX3D = vl_matrix3D(real, real, real, real, real, real, real, real, real)
VL_MATRIX3H = vl_matrix3H(real, real, real, real, real, real, real, real, real, real, real, real, real, real, real, real)

VL_ Matrix3DPro = vl_matrix3DPro(list_of_reallists)
/* or, to make for even closer type-checking, matrix3DPro can be:
vl_ **matrix3DPro([[real, real, real],**
[real, real, real],
[real, real, real]] */

and **matrix3HPro = matrix4DPro(list_of reallists)**

/* or, to make for even closer type checking that matrix3HPro can be

matrix4DPro([real, real, real, real],
[real, real, real, real],
[real, real, real, real],
[real, real, real, real]] */

VL_VECTORLIST1 = VL_VECTOR1*
VL_VECTORLIST2 = VL_VECTOR2*
VL_VECTORLIST3 = VL_VECTOR3*
VL_VECTORLIST4 = VL_VECTOR4*

VL_POINT2D = vl_point2D(real, real)
VL_POINT2H = vl_point2H(real, real, real)
VL_POINT3D = vl_point3D(real, real, real)
VL_POINT3H = vl_point3H(real, real, real, real)
VL_BOX2D = vl_box2D(real, real, real, real)
VL_BOX3D = vl_box3D(real, real, real, real, real, real)

As I noted above, these domains resemble 'C' **struct**s; the last domain would be

equivalent to a ‘C’ struct **typedef struct box3D_tag{ double XMin,**
double YMin,
double ZMin,
double XMax,
double YMax,
double ZMax} box3D;

It is not imperative to group the coordinates as pairs, triples and quadruples as I have done. But grouping them this way (the glueing is done by the functor “**vl_point2D**,” “**vl_point2H**,” etc.) is an easy thing to do in Prolog, and doing so effectively models familiar graphics objects and allows graphics programmers to think about graphics problems in ways that their mathematics training has made them accustomed (something that is especially useful with functions in the Visualib library, which renders the curves and surfaces that some readers will have studied in their differential geometry course.) That it provides such simple and elegant means for modelling problem domains is, in my view, one of Prolog’s strengths.

Further on the matter of domain declarations, I preface functors and domain declarations with “vl_”; I do the same with the names of predicates (functions) that I create. The **gluevlib.dll** that we are working towards will serve as an interface between applications that we will build in Visual Prolog and a dynamic link library. An application might call serveral library, and by prefacing the names of predicates and of domains with identifier which associates it with the Visualib library, I eliminate protential of introducing new conflicts between the names of domains or predicates that might be called from different libraries (beyond those that all ready exist because their authors did not take the trouble to give their predicates names that, by any reasonable measure of likelihood, are likely to be unique. There is an overhead in this—our practice of creating domains and functors with unique names means we must write additional to convert from one librar’s domains to another library’s domains; thus, for example, we must have a conversion programme to convert between “visualib points” (**vl_point2D**(**real**, **real**)) and, say, Fastgraph points (**fg_point**(**integer**,**integer**). Furthermore, we have go through the bother of remembering that the appropiate functor for point in two-dimensional, non-homogeneous coordinates is “**vl_point2D**” while the appropriate functor to use when addressing a point in two-dimensional, non-homegeneous coordinates using a routine from the Fastgraph library is “**fg_point2D**.” This can be a nuisance—if we were to represent 2D points as often as possible as a pair of integers, then we could call functions from several different libraries without entangling ourselves in unnecessary conversions amongst the special domains created for each library. This consideration is particularly important when we employ predicates that convert domains defined for individual libraries into C arrays —similar code can be used for different libraries, but will have slight differences that dictated by the specific structures being modelled. It is important, when modifying the boiler-plate code, to use different predicate names and different domain names for each library. Prefacing predicate names and domain names with an identifier that associates it with a specific library is a good means for doing so.

Certainly, the leaner approach can produce serious problems, as would arise if we failed to take into account that a **vl_point** involves a pair of **real**s, while an **fg_point** uses only a pair of integers; if we were to pass a Visualib function a pair of integers (using our **gluevlib.dll**), we could very well crash the programme. By using functors like “**vl_point2D**” and “**fg_point2D**” we remind ourselves that we are using terms form different domains, so we are much less likely to run into trouble. What is more, our compiler will complain if we try to sent a Visualib function a

fg_point2D or if we try to sent a Fastgraph function a **vl_point2D** (though it would not if points were declared to a pair of integers. Furthermore, the compiler will complain if we try to use a **real** as a Fastgraph coordinate, while no complaint would result if we declared a point to be a pair of integers and then sent a Visualib function which required point (i.e., a pair of coordinates) such a pair. .

In another file, "**gluevlib.dom**," another sort of global declaration appear:

```
GLOBAL DATABASE - vl_handles

determ vl_instance(HANDLE)        /* Instance handle */
determ vl_mainWindow(HWND)        /* Window handle */

GLOBAL DATABASE - vl_coord_mode
determ vl_current_coordinate_mode(integer)
```

These are declarations for what Prolog programmers call "database predicates." Some programmers include these in their predicate declaration files, while some include them their domains declarations files, since database predicates have some features of predicates (e.g., they can be stipulated in a clause as a establishing a condition that must be fulfilled for a predicate to be true) and some features of domains (e.g., they don't do anything other than to store variables of the specified type.) The qualifier "**determ**" for deterministic predicates, including deterministic database predicates, can be omitted, as "**determ**" is the default for global predicates is PDC Prolog. However, if there are nondeterministic predicates, the use of the qualifier "**nondeterm**" is required.

Predicate declarations look like this:

```
GLOBAL PREDICATES

vl_centre_dlg(HWND) - (i)
VL_ShowMessage(string,string)  - (i,i) languauge C as "_VL_ShowMessage"
vl_check_true_returned(string,BOOL) - (i,i)
PolyPolygon2DPro(HDC,LIST_OF_VL_COORDLIST2D) - (i,i) language PASCAL
PolyPolygon2HPro(HDC,LIST_OF_VL_COORDLIST2H) - (i,i) language PASCAL
PolyPolygon3DPro(HDC,LIST_OF_VL_COORDLIST3D) - (i,i) language PASCAL
PolyPolygon3HPro(HDC,LIST_OF_VL_COORDLIST3H) - (i,i) language PASCAL
```

Including the expression "language C" or "language PASCAL" simply specifies in the predicate declarations indicates the manner that the that the function calls upon its parameters —whether it passes its parameters according to the 'C' calling convention (calling parameters from right to left) or as Pascal does (calling its parameters from left to right); thus the "language C" and "**language PASCAL**" statements serve much the same role that the reserved words "**cdecl**" and "**PASCAL**" do in `C'. The functions exported from a dynamic link library always use Pascal's calling convention, so the predicate declarations for these functions will always include the expression "**language PASCAL**".

