AN ABSTRACT OF THE THESIS OF

John Wesley Atwood, Jr. for the degree of Master of Science in Computer Science

presented on January 15, 1996.

Title: Culprit Tracking: Improved Lazy Marking for Better GUI Performance.

Abstract approved: Redacted for Privacy

Margaret M. Burnett

Culprit Tracking is a technique to make lazy evaluation in a programming language even lazier. We sought to develop such a technique after noting poorly-distributed performance characteristics of graphical user interfaces (GUIs) programmed in lazy languages. A characteristic aspect of GUI programs is the intensive screen I/O. These programs are generally highly interactive and very visually oriented. We noted that significant computation time can be spent to maintain values of cells that either do not contribute to the output, or cannot possibly have changed at the given time step. We sought a pay-as-you-go implementation technique that would allow users to better specify which values they were interested in and only pay when those values could possibly change. Our breakthrough came when we made the observation that the mouse can only be at one location on the screen at any one time. When a user event occurs, it occurs at one and only one location on the screen; the system can therefore safely assume that other locations on the screen received no new event. This seemingly obvious fact allowed us to arrive at a new implementation technique we call culprit tracking.

Culprit tracking combines the desirable properties of two other techniques, eager evaluation and lazy marking, to achieve our stated cost distribution requirement that the cost of executing a program should be distributed such that the user pays for computing currently active values that are of interest to the user, and not for computing inactive values or values not of interest to the user. It is the first such technique to do so.

Culprit Tracking:
Improved Lazy Marking for Better GUI Performance

by
John Wesley Atwood, Jr.

A THESIS
submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed January 15, 1996
Commencement June 1996

Master of Science thesis of John Wesley Atwood, Jr. presented on January 15, 1996

APPROVED:

Redacted for Privacy

Major Professor, representing Department of Computer Science

Redacted for Privacy

Chair of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

John Wesley Atwood, Jr., Author

ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis advisor, Dr. Margaret Burnett. Her high standards, unflagging enthusiasm, and patience helped me immeasurably. Thanks for the untold hours you have invested in my behalf.

I would also like to thank my parents, for their constant support, and for teaching me how to learn. And finally, I thank my family, Christine, Wesley, and Andrew, for making it all worthwhile. Wesley, Andrew, wanna wrestle?

Table of Contents

Page

Table of Contents (cont'd)

Page

List of Figures

Page

Culprit Tracking:
Improved Lazy Marking for Better GUI Performance

1. Introduction

Culprit tracking is a technique to make lazy evaluation in a programming language even lazier. We sought to develop such a technique after noting poorly-distributed performance characteristics of graphical user interfaces (GUIs) programmed in lazy languages. A characteristic aspect of GUI programs is the intensive screen I/O. These programs are generally highly interactive and very visually oriented. We noted that significant computation time can be spent to maintain values of cells that either do not contribute to the output, or cannot possibly have changed at the given time step. We sought a pay-as-you-go implementation technique that would allow users to better specify which values they were interested in and only pay when those values could possibly change. Our breakthrough came when we made the observation that the mouse can only be at one location on the screen at any one time. When a user event occurs, it occurs at one and only one location on the screen; the system can therefore safely assume that other locations on the screen received no new event. This seemingly obvious fact allowed us to arrive at a new implementation technique we call culprit tracking.

1.1. Organization of this Thesis

We begin with an example to illustrate the issues that we are addressing. We then provide some background information and review related work in Chapters 2 and 3. We describe culprit tracking in Chapter 4, followed by a proof of the technique's correctness and an analysis of its complexity. In Chapter 5, we give the results of our empirical validation. We discuss future work in Chapter 6, and conclude in Chapter 7.

1.2. An Example

The program shown in Figure 1-1 consists of three windows or forms: an input form and two output forms, a digital readout and a bar graph simulating the mercury column of a

traditional indicating thermometer. The user may be uninterested in one or both of the displays. She may indicate this lack of interest by moving a form off the screen, and may reasonably expect the program to execute faster now that it is producing less output. As we will show, other techniques violate this expectation, while culprit tracking meets it. Culprit tracking allows the performance of the system to better reflect her interests. She need not pay for calculations she won't use.

