Data-Mapped User Interface Architecture for Visual Language Systems

Nita Goyal, Charles Hoch, Ravi Krishnamurthy, Brian Meckler, Michael Suckow

HP Labs, 1501 Page Mill Rd., Palo Alto, CA, 94304

"lastname" @hpl.hp.com

Abstract

Visual Language (VL) system development is getting increasingly sophisticated in part due to the arduous nature of user interface (UI) code development. This typically involves idiosyncratic calls to the windowing system that are intertwined with the rest of the logic of the VL system. To achieve an effective separation of the intertwined code, we take our cue from memory-mapped display for character based terminals and use a data-mapped architecture where all entities on the screen are mapped into data in the system. The computation of what is on the screen or changes to it can then be viewed as querying or updating this data respectively. We propose a declarative language, called HiD++, to implement this architecture that in addition to naturally expressing query and update computations, also addresses many other software engineering concerns. We have used HiD++ to implement a system for a visual langage called Rendering-By-Example (RBE) and report our experiences here. This work has been done in the context of Picture Programming project at H.P. Labs.

1. Introduction

Visual Language (VL) system development is getting increasingly sophisticated in part due to the arduous nature of user interface (UI) code development. This typically involves idiosyncratic calls to the windowing system that are intertwined with the rest of the logic of the VL system. Windowing systems were conceived to help build such ambitious user interfaces more easily. But they also increased the expectations of the end user so the problem has only become worse.

Most of the reasons for the difficulty can be attributed to tbe process of building UI code. Currently, UI code is developed using event based programming, wherein the events are generated by the windowing system and for each event the appropriate event handling code is written that is specific to the system that is being built. This event handling code can be summarized as follows:

> *This decision of what-to-do is based on the current content of the screen as well as the current state of the system as represented in the data structures of the program. The querying of what is on the screen is done using myriads of calls to the windowing system and the data structures are queried in the usual way. Once what needs to be done is decided, then it is achieved by calling the appropriate windowing methods, including the refresh, repaint, to reflect the changes to the screen.*

The resulting code is invariably very difficult to build and to maintain for any non-trivial UI.

Our approach mirrors the solution proposed in the context of memory-mapped displays. The use of memory mapped I/O for writing to and reading from the screen replaced the control sequence based I/O. The programming was simplified in this memory-mapped model, wherein a fixed location in the memory capapble of storing 24 lines of 80 characters (i.e., 24 * 80 bytes) were used to read from and write into the memory-mapped screen. As a result, writing to (reading from) screen was done by writing to (reading from) the appropriate location in the memory. This greatly simplified the screen I/O in the programs.

Extrapolating this approach, we map all the objects on the screen to data in the programs. Thus reading from and writing to the screen is replaced by the data-centric operations on the corresponding data. It is the responsibility of the "active" data to make the appropriate calls to the windowing methods to realize the necessary changes. As a result, the logic of what needs to be done in the event handling code can be summarized as follows:

> *The decision of what to do is based on the data representing the data mapped screen objects and the data structures; and the process of achieveing what needs to be done is to update the program data structres, including the data*

mapped screen objects. In other words, the event handling logic is a query dependent update, where the query computes the current state of the data and then updates the data as required.

Given this data centric view of the screen, we can program the logic of all the event handling code using a declarative language similar to QBE, SQL, Datalog, LDL etc., where the query dependent update is stated in a what-you-want manner as opposed to how-to-do-it manner in traditional programming. Experience from database system has shown that tremendous software-engineering advantage can be accrued by using a declarative language.

In this paper we propose such a language, called HiD++[1], based on higher order logic as an extension to such a language proposed in [KN88, KLK91]. The extensions needed are to provide object oriented features such as inheritance, encapsulation, and overloading as well as features to interoperate with the windowing systems. We have used this language to build a system for a visual language called Rendering By Example (RBE) [KZ95] that is a subset of the ICBE language [ZK95]. We report in this paper the advantages and problems of using such a data-mapped architecture and the declarative language for building this system. We compare it to the prior version of the system implemented in Microsoft Visual C++ that had about 25,000 lines of code.