The letters in the brackets after the dash give what Prolog programmers call the predicate's "flow pattern." That is, they tell us whether, when the predicate is called, a parameter has a value (or, as Prolog programmers say, is "bound") or whether it assumes a value as the predicate performs whatever actions it is programmed to perform. The letter 'i' indicates that the value is an input value, (i.e., that is, it is bound when the predicate is called.)

So, for example, the predicate "**centre_dlg()**" contains a single value in its formal parameter list —it is the handle of the dialog window to be centred on the screen, and, as one might expect, the value is bound when the predicate is called (for we have to tell the predicate, by using the "window handle," which window we want to put at the centre of the screen.) However, we might also have a predicate "**make_window(HWND)**" which would create a new window. In this case, the window handle would be returned when the window is created, so the flow pattern for this predicate would then be (o), since the parameter is an output parameter—a parameter that is free (i.e., not bound) when the predicate is called, and becomes bound (assumes a value) only as the predicate performs its function. Or we might have predicate "**create_child_window(HWND Parent, HWND Child)**" which would have a flow pattern (i,o)—we would give the predicate the handle to the Parent window and return the handle to the child window.

The difference between the input and output variables relates to difference in Prolog's manner of handling two sorts of variables. Whenever a new procedure is begun, a stack frame corresponding to the procedure is created. The stack frame consists, first, of a return pointer, the address the instruction point returns to when the procedure has been completed. It also contains the pointer that points to the parent frame. Arguments that are passed to the function with a value (i.e., input variables) are pushed onto the CPU's stack one by one. When the procedure has finished executing the instruction pointer returns to the parent procedure. If the procedure is a branch of deterministic predicate, the stack frame corresponding to the procedure is removed from the stack. If the function does not completely resolve, the stack frame remains on the stack so that this information can be reused if necessary.

Thus, when the argument is an output argument, the function cannot simply write the value into a location on the stack as it does with an input variable (since the stack is cleaned up when the routine finishes, and the values would be lost.) Instead, the Prolog compiler pushes onto the stack the address of where the value is to be written. This is just what a 'C' compiler does, of course, for what it passes is the address of (i.e., a pointer to) the variable instead of the variable itself. The address is that of stack location of the variable in the parent predicate, as the output of a procedure sets the local variables in the procedure's parent procedures. The use of flow patterns allows Visual Prolog to avoid involving itself with pointers, indirection and dereferencing variables as 'C' does, for flow patterns enable Visual Prolog to understand when variables and when variables' addresses (points to variables) should be used

When the clauses (the Prolog code) for a predicate occur in the same module as the predicate declaration, the Visual Prolog compiler can determine the flow pattern by itself. All it needs to do is to determine whether a variable is bound or free whenever the variable is used, by analyzing the code structure. The compiler will assign to bound variables the type '**integer**' (or whatever type is appropriate), while to those that are not bound it will assign the type 'pointer to integer' (or whatever type is appropriate).

However, when the clauses for a predicate appear in one module and the predicate is called in another, the compiler cannot decide whether the parameters for that predicate are bound or free, since it does not have access to the code in the clauses for that predicate, i.e., the conditions that must be true for the predicate to be true. So we have to tell the compiler whether to allocate memory for an integer (or whatever the appropriate type) or whether to allocate the memory for a pointer to an integer (or what the appropriate type). Accordingly, we must include a statement of the flow pattern along with a declaration of the parameter types when we declare the predicate; when it sees a parameter flagged as an input parameter of certain type, the compiler will reserve memory for that type (for a '**char**', if it is parameter is

declared to be of type '**char**' and flagged as an input variable) while, if it sees a parameter flagged as an output parameter of the certain type (for example, if a parameter is declared to be of type '**char**' and flagged as an output variable) the compiler will reserve memory for a pointer to that type (that is, in our example for a pointer to a '**char**'—or, what is the same, for a memory address of dword length that gives an address that will contain a character when predicate has run.)

Our formal parameter list for "**create_child_window(HWND Parent, HWND Child)**" indicates a new feature of Visual Prolog: it is now possible to follow the stipulation of a parameter's type with a name of the parameter. This change aligns Prolog predicate declaration files even more closely with the 'C' header files, and provides for mnenomics that inform us as to the parameter's role (like those that have long been a feature of 'C' header files) to be included in the predicate declaration file.

Predicate declaration files usually contain declarations for functions written in Prolog. They are not restricted to such predicates, however; it is possible to call functions written in 'C' from Visual Prolog. Such functions must be modeled with Visual Prolog predicates, and must also be declared in the predicates declaration file (even though this declaration is all we create, and we allow the linker to find the code for the predicate in an "**.obj**," "**.lib**" or "**.dll**" file.) There is hardly any distinction between how Prolog calls a global Prolog predicate and how it calls a 'C' function (such as might be contained in a dynamic link library.) In fact, it is possible to write some of the modules in Visual Prolog project in 'C', to compile the 'C' code into an '**.obj**' file, and include the name of the 'C' '**.obj**' file among the modules make up the Visual Prolog project (in fact, we shall have to do just this to finish our **gluevlib.dll**.) The following provisos constrain the use of 'C' based code in a Visual Prolog project—when compiling the 'C' code, one must compile in "case insensitive" mode and use the large memory model; this means, effectively, that stipulations that pointers are "**FAR**" pointers are redundant as far as Visual Prolog is concerned. (It is fortunate that the functions exported from dynamic link libraries are "**FAR**" functions, with "**FAR**" pointers, which means effectively, that they conform to this requirement.) If **struct**s or **union**s—that will be modeled with compound domains in Prolog—are used, the "byte alignment" option must be turned on; it is a good idea to always do this, regardless of whether or not one's code uses **struct**s. Further, to avoid confusion, it is best to compile with the "preceding underbars" option turned on, as 'C' compilers generally employ to this convention as their default. Then, whenever one calls a 'C' function from Prolog, one follows the predicate declaration with the statement: "**as _FunctionName**" where "**FunctionName**" is source-code name of the the 'C' function; having followed the convention regarding underbars, we preface each function name with an underbar and avoid the hassles of determining which function names should have preceding underbars and which should not.

And, finally, we will remember to use our preceding "_vl" or "_VL" identifiers, since we are really adapting boilerplate code for the Visualib dynamic link library. We will reuse the same boilerplate code, *mutatis mutandis*, for other libraries, and we want to be certain not to have name conflicts.