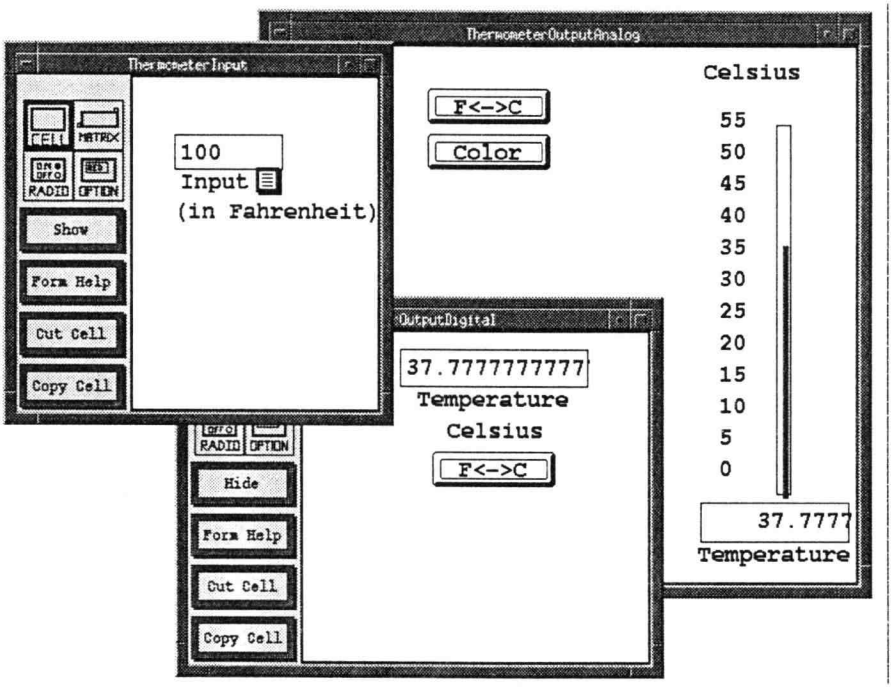


Figure 1-1. A Thermometer program.

1.3. The Cost Distribution Requirement

Our goal is a more responsive programming environment. By allowing the user to specify an interest in some cells and a lack of interest in others, and therefore redefine what values constitute the output of a program, the system can execute programs more efficiently. The system must only recompute values for cells that are of interest, and only recompute cells whose values change. We will show that some techniques recompute values of cells that

are not of interest to the user, and some techniques recompute values that cannot have changed. We therefore state this cost distribution requirement:

The cost of executing a program should be distributed such that the user pays for computing currently active values that are of interest to the user, and not for computing inactive values or values not of interest to the user. We will show that CT is the first approach that meets this requirement.

2. Background in Forms/3

While culprit tracking could be implemented in any lazy declarative language, we have implemented it in the Forms/3 language. To provide context for what follows, we give a brief overview of the Forms/3 language [Burnett 1991, Burnett and Ambler 1994], emphasizing the aspects that are relevant to this work. Forms/3 is a lazy, declarative, visual language.

2.1. A Brief Introduction to Visual Programming Languages

Visual programming is an area of programming language research whose proponents seek to advance programming languages by removing the constraint that all semantically significant notations must be a one-dimensional string of text. Shu [Shu 1988] defines visual programming languages as languages that use "some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language," and states that "in order to be considered a visual programming language, *the language itself* must employ some meaningful (i.e., not merely decorative) visual expressions as a means of programming." These visual expressions include graphics, diagrams, spatial relationships, etc. In all aspects of life, people use such notations as an aid to reasoning about problems and researchers in visual programming languages (VPLs) seek to allow and facilitate the use of these notations as semantically meaningful parts of programs.

Visual programming presents three opportunities that we feel are of particular relevance to GUI programming: the notions of concreteness, responsiveness, and domain mapping.

Concreteness is a property of a programming system where the programmer can specify and examine concrete, specific values, rather than abstract parameters, while programming. A system supports concreteness by allowing the programmer to use sample values in a program, rather than having the programmer specify abstract parameters whose values are known only at runtime. Concreteness can be used to specify a program, such as by

sketching a user interface, and can also be combined with responsiveness to provide immediate feedback and testing as a program is created.

In a responsive VPL, the program executes whenever a program change is made or an input is given. Tanimoto [Tanimoto 1990] identified four levels of responsiveness (he calls it *liveness*). At level 1, visual expressions are meaningful to the programmer but not the computer, as in flowcharts and ASCII line art that appears as commentary in textual code. At level 2, visual expressions are meaningful to both the human and the computer, as in executable flowcharts. At level 3, the computer responds immediately to any change to the program, without requiring the programmer to request execution, as in spreadsheets. At level 4, the computer responds continuously, without explicit changes by the programmer, such as in response to non-user input (a system clock, A/D sensors, etc.). In a responsive language, the distinction between the language and the support environment is blurred. The traditional edit/compile/execute cycle is collapsed to the act of making a change and examining the result. Interpreted languages such as APL and Lisp, and incrementally compiled languages such as Turbo Pascal were first steps in the trend toward more responsiveness, and later languages such as Prograph, VPL, and Forms/3 achieve even more responsiveness by incrementally executing a program whenever the programmer makes a change.