We first describe in Section 2 the list of problems faced in building VL systems and cull the desiderata for the architecture. Then we propose the data-mapped architecture in Section 3. In Section 4 we describe the declarative language HiD++. Finally, in Section 5 we recount our experience in building the system and how this architecture and language address the problems enumerated in Section 2.

2. Problems in building VL systems

Building a system that implements any nontrivial visual language poses numerous problems in the development phase that ultimately limit the scope of the language. In this section we enumerate some of these problems briefly[2], and state the design goals for overcoming these problems:

1. **Evolution problem**: As the visual languages are very young when compared to the textual languages, the user community for VLs is still evolving and making new demands on the languages . The technology is also evolving continuously; not only through the set of building blocks (e.g., widgets) of the VL and the interactions between these building blocks, but also through innovative gestures and modalities. As the VL systems have to continuously evolve too in response to the needs of the VL users, the software sooner or later becomes immalleable, that is, it becomes very hard to make any incremental improvements thus limiting the scope of the language.

   Goal: Have a development tool that makes the software flexible and easily extendible.

2. **GUI programming problem:** Developing graphical user interface code is problematic because the windowing system calls are idiosyncratic and platform-specific; often there are specialized library calls to accomplish simple tasks. The windowing systems have specific protocols for interaction that often infiltrate even into platform-independent code. This complicates the VL system building process since this GUI code is interspersed with the logic of the VL system itself and makes it hard to debug the software or to make incremental changes.

   Goal: Separate GUI code from the rest of the VL system.

3. **Performance assurance problem:** Though it is known that the display phases such as repainting and refreshing of most windowing systems are slow, the performance bottlenecks in VL systems are still largely unknown. Therefore, the steps taken to assure good performance often need to be modified as more insight is gained. This can be hard as the performance driven decisions are usually implicit in the data structures and procedures, and making any changes might require a major redesign of the system.

   Goal: Initial system development without concern for performance with the assurance of "hot-spot" refinement later when performance bottlenecks are identified.

4. **Compilation problem:** Traditional textual languages have linear syntax and therefore limited context sensitiveness. Thus the context free grammars worked

[1] HiD++ stands for High-level Data language and the ++ denotes the OO features in the lanuage. This is pronounced as Heidi-plus-plus.

[2] We do not claim primary authorship of these problems as most of these have been observed by many researchers in this area as early as two decades ago in building the Query-By-Example system. The purpose of this enumeration is to categorize them so as to refer to them later in this paper. Therefore, we make no attempt to ascribe the proper credit of authorship.

reasonably well for these languages. Visual languages, on the other hand, have much more context sensitive behavior. This rules out the use of context free grammars and the use of traditional tools to build the compiler.

Furthermore, visual programs are constructed in an incremental fashion. This requires that the compilation of these programs be incremental. Traditional tools such as Yacc/Lex do not address incremental compilation very well.

Goal: To provide the capability to parse and translate context-sensitive languages incrementally.

5. **Save/load problem**: Saving and loading refers to making the VL programs persistent; saving is storing the program on a permanent medium (*e.g.*, disk) and loading is reconstruction of the relevant structures for the program in memory from the storage medium where it was saved. Saving and loading for textual programs is very simple but is nontrivial for VL programs. Since the language building blocks have complex structures and are interrelated, save/load requires that the references should be mapped correctly. Also, if the VL system is evolving, a new version of the save/load mechanism must ensure that it is compatible with the previous versions. Most VL sytems currently do not address software engineering amenities such as version control, source control, or concurrency control with guaranteed recoverability.

   Goal: To relegate the save/load functionality to the system.

In developing the system for the RBE language [KZ95], we were motivated by these goals to solve the aforementioned problems. Towards this end we have developed a data-mapped architecture of the system and a declarative language that is used as a tool to implement this architecture. We describe the data-mapped architecture in section 3 and the declarative language in section 4. In section 5, we revisit the problems of building VL systems and dicuss how the proposed architecture addresses these problems.