In sum, predicate declarations in Visual Prolog often take the form:

integer VL_CoordList2DToBinary(VL_COORDLIST2D,BINARY) - (i,o)
integer VL_CoordList3DToBinary(VL_COORDLIST3D,BINARY) - (i,o)
integer VL_VectorList2ToBinary(VL_VECTORLIST2,BINARY) - (i,o)
integer VectorList3ToBinary(VECTORLIST3,BINARY) - (i,o)

These predicates have (i,o) as their flow pattern, since they are given list of coordinates of a

specified type and give back a value of a binary type (a type on which we shall soon comment.) Furthermore, the predicates return an integer value as 'C' functions usually do (in their cases, the number of elements in the lists passed to them.) Return values are not a feature of "standard" or Clocksin & Mellish Prolog, as Prolog (which short for "Programming in Logic") is modelled on the predicate calculus and not on the mathematical conception of a function (with its range and domain variables) as the 'C' function is. Visual Prolog provided for return values to make it possible to send and return values to the Windows operating system. Visual Prolog's ability handle return values makes it convenient to call upon functions in a dynamic link library that have return values; the return parameter type is simply stipulated before predicate name (as shown above.) Allowing functions to have a return value also allows Visual Prolog programmers to copy the function declarations from the header file for a 'C' library almost verbatim—you have to substitute "**integer**" for "**int**", and, under appropriate conditions, "**string**" for "**char***", "**real**" for "**double**" etc—thereby making what this project demands of us considerably less onerous.

Another small pitfall confronts us here. 'C' makes extensive use of buffers (for example for strings) which it allocates before calling a function, even though the buffer is filled up ("acquires its value") as the functions runs. Since the variable (the buffer) acquires its value as the function runs, it might seem that such a variable should be an output variable. However, this is not so—since the variable is allocated, its predicate declaration should show the buffer as being an input rather than an output variable. Prolog generally only allocates buffers when it assigns them their values, not before. So how do we get the values for "the buffer" that the 'C' functions requires before we assign them values (which the function itself does as it runs.

The answer comes from recognizing that such buffers can be modeled in Visual Prolog with either of two types, strings or binaries. Buffers for strings can created in the following (somewhat unusal) way:

```
GLOBAL PREDICATES

CFunctionName(string) - (i)

CONSTANTS
buffer_size = 80

CLAUSES
. . .
  str_len(OutBuff, buffer_size),
  CFunctioname(OutBuff),
. . .
```

Binaries we shall have occasion to consider later. But the suffer the same fate as our '**OutBuff**' variable does—they end up being overwritten with new values. This is the essence of understanding binaries.

We now can rewrite 'C" functions that use **int**s or **char***s or **double**s or other single values. What about those functions use pointers to arrays of values? This is a considerably thornier. It is, as we shall see, possible to implement arrays of **char**s or **word**s or **dword**s, and even arrays of function pointers (which are modelled as **dword**s arrays.) But Visual Prolog, like standard Prolog, doesn't have an array type. However, because Prolog originated as an artificial

intelligence language, it has consider list processing capabilities built into it, and the list is data structure well suited for graphics programming—a polygon, after all, is conveniently represented as a list of points. In 'C' you use arrays instead, so 'C' racks up considerable overhead in having to pass around array lengths in order to access the array elements, etc.—you have to know about how large the array should be, and allocate enough memory for it, and if you are looping through all its members, you have to know, or be able calculate, how to move from member to member (probably using some statement like '**for (int i=0; i < ((sizeof(array1) / (sizeof(int)); i++)**')—while the Prolog machine takes care of all of this for you.

Modelling 'C' arrays as lists in Visual Prolog has evident advantages. Doing this it a bit arduous—but not so difficult as to make the effort unreasonable. To do so, we make use of **binary** terms as intermediaries. Binaries are a special type of term in Visual Prolog. Prolog terms are generally passed by value, and so they cannot be altered by the operations that a predicate performs. That is why Prolog terms often have the form **TransformValue(integer InValue, integer OutValue)**; here we do not alter the value of **InValue**, but look in another variable, **OutVal** (that is free when the predicate Transform value is called, but becomes bound as the predicate performs is work), for the transformed value. 'C,' on the other hand, often passes values by reference. It is as though we could have a predicate **_TransformValue(integer Val)**, that could be handed a variable name (`**Val**') and an initial value for that variable, could massage the variable's value and return a new value that still goes under the name '**Val**.'

Prolog is uses pattern-matching to perform many logical operations, and the pattern-matching on which it relies makes it impossible to assign different names to the different values housed at different times under that a dynamic variable's name. That is, though one might wish to, one cannot handle values that passed by reference **_TranformValue(integer *Val)** by writing it as a 'C' function and then placing the following wrapper around it:

```
TransformValueWrapper(InVal,OutVal):-
        _TransformValue(InVal),
        OutVal = InVal.
```

assuming that when **_TransformValue()** runs, it executes the lines sequentially, so that the final line assigns "**OutVal**" the value that "**InVal**" has assumed while "**_TransformValue()**" ran. This code fragment would simply create havoc.

To interface with 'C' functions that take in a value under a given variable name, massage that value, and return a new value under that same name, Prolog needs the capability of passing values by reference, and not just by value (as all standard Prolog functions do.) That is, the predicate would have to have the capability of accepting a variable's address, finding the value (or the series of values) stored at that address (or beginning at that address), operate on those values, and store the new value or values in the same block of memory as the original variables occupied (or starting at the same memory address as the original block of values.) This is what Visual Prolog's **binary** domain allows us to do. A Visual Prolog **binary** is a special domain for handling data such as bitmaps and memory blocks that Visual Prolog has no other means for representing. The Prolog Development Center's primary reason for introducing **binary** terms was to provide an efficient way for Visual Prolog to interface to non-logical objects, including objects created in other languages such as 'C.' A **binary** term in Visual Prolog is simply of sequence of bytes, while a **binary** created by Visual Prolog is a sequence of bytes preceded by a **word** (on sixteen bit platforms, or by a **dword** on thirty-two bit platforms) that states the number of bytes the sequence contains. Visual Prolog provides the predicates

"**setbyteentry()**," "**setwordentry()**," "**setdwordentry()**," and "**setrealentry()**" to set the value of a term in a **binary** block to a specific value, and "**getbyteentry()**," "**getwordentry()**," "**getdwordentry()**," and "**getrealentry()**" to access stipulated terms in a **binary** block.

Unfortunately, it is not so easy to fetch the value of a particular memory address, convert the contents of address into an item belonging to one of the standard domains (**integer**, **real**, **string**, etc), operate on the values, then store the resulting set of values in memory, starting at the same address as the first element in the original memory block occupied. Fortunately, tools do exist that enable us to do this somewhat more conveniently than Visual Prolog can do unaided. These tools were first constructed by Paul Morrow and posted in the Prolog Development Center's Compuserve forum (PCVENB, Prolog Development Center.) While they did not run (from within a " **.dll**", in Windows 95, and using our many, specially-defined compound domains) in the form I found them, it took only simple modifications (that have made them considerably less elegant than they were in the form I found them.) Readers should not assume that any errors they might find in my version were present in the original—rather they should assume that I am likely their source. However, they should assume that the good ideas came from Paul Morrow.