A third concept of interest is that of domain matching. If language designers can aid the programmer in thinking about the problem is terms of the problem domain, the programmer can deal with fewer terms and concepts. In the realm of GUI programming, specifying a program by laying out windows seems a closer domain match than specifying a window textually. This is not news, as evidenced by the many GUI-building products which use the word "visual". Most of these products, however, are based on textual imperative languages; while our goal it to explore the use of a declarative, visual approach.

2.2. Form-Based Programming with Temporal Assignment

Forms/3 extends the form-based, or spreadsheet, model of visual programming. In Forms/3, a program consists entirely of cells, each of which has a formula. There is no notion of control flow as there is in imperative languages such as C and in some object oriented languages such as C++ and Smalltalk. Instead, Forms/3 adopts a declarative model; each cell's formula completely determines the value of that cell.

Forms/3 implements a type of single assignment termed *temporal assignment.* A cell's value is not a single data item, but a (conceptually) infinite vector of data items along a time line. This time line is termed *logical time,* and a cell's value is termed a *temporal vector* (TV). A cell's temporal vector is similar to a *stream*, but there is no notion of consuming values. Forms/3 introduces this notion of logical time in order to support time-oriented computations (such as the notion of a clock's values or values of an event stream) while still preserving the declarativeness of the language. Temporal assignment enables referential transparency.

A value in a TV may be followed by a new value in the TV at a later logical time, but remains accessible to the user for viewing and for access via the time-based operators (earlier, prev, and fby). That TVs are infinite in concept poses no real difficulty because the vector is populated with single data items only upon demand. The user can, as an aid to debugging, view prior values of cells by manipulating his position on the logical timeline, can move time backwards and forwards via a timeSlider scrollbar on the screen. Figure 2-1 shows some cells of the thermometer program and the temporal vectors of those cells.

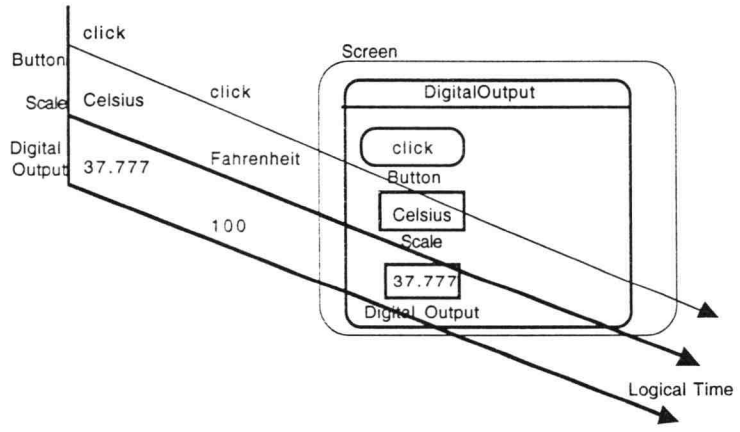


Figure 2-1. Temporal Vectors of 3 Cells in the Thermometer Example.

2.3. Event Receptors - Events are Values

In Forms/3, *user events* such as mouse clicks and keyboard keypresses are values. These values are of type *Event,* just as other values are of type integer, real, string, etc. The mapping of events into values is accomplished via instances of the eventReceptor type, defined on a Forms/3 primitive form. To make areas of the screen sensitive to mouse and keyboard events, the user copies the eventReceptor form and tailors it to his program. The temporal vectors of event receptors differ from those of other cells only in that the argument to the eventReceptor formula is the sequence of user events, and future values of that sequence cannot be known until the progression of time allows the system to discover those values. The system can treat the temporal vector of an event receptor identically to those of other cells because it only references nodes of a temporal vector at logical times less than or equal to the latest logical time.

2.4. Memoization

Because time can be moved backwards, any node in a cell's temporal vector of values may be re-displayed. Values (except for events) could be recomputed, by the declarative, referentially transparent nature of Forms/3, but instead they are memoized for performance improvement. The current Forms/3 implementation maintains a doubly linked list for each cell, with O(1) entry points at the first, last, and most recently accessed value in the cell's TV. The number of values saved could be reduced via LRU or other heuristics, but this would not change the evaluation model or the semantics of Forms/3 because of the property of referential transparency.