3. Data-Mapped Architecture

In this section we propose a data-mapped architecture for building a VL system. *Data-mapped* means that all entities on the screen are mapped to data in the system and the computation of what is on the screen or changes to it can be viewed as querying or updating this data, respectively. The data-mapped architecture is motivated by the fact that event-based programming, which has become a standard in visual systems, is naturally represented in the data-mapped paradigm.

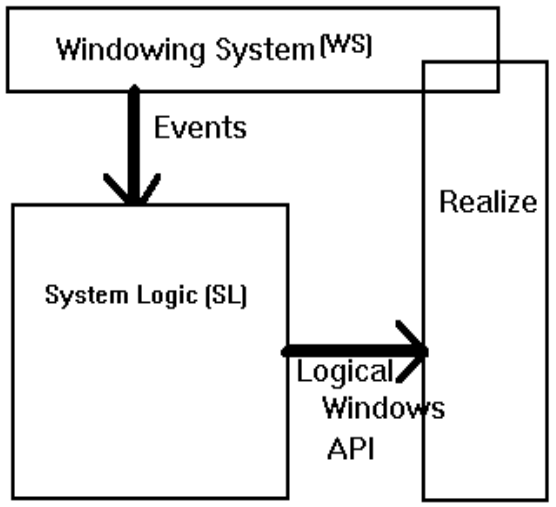

Figure 1: Architecture of the System

Event based programming refers to the process by which an event, *i.e.*, an action on the screen by the user, is captured by the windowing system which in turn calls the appropriate event handling code that determines what is to be done in response to that event. This means that most of the code in the VL system is event handling code, either called directly by the windowing system or indirectly by other procedures that are eventually called from the windowing system.

The proposed data-mapped architecture is shown in Figure 1. It consists of 3 main modules, the Windowing System (WS), the System Logic (SL) and the Realize module. The events are captured from the screen by the WS which calls the appropriate event handling code in SL. The System Logic computes the required changes to the screen and the Realize layer interacts with the WS to effect these changes on the screen.

SL has a data-mapped view of the screen which is utilized very effectively by the event handling code. This code consists of the following main steps:

1. Getting information on the current screen state to test the preconditions of the event, as a query on the data in the SL.

2. Computing the necessary action on the screen based on the current screen state, as update to the data in SL.

3. Computing the new screen state, as a fixed-point computation on the data in SL.

Let us consider a simple example to illustrate the architecture:

Figure 2: Click Event on a Widget

Most widgets in focus can have "handles" -- 8 small black squares one on each corner and edge -- that can be used for resizing or dragging the object. Consider a textbox widget as in Figure 2 and a "Click" event on it. The desired effect is that the widget must get the focus and therefore the handles must appear around it.

Without a data-mapped architecture, the UI code for this scenario will have explicit calls to determine whether to place the handles around the textbox, and if so, their positions. Now if a new widget is introduced to the system, the Click event for that widget also needs to make such calls. This also holds true for all objects other than handles that are affected by the Click event.

In the data-mapped architecture, we have a complex data object instance for each handle on the screen that stores its visual properties such as size, position, color etc. We also have data representing the widgets on the screen. When the Click event is captured by the windowing system it is passed on to the SL. The event-handling code in SL computes the state of the system and determines that the necessary action is to give focus to this widget. Therefore it updates its data corresponding to the "focus object" to now point to this widget. SL then computes the new state of the system which will automatically compute the positions of the handles and put them around the focus object. In response to the updated data, the Realize layer will update the screen to reflect the change in the handle positions. The Click event handling code for the widget does not have any reference to the handles; therefore, any new widget that is introduced does not need to bother with making explicit calls to the code for Handles.

Note that the event handling code in SL has been completely separated from the GUI code which is now limited only to the interactions of SL with the WS and Realize. The effect of the data-mapped architecture is that any operations on the screen are operations on the data and vice versa.