Morrow's binary tools rely on three principal, special purpose domains: **fields**, **value**s, and **fieldvalue**s, as well as lists of these three principal types. Each of these principal types is a compound object, subsuming terms that also belong to specially-defined types. A **field** consists of a **fieldname** and a **fieldtype** (glued together by the functor, "**field**"), while a **fieldname** is a member of the set {**functor**, **ptr_to_str**, **integer_value**, **string_value**, **ptr_offset**, **num_of_ptrs**} stipulating the type of value the **field** contains (Morrow defined a **fieldname** simply as a **string**, so a **fieldname** could be any arbitrary string of one's devising—that would allow a programmer to tag **fieldtypes** with unique **fieldname** identifiers; however, every attempt I made to use them in this way failed, while used in the way I outline here resulted in success.) A **fieldtype** is any member belonging to the set { **bin_(length)**, **byte_()**, **dword_()**, **string_(length)**, **word_()** }, where "**length**" is itself a domain (of integer type) that we instantiate with integer values that specify the length of the **string** or the number of bytes in the **binary** object; but further, because another library might have a slightly different set of **fieldtypes** (perhaps it might include a long_(), or perhaps it might not need a **dword_()**), it is probably wise, anticipating the future possibility of allowing some of these predicates tobe exported, to ensure that our domains have unique names, by following our convention of using our "vl_" identifier. Thus our "**fieldtype**" becomes a "**vl_fieldtype**" and the set of possible **vl_fieldtypes** is: **vl_bin_(length)**, **vl_byte_()**, **vl_dword_()**, **vl_string_(length)**, **vl_word_()** A **vl_value** is any member of the set { **vl_bin_(BinaryVal), vl_byte_(ByteVal), vl_dword_(DWordVal), vl_string_(StringVal), vl_word_(Word)** }—declared in our domain declaration as "**vl_bin_(BINARY)**," "**vl_byte_(BYTE)**," "**vl_dword_(DWORD)**," "**vl_string_(string)**," and "**word_(WORD)**"—where **BinaryVal**, **ByteVal**, **DWordVal**, **StringVal**, and **WordVal** are variables that contain, respectively, the **binary**, **byte**, **dword**, **string**, or **word** values to be stored at a particular memory index or in a **binary** term.

Here is the complete set of domain declarations for our **binary** tools.

GLOBAL DOMAINS

```
length =  integer
fieldtype =  bin_(length);
             byte_();
```

```
                dword_();
                string_(length);
                word_()
fieldname =  functor;
                ptr_to_str;
                integer_value;
                string_value;
                ptr_offset;
                num_of_ptrs

field       = field(fieldname,fieldtype)
fieldlist    = field*

fieldvalue         = fieldvalue(fieldname, value)
fieldvaluelist    = fieldvalue*

value  = bin_(BINARY);
               byte_(BYTE);
               dword_(DWORD);
               string_(string);
               word_(WORD)
valuelist  =  value*
```

These domains allow us to construct tools that simplify the tasks involved in passing parameters by reference in Visual Prolog. This capacity allows us to dynamically change the values of variables and so to model a typical 'C' function which does not have a separate input and output variable, but modifies the value of a variable passed to it and stores the new value under the same variable name (at the same address.) Here are the predicates declarations for the set of tools:

```
ifndef in_project
GLOBAL PREDICATES

integer movemem(ULONG, ULONG, UNSIGNED) - (i, i, i) language c as "_MEM_MovMem"
enddef

GLOBAL PREDICATES

vl_bin_Create(BINARY,VL_FIELDLIST,VL_VALUELIST) - (o,i,i)
vl_bin_FormatFieldList(VL_FIELDLIST InList, string OutStr) - (i,o)
vl_bin_FormatValueList(VL_VALUELIST InList, string OutStr) - (i,o)
vl_bin_FormatFieldValueList(VL_FIELDVALUELIST InList, string OutStr) - (i,o)
vl_bin_MapToFieldValueList(BINARY InBin, VL_FIELDLIST InFldList,
VL_FIELDVALUELIST OutFldValList) - (i,i,o)
vl_bin_MapToValueList(BINARY InBin, VL_FIELDLIST InFldList, VL_VALUELIST
OutValList) - (i,i,o)
vl_bin_Update(BINARY PassedByRef, VL_FIELDLIST InFldList, VL_FIELDVALUELIST
InFldValList) - (i,i,i)
```

vl_mem_GetValue(ULONG MemIndex, VL_FIELDTYPE FldType, Value OutVal) - (i,i,vl_word_(o)) (i,i,vl_dword_(o)) (i,i,vl_string_(o)) (i,i,bin_(o)) (i,i,byte_(o))

vl_mem_MapToValueList(ULONG MemIndex, VL_FIELDLIST InFldList, VL_VALUELIST OutValList) - (i,i,o)
vl_mem_MapToFieldValueList(ULONG MemIndex, VL_FIELDLIST InFldList, VL_FIELDVALUELIST OutFldValList) - (i,i,o)
vl_mem_PutValue(ULONG MemIndex, VL_FIELDTYPE Type, VL_VALUE ValToStore) - (i,i,i)

vl_mem_PutValueList(ULONG MemIndex, VL_FIELDLIST TypesList, VL_ValueList ListOfValsToStore) - (i,i,i)
vl_mem_Update(ULONG MemIndex, VL_FIELDLIST TypesList, VL_FIELDVALUELIST FldValsLists) - (i,i,i)

The clauses for these predicates are given in the **bintools.pro** listing, and should be self-explanatory. The **vl_bin_Update()** predicate (and it the associated predicates **vl_mem_Update()**, **vl_mem_PutValue()**, and **vl_mem_PutValueList()**) do the dirty work (or at least, for Prolog, the uncharacteristic work) involved in passing parameters by reference, rather than by value. The **vl_bin_Update()** predicate takes a memory address and updates its content so that it is the starting point for a series of values —those contained in the **InFldValList** (the **vl_fieldlist** sent along with the **vl_fieldvalue** list simply allows **vl_bin_Update()** to calculate how much memory is needed, since it contains information about the **vl_fieldtypes** involved and the length of values whose types are not fixed.) Thus when the function is called, the memory address given by **MemIndex** contains one value, and after the predicate it has run, it contains a different value (and so on with the block of memory pointed to by consecutive addresses, up to the extent required.)