The combination of TVs and lazy evaluation with memoization introduces issue of how the system can determine, at a given logical time, the correct value of a cell. Forms/3 handles this by *marking* values in a TV with the value's time of definition, time of expiration, and time of termination. A mark for a value at a given time indicates that the value has not yet been memoized and therefore must be computed.

3. Related Work

We first discuss 4 prior techniques of evaluation and then survey related systems, noting, when known, which of these techniques the system uses.

3.1. Prior Techniques of Evaluation

The 4 prior techniques are: eager evaluation, lazy evaluation with no saved values, lazy evaluation with values saved and eagerly marked out-of-date, lazy evaluation with saved values, lazily marked out-of-date, and lazy evaluation with saved values lazily marked also with culprit tracking. Throughout the remainder of this paper, we refer to these techniques by these abbreviations:

EE:	Eager Evaluation
NM:	Lazy evaluation with No Memoization and therefore no marking
EM:	Lazy evaluation with Eager Marking
LM:	Lazy evaluation with Lazy Marking
CT:	Lazy marking with Culprit Tracking

3.1.1. Eager Evaluation

Eager evaluation is the most straightforward evaluation technique. Under EE, a change in the value of a cell (such as the user clicking on the F-C button) is propagated to all cells that depend on (are clients of) this cell. However, EE fails our cost distribution requirement because values off screen may be recomputed, even though the user is not interested in these values. In the Thermometer example, EE recomputes the analog display form even if it is off the screen.

3.1.2. Lazy Evaluation

An alternative which has gained prominence in the functional languages community is call by need, or lazy evaluation. Lazy evaluation is an implementation of normal order evaluation, in which an expression is evaluated only when that value is needed. Because

the order of evaluation is not explicitly specified by the programmer, any system using lazy evaluation must disallow side effects and maintain referential transparency.

3.1.2.1. Lazy Evaluation with No Memoization

A naive implementation of lazy evaluation might simply compute the value of an expression whenever that value is needed. Such an approach would be needlessly inefficient, however, because the system can, due to referential transparency, save computed values so that they need not be recomputed. In the thermometer example, this technique, NM, would recompute all cells in the program at every time step because no values are saved (and even at the same time step if the user iconifies and displays a window.

3.1.2.2. Lazy Evaluation with Eager Marking

To take advantage of referential transparency, a system can memoize: save computed values, along with the parameters that produced each value, so that needless recomputation is avoided. However, introducing saved values into a language with time-based sequences of values also introduces the danger that the system may use an out-of-date value. To guarantee that the correct value is used, the system needs to track which values are current and which are out of date. An approach that accomplishes this is lazy evaluation with eager marking (EM). The system computes lazily, but eagerly marks out-of-date the subgraph of all clients of changed values in the program graph. This is different from eager evaluation because EM eliminates computation of undemanded values. In the thermometer example, EM would recompute only on-screen cells, but eagerly mark any changed clients, even if they don't contribute to the output.

3.1.2.3. Lazy Evaluation with Lazy Marking

The original implementation of lazy marking in Forms/3 avoids traversing the client subgraph by placing an expiration time on each value it computes. Often this expiration time is the next logical time, but a computation may also be valid over a longer range of time, as with constants or an event receptor cell (and its clients) where clicks have

happened sparsely. The LM algorithm is given here:

```
Demand(x)
If an unexpired value for x exists, then
    return it and its expiration time.
Else if x's formula is a constant,
    return it, and an expiration time of
        infinity.
Else
    For each direct supplier of x:
        Demand the direct supplier.
    Perform the operation.
    Save the result in Answer.
    Save the minimum of the suppliers
        expiration times in Expiration time.
    Place a mark for x at Expiration Time.
    Return Answer and Expiration time.
```

```
Render-screen
For each form on-screen:
    For each cell on the form:
        If expiration time <= current time,
            re-display(cell).
```

Figure 3-1. The Demand Function for the LM algorithm.

LM avoids traversals for marking, and adds only a constant, O(1), to the cost of a single demand. However, in rendering the screen, it must "poll" all on-screen values at every time step to ensure that correct values are displayed, and therefore, may demand cells whose values are already current.