Also note that since SL just performs computations on its data as query dependent updates, these computations can be stated declaratively; *i.e.*, state the queries on the data and the subsequent updates in a what-you-want manner using a declarative language as opposed to the how-to-do-it manner used in most traditional programming languages. The advantage of using a declarative language is that it provides a data structure independent way to write the programs, thereby addressing the performance assurance problem. We will elaborate in sections 4 and 5 how the performance can be assured after the event handling code in SL has been written and debugged without initially being concerned about performance.

Examples of declarative languages with useful queries and updates on data are QBE and SQL. Datalog, LDL and Prolog [Ull89, NT89] generalize this declarative programming to computation beyond relational algebra. But they consider only flat relations as data, whereas in VL systems we typically have complex data. In [KN88, KLK91], a complex data language based on higher order logic was proposed. In Section 4, we describe a modified version of that language that we used in our development process.

4. Declarative Language HiD++

The conceptual structure of the language proposed in [KN88, KLK91] is based on objects and expressions on objects. An object can be classified into one of three categories: an atomic object, a tuple of objects, or a set of objects. Examples of atomic objects are integers, characters, etc. A tuple object is recursively defined as a collection of attribute/objects pairs, in which the object corresponding to any attribute can be atomic, tuple or set object. A set object is a collection of atomic, tuple or set objects. For example, {(name:john, sal:10K),...} is a set object whose elements are tuple objects. The lanugage also allows lists and arrays similar to the sets but for brevity we omit that discussion here.

Elements of a tuple can be accessed by reference to the unique attributes, serving to ``name" objects, whereas a set is queried by the contents of the object alone. Attributes of a tuple or an element of a set can be either extensionally specified to be a particular object or can be intentionally specified by a rule as in any Horn clause language (*e.g.*, Datalog, LDL [Ull89]). A rule is similar to a query as in QBE or SQL with the addition that the condition in the query can refer back to the attribute that is being defined to form recursive rules. Explicitly specified attributes are called *base (facts)* and intensionally specified attributes are called *derived*. Thus, in C++ terminology, base facts correspond to data members and derived facts correspond to const (*i.e.*, no side effects/updates) methods.

Data Declaration

```
Class Universe
 Base:[  focusObj:Visual*`.
         static theUniverse:Universe*`.
         handles:{Handle*}`.                    ]
endClass

Class Visual
  Foreign:[ virtual top:COORD `.
            virtual left:COORD `.
            virtual width:COORD `.
            virtual height:COORD `.
            virtual visible:Boolean `.    ]
  Base:[  canhavehandles:Boolean `.  ]
  Procedure: [ virtual Click()`.  ]
endClass

Class Handle isa {  public Visual }
  Base:[            side:HandleSide `.
          static handle_height :=COORD(96.0)`.
          static handle_width :=COORD (96.0)`.   ]
  Derived:[  static onobject:Visual `.
               virtual visible:Boolean `.
               virtual top:COORD  `.
               virtual left:COORD  `.                    ]
endClass
```

Figure 3: Example of HiD++ Program

The derived facts (methods) can be overloaded, inherited and redefined, as in most OO languages, based on the inheritance hierarchy. A class, similar to C++ classes, is declared to be a set of tuple instances along with all its parent classes in the inheritance hierarchy. Therefore, each object instance has attributes (*i.e*, methods & data members) that are either inherited from its parent classes or are defined in the class of the object. Furthermore, each class provides the usual encapsulation features such as visibility and protection. As a result, a rule can only use attributes that adhere to the encapsulation restrictions.

The declaration of the program corresponding to Handles example is in Figure 3. Following is a rule to illustrate derived facts (rules are denoted by left arrow :-). Handle is a class (tuple object) with a derived atomic attribute "onobject", the widget on which the handle must appear. It depends on two base facts; the object that has the focus (focusObj) and whether this object is allowed to have handles or not (canhavehandles).

```
Handle::onobject = O:Visual  :-
        Universe::theUniverse = U:Universe &
        if  U.focusObj = NULL  then  O = NULL
        else U.focusObj = FO:Visual &
             if FO.canhavehandles = $bool_true:Boolean
            then  O = FO  else O = NULL
            fi
        fi `.
```

Because we need updates in our architecture and also to model procedural actions, we allow attributes to also be procedures that can make updates to the base facts. Intuitively, these can be viewed as query dependent update that are allowed in QBE or SQL. The approach taken here closely mirrors the LDL proposal [NT89, KLK91]. We allow recursive procedures that have a top down semantics similar to production rules and the updates must respect the encapsulation restrictions. These procedures can call external C procedures as well with the usual procedural semantics.