These predicates allow one to write routines for converting a user-defined compound domain (say a **vl_point2D = vl_point2D(real, real)** into a **binary** that can serve as an array of **real**s (or **double**s) passed to 'C' function, or to fetch an array from a 'C' function and return it as a Prolog list. The example points up my preference for handling domains in Visual Prolog. One could simply define a point as a **reallist**, and use a single routine (**VL_RealListToBinary()**) to transform the lists of coordinate values into **binary** form; the resulting **binary** could then be passed to the Visualib function. This is not the approach I prefer: I prefer to model graphics objects after my customary ways of thinking about them; and I find it natural to think of line as joining two points—that is, as joining **vl_point2D(X1,Y1)** to **vl_point2D(X2,Y2)**. Moreover, given a larger number of coordinates, it is just all too easy to leave out one from a list as follows, **Coords = { XMin**, **NextX**, **FollowingX**, **PenultimateX**, **FinalX**, **YMin**, **NextY**, **FollowingY**, **PenultimateY**, **FinalY**, **MinZ**, **NextZ**, **PenultimateZ**, **FinalZ }** (and in most cases Visualib would accept the list without screaming) while forgetting the third coordinate in **vl_point3H(X4,Y4,Z4,H4)**, is a little less likely, and anyway, a construction like **vl_point3H(X4,Y4,H4)** would be flagged for you by the compiler. Further, I find the code much more readable: the coordinates are already grouped together into points. What is more, this approach enables one to pass routines points of whatever type (**vl_point2D**, **vl_point2H**, **vl_point3D**, or **vl_point3H**) as variables—**vl_line3H(Point1,Point2)** is certainly conceptually much simpler conceptually than "**vl_line3H([X1, Y1, Z1, H1, X2, Y2, Z2, H2])**"—of course, you can do this in 'C' do, but you can do it Prolog even when if the predicate creates a point (that it it

allows for the use of buffers in the place of output variables), and not simply when it references an already existing object. These advantages increase as one works with increasingly more complex geometric structures—with Hermite surfaces or Bézier curves. Finally, I much prefer this approach, despite its additional overhead, because of the tighter type checking involved—the Visual Prolog compiler will check that I am using **vl_vector3**, and not a **vl_vector2** when a **vl_vector3** should be used, or a **vl_point2H**, and not a **vl_point2D** when a **vl_point2H** is required, but defining **vl_vector3** and **vl_point2H** as reallists, and using a single routine to convert any **reallist** to a **binary** (that would masquerade as a 'C' array) would deprive me of this advantage, which I believe produces net time savings. However, some readers might wish to go the route of using the single data type, the **reallist**, for any collection of coordinates, and a single conversion routine to convert any collection of coordinates into a **binary** that will masquerade as a 'C' array.

Here is the clause for one such predicate (for which, since it is a global predicate, i.e, it can be called from other modules, the predicate declaration is contained in a separated predicate declaration file as: **integer VL_CoordList2DToBinary(VL_COORDLIST2D InCoordList, BINARY OutBin) - (i,o)** and **VL_BinaryToCoordList2D(Binary InBin,integer Count, VL_COORDLIST2D OutCoordList) - (i,o)**)

```
PREDICATES

CoordList2DToBinary_aux(COORDLIST2D,FIELDLIST,BINARY,integer,integer)

CLAUSES

VL_Coord2DToBinary(vl_point2D(X,Y),Bin):-
        FldList = [ vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()) ],
        real_ints(X,XPart1,XPart2,XPart3,XPart4),
        real_ints(Y,YPart1,YPart2,YPart3,YPart4),
        ValList =  [ word_(XPart1),
                     word_(XPart2),
                     word_(XPart3),
                     word_(XPart4),
                     word_(YPart1),
                     word_(YPart2),
                     word_(YPart3),
                     word_(YPart4)],
        vl_bin_Create(Bin,FldList,ValList).

VL_CoordList2DToBinary([],Bin,0):-!,
        term_bin(string,"",Bin).
```

```
VL_CoordList2DToBinary([vl_point2D(X,Y)|Rest],Bin,RetVal):-
        FldList = [vl_field(integer_value, word_()),
                   vl_field(integer_value, word_()),
                   vl_field(integer_value, word_()),
                   vl_field(integer_value, word_()),
                   vl_field(integer_value, word_()),
                   vl_field(integer_value, word_()),
                   vl_field(integer_value, word_())],
% vl_bin_FormatFieldList(FldList,Str1),
% VL_ShowMessage(Str1,"FieldList:"),
        real_ints(X,XPart1,XPart2,XPart3,XPart4),
        real_ints(Y,YPart1,YPart2,YPart3,YPart4),
        ValList =    [ word_(XPart1),
                       word_(XPart2),
                       word_(XPart3),
                   word_(XPart4),
                   word_(YPart1),
                    word_(YPart2),
                    word_(YPart3),
                   word_(YPart4)],
% vl_bin_FormatValueList(ValList,Str2),
        % VL_ShowMessage(Str2,"Value List:"),
        vl_bin_Create(Bin,FldList,ValList),
% VL_ShowMessage("Bin created","Debug"),
        VL_CoordList2DToBinary_aux(Rest,FldList,Bin,1,RetVal).


VL_CoordList2DToBinary_aux([vl_point2D(X,Y)],InFldList,Bin,InCount,OutCount):-!,
                vl_bin_MapToFieldValueList(Bin,InFldList,InFldValList),
% VL_ShowMessage("Making up final FldList", "In recursive clause"),
        AddFldList =[ vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_()),
                      vl_field(integer_value, word_())],
% vl_bin_FormatFieldList(AddFldList,Str1),
% VL_ShowMessage(Str1,"Final AddList:"),
        vl_append_FieldLists(InFldList,AddFldList,NewFldList),
% vl_bin_FormatFieldList(NewFldList,Str2),
% VL_ShowMessage(Str2,"List to date:"),
        real_ints(X, XPart1, XPart2, XPart3, XPart4),
        real_ints(Y, YPart1, YPart2, YPart3, YPart4),
```

```
% VL_ShowMessage("Making up final Fieldvaluelist","Debug"),
        AddFldValList =[ vl_fieldvalue(integer_value, word_(XPart1)),
                         vl_fieldvalue(integer_value, word_(XPart2)),
                         vl_fieldvalue(integer_value, word_(XPart3)),
                         vl_fieldvalue(integer_value, word_(XPart4)),
                         vl_fieldvalue(integer_value, word_(YPart1)),
                         vl_fieldvalue(integer_value, word_(YPart2)),
                         vl_fieldvalue(integer_value, word_(YPart3)),
                         vl_fieldvalue(integer_value, word_(YPart4))],
% vl_bin_FormatFieldValueList(AddFldValList,Str3),
% VL_ShowMessage(Str3,"Final AddFldValueList:"),
    vl_append_FieldValueLists(InFldValList,AddFldValList, NewFldValList),
% vl_bin_FormatFieldValueList(NewFldValList, Str4),
% VL_ShowMessage(Str4,"Fieldvaluelist to date:"),
% VL_ShowMessage("Final update for binary", "Debug"),
    vl_bin_Update(Bin, NewFldList, NewFldValList),
    OutCount = InCount + 1.


VL_CoordList2DToBinary_aux([vl_point2D(X,Y)|Rest], InFldList, Bin,
RetValAccum, OutRetVal):-
        bin_MapToFieldValueList(Bin, InFldList, InFldValList),
% ShowMessage("Making up new FldList", "In recursive clause"),
        AddFldList=[vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_()),
                    vl_field(integer_value, word_())],
% vl_bin_FormatFieldList(AddFldList,Str1),
% VL_ShowMessage(Str1,"AddList:"),
        vl_append_FieldLists(InFldList, AddFldList, NewFldList),
% vl_bin_FormatFieldList( NewFldList, Str2),
% VL_ShowMessage(Str2,"List to date:"),
        real_ints(X, XPart1, XPart2, XPart3, XPart4),
        real_ints(Y, YPart1, YPart2, YPart3, YPart4),
% VL_ShowMessage("Making up new Fieldvaluelist","Debug"),
        AddFldValList =[ vl_fieldvalue(integer_value,word_(XPart1)),
                         vl_fieldvalue(integer_value, word_(XPart2)),
                         vl_fieldvalue(integer_value, word_(XPart3)),
                                 vl_fieldvalue(integer_value, word_(XPart4)),
                          vl_fieldvalue(integer_value, word_(YPart1)),
                         vl_fieldvalue(integer_value, word_(YPart2)),
                         vl_fieldvalue(integer_value, word_(YPart3)),
                         vl_fieldvalue(integer_value, word_(YPart4))],
```

```
% vl_bin_FormatFieldValueList(AddFldValList, Str3),
% VL_ShowMessage(Str3,"AddFldValueList:"),
        vl_append_FieldValueLists( InFldValList, AddFldValList,NewFldValList),
% vl_bin_FormatFieldValueList(NewFldValList,Str4),
% VL_ShowMessage(Str4,"Fieldvaluelist to date:"),
% VL_ShowMessage("Updating binary","Debug"),
     vl_bin_Update(Bin,NewFldList,NewFldValList),
% VL_ShowMessage("Binary updated; now recursing. . .","Debug"),
        NewRetValAccum = RetValAccum + 1,
        VL_CoordList2DToBinary_aux(Rest, NewFldList, Bin,NewRetValAccum,
OutRetVal).


VL_BinaryToCoordList2D(InBin,NoPoints,CoordList):-
        NoCoords = 2 * NoPoints,
        vl_makeup_fieldlist(0,NoCoords,[],FldList),
        vl_bin_MapToValueList(InBin,FldList,ValList),
        VL_ValueListToIntegerList(ValList,IntList),
        VL_IntegerListToRealList(IntList,RealList),
        VL_RealCoordsToCoordList2D(RealList,CoordList).
```