We state all the event handling code in the System Logic as procedures in some class. These procedures query the current data and update the base facts. Going back to the Handles example, a click on a widget not previously in focus only needs to update the base fact "focusObj" to now be this new widget. The Click procedure (procedures are denoted by right arrow -:) is as follows:

```
V.Visual::Click()  -:
        (Universe::theUniverse).focusObj  := V  `.
```

The atomic update is denoted by the := symbol. Once the Click procedure has updated the base fact, the system must ensure that the resulting changes to the screen are propagated through the Realize layer. For instance, now the handles must appear around this focusObj. The mechanism used in the language to propagate object properties that are manifested on the screen is that of *foreign* attributes.

The entities on the screen are mapped to data objects in the program; therefore, we need to map visual properties of an entity to attributes of the data object. This is done by declaring these properties as foreign attributes which can be base or derived facts. Since the operations on these properties are to query them for information or update them through the Realize layer, we allow procedures for accessing and updating the foreign base attributes. These procedures will typically make Realize calls which in turn will make windowing API calls to achieve the desired effect. For example, for a visual widget on the screen, the property whether it is visible or not is expressed as a foreign attribute. The accessor and updator procedures are respectively:

```
V.Visual::visible = B:Boolean  -:
        V.GetVisibleFromWindows() `.

V.Visual::visible := B:Boolean  -:
        V.SetVisibleInWindows() `.
```

Intuitively, these accessor and updator foreign procedures redefine the system default accessor and updators for atomic, tuple and set objects that are foreign (Visual::visible attribute in the example). For foreign attributes that are set objects, there will be two updator procedures; one, for inserting into the set, second, to delete from the set. For foreign attributes that can be derived in System Logic, it is possible to just write a rule as for a derived fact. For instance, in the Handle class that inherits from Visual class, Handle::visible foreign attribute can be derived based on the object that has the handles:

```
H.Handle::visible = B:Boolean  :-
        Handle::onobject = O:Visual &
        if O<>NULL  then B = $bool_true:Boolean
        else B = $bool_false:Boolean
        fi `.
```

Any change in the data in the body of the rule (such as "onobject") will automatically trigger the computation of the new value of Handle::visible and since this is a foreign attribute in Visual, this new value will be realized on the screen. As a result, the usage of these foreign attributes is indistinguishable from any other base or derived data except that for the derived foreign attribute, the system treats it as if it were a materialized view. (We elaborate on materialized views in section 5).

The use of foreign attributes to reflect the visual properties of objects on the screen in the data in SL is the essence of the data-mapped architecture. It facilitates the clean separation of SL code from the GUI code and ensures that the internal data is a true proxy for the screen. It also localizes the dependencies among objects and attributes which makes code development much easier and thus helps in addressing the evolution problem of UI software.

For instance, the handle code works for all widgets that can have handles. It automatically decides which widget must have the handles and which should not, and where they must be positioned etc. Now, if a new widget is added that also can have handles, we can develop the code for that widget without any concern for when the handles should appear around it and how they must be positioned. In contrast, without the foreign attributes being screen proxy, we would have had to actively decide for every new widget and event when the handles should appear and be erased. These kind of dependencies multiply fast and ultimately make the software immalleable since any incremental addition to the code must also make sure that all the dependencies have been handled. The data-mapped architecture does away with this nonlinear effect of dependencies.

In the next section, we discuss our experience using HiD++ to develop ICBE system at HP Labs.

5. Experience in using HiD++

We are currently using HiD++ to develop the ICBE system. First we implemented part of ICBE system using Visual C++, but soon were convinced of the futility of the approach. Then we designed HiD++ and built a compiler that generates C++ code for given HiD++ programs. Using HiD++ we have recoded a significant part of ICBE that is estimated to be more than 25,000 lines of C++ code in the previous (C++) version of the system. In contrast, the HiD++ source code is about 5000 lines and the generates about 34,000 lines of C++ code.

We have yet to observe any major performance bottlenecks in the ICBE system written in HiD++, other than the anticipated view maintenance problem described in subsection 5.4. Running on a 90MhZ Pentium, the degradation in performance is below the threshold of human perception for most (if not all) of the interactions. Even though the performance of this system is yet to be put to critical and comprehensive tests, our experience thus far has corroborated our design assumption of 80/20 rule; *i.e*, all but a small portion of the system would not have any performance problems and these few problems are expected to be alleviated using performance assurance capabilities of the language.

In the rest of this section we discuss the capabilities of the language to deal with the problems enumerated in section 2.

5.1. Compiling capabilities

Koster proposed a Turing complete language called Affix Grammars in the context of Algol 68 [K71]. Affix grammars are context free grammars with the following additions:

1. predicates called affixes are allowed in the body of the grammar rules and they have to evaluate to true for a rule to be applied in a derivation tree;

2. affixes can read and write into the global symbol table that provides the global context.

Intuitively, affixes can provide the semantic checks and there is no restriction on how affixes are written as long as they evaluate to true or false. Koster also showed that affix

grammars have a top-down parser. The implication is that if the grammar is constructed carefully and the look ahead logic in the form of affixes are chosen appropriately to disambiguate multiple rule applicability, then a recursive descent parser can be generated semi-mechanically.

This recursive descent parser is constructed as follows:

- Each nonterminal is a HiD++ procedure;
- Each terminal is an accept routine;
- Each affix is a derived view over the data, resulting in true or false.

Note that if the intent is to do syntax directed translation, then the body of these procedures can have update statements to reflect the changes to the data. The main reason for generating the HiD++ program semi-mechanically from the affix grammar is that HiD++ procedures are similar to production rules with top-down semantics.

Thus, HiD++ is, by design, suited for writing a recursive descent parsing/translation and Turing complete property of affix grammars assures that context sensitive semantics can be handled. Further, because it is recursive descent parsing, it is also suited for incremental parsing and translation.

Finally, the use of classes and subclasses, that represent and mimic the conceptual program, allows the HiD++ program to reason over the program as rules (i.e., derived views). In contrast, traditional parse trees were representative of the way the input program was parsed more than the conceptual program itself.

Our experience using HiD++ for ICBE has been very promising. In fact, we were so convinced of this capability that we designed and boot-strapped the compiler for HiD++ in HiD++. This compiler generates the C++ code for HiD++ programs and all but the code generator was implemented in HiD++. The code for the rules and procedures is less than 10 pages.

5.2. Save & Load Capabilities

Saving and loading refers to making the VL programs persistent. Providing persistence to a program is to write to disk the appropriate internal data structures of the program with proper naming capability for subsequent retrieval by name.

Providing persistence for the data is a well understood topic in database systems and we do not dwell on this aspect of the problem. But we do want to emphasize *that the HiD++ system provides the capability to save and load any program (i.e., the corresponding data structures) without any knowledge of the visual language that it represents.* Thus, relieving the implementer of the visual language system from the responsibility to explicitly program this capability.

Naming and retrieval of a program by name is also quite straightforward as this can be viewed as indexed access of the program, where the index is itself data to be kept persistent.

Given that save and load has been identified as the problem of keeping persistent the data corresponding to the program, then extending this to versioning of programs for multiusers is another well studied database problem. This concerns the versioning of data to deal with multiple versions of the program. The multiuser capability requires the concurrency control and recovery capability of DBMS to ensure correct access to data (i.e., program). Note once again, these capabilities can be provided without any knowledge of the visual language.

5.3. Performance Assurance capabilities

There are three ways to improve performance of a HiD++ program. These are:

1. a derived rule is materialized so that it is not recomputed many times;
2. an aggregate data access is improved through the use of different data structures or auxiliary data structures (e.g., indices);
3. a rule or procedure is implemented using a more efficient algorithm.