The code should be pretty much self-explanatory. If commenting out markers (the "**%**"s) are removed, the programme reveals its workings as it runs, through the specially defined "**VL_ShowMessage()**" predicate, which calls the standard MS-Windows "**MessageBox()**" function with the handle of the currently active window and "mb_ok" among its parameters.

A feature of the code above requires special comment. Some alert readers might have noticed that our list of types in the set defining **vl_fieldtype** did not contain a **real** (or a **double**)—a sixty-four bit data type. It would be easy enough to incorporate such a datatype, but since Visual Prolog does provide a predicate for breaking a sixty-four bit object into four sixteen bits objects, the effort is probably not worth it. The predicate is **real_ints(real, integer, integer, integer, integer)**, and it has a compound flow—the flow can be either **(i,o,o,o)** or **(o,i,i,i)**; that is, it can either break a **real** down into four sixteen-bit (**integer**) values or compose a **real** (a sixty-four bit value) out of four sixteen-bit (**integer**) values. In the code above, we use **real_ints(X, XPart1, XPart2, XPart3, XPart4)** to break the **X**, a **real** value, into four sixteen bit-values that will be stored in successive memory locations (and so constitute a Prolog **real** or a 'C' **double**.)

Having converted a list of **real**s into a **binary** value with **VL_CoordList2DToBinary()**, we can pass the **binary** term compounded of the values contained in the **reallist** to Visualib functions that expect an array of **real**s (i.e., **double**s.) Thus we can define a predicate **Polygon2DPro()** to call the function **Polygon2D()** form the Visualib library in the following way:

CLAUSES

```
Polygon2DPro(DeviceContext,CoordList):-
        NoPoints = VL_CoordList2DToBinary(CoordList,Bin),
        Type = val(WORD,vl_2d),
        Polygon2D(DeviceContext,Type,Bin,NoPoints).
```

As well as calling the 'C' function, I have also defined **Prolog2DPro()** to call the 2D version of

function (as opposed to its 2H version.) This is the role of the **Type** parameter: **vl_2d** is Visualib constant (defined in **visualib.h**) which needs to be translated into a Prolog constant (with the statement "**vl_2L = 0x1000**") which by preference is included in a **visualib.con** file.) Further, while the Visualib library uses the same function name for both the 2D and the 2H versions, I have preferred to segregate the functionality of the Visualib functions into two Prolog functions, **Polygon2DPro()** and **Polygon2HPro()**, as I could see myself, under the strain of thinking about graphics effects, passing the **Polygon2D()** function a **vl_2H** value when I meant to pass it **vl_2D** and blowing up the programme because the predicate expected three useful values, while I passed it only two.

As I have declared it, the **Polygon2DPro()** and **Polygon 2HPro** predicates takes care of handling the **vl_2D** or **vl_2H** values for me, while Visual Prolog's type-checking protects me handing my polygon predicate calls the wrong type of coordinates (something which the dual functionality of the native library **Polygon2D()** prevents the compiler from checking.) It should also be noted that unlike the 'C' function, the Visual Prolog version keeps track of the number of points passed, so the bookkeeping involved in keeping track of this (which can be considerable when the points are generated automatically or algorithmically) is avoided.

In order to use these predicates, we have to declare them in a predicates declaration file—since the clause for this predicate appears in a file entitled "**draw.pro**" I included the predicate declaration in a file entitled "**draw.pre**"—the predicate declaration reads: **Polygon2DPro(HDC, VL_COORDLIST2D) - (i,i)**. Furthermore, in order for this clause to recognize the Visualib library function, "**Polygon2D()**," it has to have a predicate declaration—this predicate declaration appears in **vlibdll.pre** as: **Polygon2D(HDC,BINARY)**. Note that the second parameter in the predicate's formal parameter list is a "**BINARY**"; from 'C' point of view, this binary term is simply an array of **doubles** (a set of **doubles** stored sequentially in a block of memory.)