It is argued here that all of the above transformations can be retrofitted to a VL system written in HiD++ as easily as if it were done originally.

Use of the materialized derived view instead of computing the derived view every time it is needed is done by redefining the accessor functions for the derived view to access the materialized view. The redefined accessor function will be automatically used wherever the derived view is used. That is, none of the rules and procedures that use this derived view need to be changed.

The use of different storage structures or indices for aggregate data was the key performance improvement tool used in database technology. This was possible because the database languages are declarative and data structure independent. As we have the same advantages, we can apply the same methods here.

If there is a need to compute a rule or procedure in a more efficient manner than HiD++ would execute, then it is possible to replace that rule/procedure by a program written in an imperative language. Such replacement is straightforward to incorporate given that ultimately the generated code is C++ and it can be linked in that stage. Note once again that none of the rules and procedures that use this rule/procedure need to be changed.

In summary, we have shown that the above techniques of *performance improvement can be done after the identification of performance bottlenecks, as hot-spot refinement.* This divides the problem of performance assurance to first assuring correctness and then addressing the efficiency.

We do not imply that all conceivable performance improvements can be done here. Based on our experience though, we feel that the above list of improvements should be sufficient.

5.4. UI manipulation capabilities

Manipulating objects on the screen through data mapped objects in the program simplifies the system design in many ways that were discussed in the previous two sections. So we do not dwell on this topic here.

One observation with respect to interaction with the windowing system needs mention. Foreign attributes such as Handle::visible that are defined as a derived view need to be reflected to the windowing system whenever the derived view changes. Naive implementation of this is to reflect all values irrespective of whether it changed or not. This was expected to be a performance bottleneck and therefore our design called for implementing a "dirty data" management scheme. Currently, we have not implemented it yet. As a result we use the naive method to reflect the values onto the windowing system. Therefore, we currently observe perceptive sluggishness whenever there is a need for many refreshes such as in the case of drag-drop. We are reasonably convinced that the dirty data management scheme will address this problem because this problem stems from excessive useless work (i.e., calls to windowing system).

However, if the ordering of calls to windowing system turns out to be important to alleviate the performance bottlenecks, then we will need to use procedural code to reflect the data. This will be similar to the intertwined UI code in the traditional approach. We have not yet observed the need for this; our hope is that they will be few and far between and therefore will not be a major problem.

6. Conclusion

We discussed the common problems faced in building visual language systems and proposed a data-mapped architecture to address these problems. We described this architecture that expresses objects on the screen as data in the system and reduces event handling code to queries and updates on this data. Since query dependent updates are easily expressible through declarative languages, we also proposed a declarative language for complex objects that is a powerful tool for building a VL system with the data-mapped architecture. We finally reported our experience in building such a nontrivial system for the RBE language as part of the Picture Programming project at H.P. Labs.

References

K71 Koster, C.H.A., "Affix Grammars", ALGOL 68 Implementation, Elsevier North-Holland (J.E.L. Peck (ed.), Amsterdam, 1971) pp 95-109.

KLK91 Krishnamurthy, R, W. Litwin, and W.Kent, "Language Features for Interoperability of Databases with Schematic Discrepancies", Int'l conf. of SIGMOD, 1991, Denver.

KN88 Krishnamurthy, R, S. Naqvi, "Towards a Real Horn Clause Language", Proc. of VLDB, Los Angeles, 1988.

KZ95 Krishnamurthy, R., M.Zloof, "Rendering By Example (RBE)", Int'l conf on Data Engineering, Taipei, Taiwan, 1995

NT89 Naqvi, S., S. Tsur, "A Language for Data and Knowledge Bases", W. H. Freeman, 1989.

Ull89 Ullman, J. D., "Database and Knowledge-Base Systems", Vol. II, Computer Science Press, 1989.

ZK95 Zloof, M., R. Krishnamurthy, "ICBE: Interoperation and Customization By Example", In preparation, 1995.