The predicates in the **draw.pro**, **surface.pro**, **product.pro** modules treat the functions contained in the Visualib dynamic link library in just this way. The **convert.pro** module provides means converting among the various domains we built to accommodate Visualib functions. The **common.pro** module provide various goodies, such as the means to check the lengths of lists of various types of coordinates, and for throwing up a message on the screen. A bit of its code perhaps requires special mention:

```
vl_error_handler(WndProcName) :-
    lasterror(ErrorNo,FileName,_,Position),
    format(STR,
                "EXIT! in predicate: %\nError:%\nFile:%\nPos: %",
        WndProcName, ErrorNo, FileName, Position),
    VL_ShowMessage("Error",STR),
    exit(0).

MaterialProcWrapper(HandMat):-
   GStack = MarkGStack(),
       trap(MaterialProc(HandMat),_,
               vl_error_handler("MaterialProcCallBack")), ,
   ReleaseGStack(GStack).
```

The **error_handler()** predicate provides means for handling errors that are sometimes returned when a predicate call results in an error—it displays the error (by relying on the PDC Prolog built-in predicate "**lasterror()**." The various **FunctionNameWrapper()** predicates, such as

MaterialProcWrapper() invokes the means for trapping and displaying these error messages, and ensures that stack consumed by the process call is released after the process has terminated. This is the safe method recommended for handling callbacks in PDC Prolog (or in Visual Prolog projects that use the 'winbind' method.)

Having developed clauses for the predicates that we wish to export, and having declared these as **GLOBAL PREDICATES** in our various predicate declarations files and specified there that the predicates conform to the **PASCAL** calling convention, we must prepare to export these predicates. 'C' programmers might have noticed that our Visual Prolog predicates contain nothing equivalent to "**_export**" which we see so commonly in 'C' code for dynamic link libraries. Instead, we rely on the project's "**.def**" file and the Optlinks linker (provided as part of the Visual Prolog package) to create the proper prologs for the exported predicates.

To do so we list the predicate names in the **EXPORTS** section of the "**gluevlib.def**" file.

```
EXPORTS
        WEP             @1 RESIDENTNAME
    MaterialProcWrapper
    NormalProcWrapper
    PixelProcWrapper
    DrawProcWrapper
    BidimArrayProcWrapper
    TridimArrayProcWrapper
    PushTransformation2DPro
    PopTransformation2DPro
    Polygon2DPro
    Polygon2HPro
    PolyPolygon2DPro
    PolyPolygon2HPro
    Polyline2DPro
    Polyline2HPro
```

The standard Windows Exit Procedure (the **WEP()** function) must be included among the exports although, as we shall see, it requires special treatment—or, rather, non-treatment.

We also have to import the predicates (functions) from the Visualib dynamic link library. Again, the Optlinks linker accomplishes this under the direction of the project's "**.def**" file. The **gluevlib.def** file contains statements of the following form to accomplish this.

```
IMPORTS

    VISUALIB.dotproduct
    VISUALIB.crossproduct
    VISUALIB.mixproduct
    VISUALIB.vcos
    VISUALIB.vsin
    VISUALIB.vangle
    VISUALIB.vpara
    VISUALIB.vperp
    VISUALIB.vflat
    VISUALIB.InitializeVisualib
    VISUALIB.ExitVisualib
```

VISUALIB.SelectColor
VISUALIB.SetPoint2D
VISUALIB.SetPoint3D
VISUALIB.SetPoint2H
VISUALIB.SetPoint3H
VISUALIB.SetBox2D
VISUALIB.SetBox3D
VISUALIB.PenColor
VISUALIB.BrushColor
VISUALIB.PaintColor

All the functions in the Visualib dynamic link library must be treated in this way.

Furthermore, as conforms to the usual requirements for dynamic link libraries, we must declare in **gluevlib.def** that the project we are creating is a library (with the "**LIBRARY GLUEVLIB**" statement) so that the data-segment value for the library will be built into the exported predicates' prologs (enabling the library to handle the DS != SS condition that dynamic libraries must handle), that the project results in a Microsoft Windows executable, (with the **"EXETYPE WINDOWS"** statement), set the usual constraints on how Windows memory manager will handle the code (with the "**CODE PRELOAD MOVEABLE DISCARDABLE**" statement) and on how the Windows memory manage will handle data (since this project results in a dynamic link library, the data must be confined to a single segment, a condition declared with the "**DATA PRELOAD SINGLE**" statement), and establish the size of the heap and stack (with the "**HEAPSIZE 18000**" and "**STACKSIZE 20000**" statements.)

Even so, we are not quite ready to build our dynamic link library. Like all dynamic link libraries, our project needs a **LibMain()** function, which we have not yet provided. Doing so is actually a two stage procedure.

Creating "**LibMain()**" function itself brings us to one of the final potential pitfalls (though one that might have been predicted by readers who have remembered that I remarked that it is convenient to tell the Application Expert, when first creating the project, that the main file will be a 'C' file): the most convenient way of creating the "**LibMain()"** function that MS-Windows requires of all dynamic link libraries, and so of performing the various initialization routines MS-Windows requires, is to use a 'C' "**LibMain()**" function and to include that certain standard `C' libraries associated with it. Hence the first stage in handling MS-Windows' requirement for a "**LibMain()**" function is use the following code (assuming you are using Borland C++), which I have placed in a file called "**main.c**":

```
#include "d:\bc45\include\windows.h"
#pragma argsused

int CALLBACK LibMain(HANDLE hInstance,WORD wDataSegment, WORD wHeapSize,
                     LPSTR lpszCmdLine)
{
        if (wHeapSize > 0) UnlockData(0);
        return (1);
}
```

Compile this code to a **main.obj** file, then enter the resulting **main.obj** into Project Window's list of modules.

The second stage in creating the code for handling the "**LibMain()**" function results from the fact that Visual Prolog calls "**LibMain()**" indirectly, through a **PrologLibMain()** predicate, for which we need a clause. Here it is:

PrologLibMain(_,_,_,_,1).

Despite the fact that the **WEP()** function/predicate is listed in the project's "**.def**" file, it does not need to be included in **main.c**, or anywhere else. However, 'winbind' projects do require a dummy goal (which in fact is never called) for, except when the Visual Programming Interface is used, Visual Prolog will not compile a project unless a goal is present. To satisfy the need for a goal, we simply write this:

GOAL
beep().

To use our **main.c** file as the project's main file, standard 'C' libraries must also be included in the object. The easiest way to do this is to select the Visual Development Environment "/options/project/make options/symbols" menu item and insert the following line (with appropriate adjustments for the directories you use, near the top of the file that pops up when you select this menu item. (It is the second line in my symbols file.)

clibs=d:\bc45\lib\cwl.lib d:\bc45\lib\emu.lib d:\bc45\lib\mathwl

It would be prudent, since the symbols file is now popped up, to check that the symbol **$(PRODIR)**, is identified with the name of the directory where your Visual Prolog executable is actually located—I run Visual Prolog off my CD-ROM drive, which is **'f:**,' so the line in the symbols file that defines the meaning of the symbol indicating the path to the prolog directory should read: "**$(PRODIR) = f:\run\bin**." You are wise to check the correctness of this identity, because the Visual Prolog project manager often seems to think that **$(PRODIR) = c:\vpro\bin**, and often sets up my link files this way. When this happens it cannot, for example, find the Optlinks linker which supposed to be in the **$(PRODIR)** directory. A good way to check the symbols that Visual Prolog uses when compiling and linking is to examine the make file that pops up when you select the "/options/project/make options/preview script" menu item.

In addition, one needs to include some startup code contained in the 'C' library "**c0dl.lib**." To indicate to the Optlinks where to find this startup code, one should again activate Visual Development Environment's /option/project/make options/symbols menu item and insert the following line into the file that the menu item pops up.

initobj=d:\bc45\lib\c0dl

The "**PDCGroup.obj**" file is a final requirment, and it which must be linked into the executable (that is, into the dynamic link library) before any "**.obj**" files generated by 'C' are called. The easiest way to do this is place the **pdcgroup.obj** file right at the beginning of the files in the list of ".**obj**" sent to the linker. The best way to do this is to go into "/options/project/make options/symbols" menu item and modify the **PROJECT_OBJ** line so that it reads:

PROJECT_OBJ=f:\run\lib\win16\bc\pdcgroup + OBJ\MAIN + OBJ\COMMON + OBJ \CONVERT + OBJ\ARRYLIST + OBJ\DRAW + OBJ\MACROGLUE + OBJ\PRODUCT + OBJ \SURFACE + OBJ\TRIG + OBJ\VECTMATH + OBJ\BINTOOLS + OBJ\ARRYLIST

(assuming that you our running Visual Prolog off your CD-ROM drive, that your CD-ROM drive is "**f:**," and that **pdcgroup.obj** file appears in the directory specified for it, and all your object files

are contained in the **OBJ** subdirectory (which the Visual Prolog Application Expert, as its default condition, sets up for you.)

Nearly done. However, we would encounter a final potential pitfall if we were to try to compile the code at this point—we would find that the Optlinks linker returns the message:
"f:\run\winbind\lib\winbind.lib(windll) Offset 010D2H Record Type 90
Error 1: Previous Definition Different : WEP

Project not built.
The problem is that **winbind.lib** contains a module "**windll**" that contains a function **WEP()**, (as running tlib.exe, or a comparable utility, on it, to produce a listing file, will confirm) and that **WEP()** is defined in both the Borland library (**c0dl.lib**) and i **winbind.lib**; the two versions of the **WEP()** makes the Optlinks linker protest. The solution is to remove the "**windll**" module from the **winbind.lib** (using tlib.exe or some comparable utility to produce a new version of "**winbind.lib**") and to link in only the code for **WEP()** contained in the Borland **c0dl** library (or that in **c0dl** library for whichever 'C' compiler you use) by substituting your new **WEP()**-less **winbind.lib** for the version that delivered by the Prolog Development Center. (You can make this substitution by selecting the "/options/project/make options/symbols" menu item and substituting its path and filename for the original version in the symbols file, under the heading "prolib." That is use insure the relevant line in your symbols file reads:

pdclibs=c:\prolog\windows\gluevlib\obj\winbind.lib $(prolib) (where **$(prolib)** is defined as **prolib=f:\run\lib\WIN16\PROLOG.LIB f:\run\LIB\WIN16\WIN16.LIB**)

The dynamic link library compiles. The final thing one needs to do is to use Implib, or some comparable facility, to create a import library for the dynamic link library so that the linker will be able resolve calls to the dynamic link library made by applications you create.

Now you can take the library out for test run by creating a test driver programme. You can rely on the efficiency of the Visual Prolog's Visual Programming Interface to assist you. One creates a new project (that I have imaginatively entitled "test.") Select Window in the application's Project Window, then create the skeleton code to handle the 'Create' message. Then click on the button to edit that code, and insert the following.

```
LibHandle = LoadLibrary("gluevlib.dll"),
check_libhandle_value(LibHandle),
InitializeVisualib(),
Wnd = GetActiveWindow(),
DC = GetDC(Wnd),
BrushColor(DC,3),
PenColor(DC,3),
Sphere(DC,60),
Ellipsoid( DC, 80, 40, 60),
ViewerHand = NewViewer(vl_twod),
ViewerRect = rect(0,0,100,100),
SetViewerFrame(ViewerHand,ViewerRect),
CoordList =   [ vl_point2D(30.0,20.0),
                vl_ point2D(60.0,40.0),
                vl_point2D(90.0,50.0),
```

```
                    vl_point2D(50.0,60.0) ],
        Polygon2DPro(DC,CoordList),
        ExitVisualib(),
% dlg_Note("Notice",
                "Getting ready to free gluevlib.dll library"),
        FreeLibrary(LibHandle).
% dlg_Note("Notice","Just finished.").
```

The code illustrates an important point: although the Visual Prolog Visual Programming Interface (VPI) redefines many domains—so that, for example, **WIN** appears as a domain in VPI predicates which have analogous roles to Windows API functions that would have used **HWND** variable. Despite these redefinitions, we can still mix 'winbind' methods and 'winbind' predicates with VPI methods and VPI predicates. Thus, for example, when creating a project that uses the Visual Programming Interface, we can still get the **HWND** for the active window, and pass that window handle to a 'winbind' predicate. Similarly, although the Visual Prolog's Visual Programming Interface makes no use of device contexts for its graphics predicates, we can still get create a device context and pass the handle to the device context we created to a 'winbind' predicate (used in a VPI project.) Or, although the Visual Programming Interface defines the domain **PICTURE** and demands that objects of that domain be passed to VPI predicates for bitmap handling, we can still use 'winbind' predicates that take or return handles to bitmap maps (**HDIB**s) in VPI projects and, what is more, we can even send those handles to predicates to predicates which the VPI has declared as belonging to the **PICTURE** domain. But to do this, we must be sure to include the path to the directories containing the "**windows.con**.," "**windows.dom**," and "**windows.pre**" files among the Application Expert's "include" directories, and the path to "**winbind.lib**" among the Application Experts "libaries" directories.

Make sure that Gluevlib import library is one of the modules loaded into the application's main Project Window (for the linker needs the import library in order to resolve predicate calls to **gluevlib.dl**l or Visualib functions within the **vlibtest.exe** programme), and that **gluevlib.dll** and **visualib.dll** are available either in the same directory as **vlibtest.exe** or in the \windows\system directory. Compile, run and enjoy. Three minutes to produce a graphics application, from creating the project to seeing the result on screen, using Visual Prolog's impressive code generation facilities and Visualib's high level graphics routines! It would have been unthinkable even a couple of years ago.