3.2. Other Systems

There has been a great deal of work related to declarative GUIs in declarative visual languages and sublanguages. Garnet [Myers et al. 1990] is a well-known visual/textual GUI builder that includes declarative visual sublanguages. Garnet's evaluation strategy uses constraint satisfaction via lazy evaluation and eager marking (discussed in section 3.1.2.2). NoPumpG II [Wilde and Lewis 1990] is a declarative visual programming language for graphical I/O based upon the spreadsheet metaphor. Like commercial spreadsheets, it uses eager evaluation (and therefore no marking) to propagate changes. Most declarative visual languages intended for the domain-specific task of GUI building, including Fabrik [Ingalls et al. 1988], ConMan [Haeberli 1988], and InterCONS [Smith 1988], also use variations on eager evaluation. VPL [Lau-Kee et al. 1991], a general-

purpose declarative visual programming language, uses lazy evaluation with no memoization of prior values (and thus no marking).

The Fudgets system [Carlsson and Hallgren 1993] and the Concurrent Clean I/O System [Achten et al. 1992] are two recent investigations into declarative GUI programming in lazy functional languages. Fudgets, although it is implemented in the lazy languages Haskell and LML, uses an eager strategy for GUI object renderings. The Concurrent Clean system uses a partially eager evaluation mechanism for maintaining display state, invoking programmer-provided event handlers whenever applicable events occur.

Prior to the interest in declarative GUI specification, there was some related research in temporal aspects of lazy applicative languages. Typical of this work were the optimizations done by Ostrum [Ostrum 1981], Faustini [Faustini 1982], and Denbaum [Denbaum 1983] in the context of the lazy textual dataflow language Lucid [Ashcroft and Wadge 1985]. Most of these approaches include memoization, some using eager marking and some using the notion of expiration times (which is an important aspect of lazy marking), but to our knowledge, this work has not been extended to meet the requirements of GUIs. Plane Lucid [Du and Wadge 1988, 1990], an intensional spreadsheet that is an extension of Lucid, shares with Forms/3 the spreadsheet paradigm, declarativeness, and an explicit time dimension, but the authors do not address the issues of GUI cost distribution.

Directly related to the work of this thesis is prior work in the Forms family of languages. Culprit tracking takes as its starting point the lazy marking technique of Burnett's original implementation of Forms/3 [Burnett 1991].

4. Culprit Tracking

We mentioned earlier our key observation that the mouse can only be at one location on the screen at any given time. While this observation is an obvious one, the system was not taking advantage of the implications of this fact in ensuring that all on-screen values were current. Culprit tracking is an extension of the LM algorithm that capitalizes on this observation to eliminate the "polling" work needed to keep displayed values up-to-date.

To facilitate discussion, we speak of *suppliers* and *clients* of cells. *Direct suppliers* of a cell X are the argument cells in a X's formula. Indirect suppliers of X are the argument cells in the formulas of X's direct and indirect suppliers. Where we use "supplier" unmodified, we mean all suppliers, direct and indirect. A client of a cell is one that references the cell in its formula, or references another of the cell's clients in its formula. We have not needed to distinguish between direct and indirect clients, "clients" means all clients, direct and indirect.

When the system receives a user event, no other user events from other areas of the screen can possibly occur at this time. Therefore, cells dependent only on those other areas cannot have received an event. We call this user event the *culprit* because it causes the recomputation of affected cells on the screen. The cell that holds the event, the event queue, is the *primary victim (PV)*, and clients of the primary victim are also *victims*.

We can capitalize on this observation by partitioning the cells of a program into 3 categories:

1. Victim cells: cells that are time dependent because they are dependent on a user event.

2. Temporal cells: cells otherwise time dependent. The formula of a temporal cell has a reference to the system clock, or a reference to a prior value via the *earlier, fby,* or *prev* operators.

3. Static cells: cells not time dependent. Examples include constants such as integers or strings, and cells whose suppliers are all static.

The latter two categories present no opportunity for improvement. Cells in the temporal category must be recomputed at every logical time step, while those of the static category need never be recomputed. Cells of the victim category do present the opportunity for improvement. We further categorize cells in the victim category into subcategories, one for each primary victim (one for each event-queue cell in the program). For example, in the thermometer program all cells dependent on the 2 buttons are in the victim category, which has 2 subcategories, 1) those cells dependent on the F <-> C button, and 2) all cells dependent on the Color/BW button.

(Sub)Categories are not necessarily disjoint. A cell's formula may have both a temporal operator and an event receptor. Such a cell is a member of both the temporal and victim categories. A cell may also be a member of multiple subcategories of the victim category. When a cell references (directly or indirectly) multiple primary victims as arguments in its formula, it must be recomputed when any of those primary victims has been given a new value by a culprit.

The system can make use of this partitioning of cells into (sub)categories to achieve the desired computational savings. When logical time moves forward, the system must render the screen to display current values; however, if the reason for time's advance is a user event, the system need only render cells in the temporal category and in a single subcategory of the victim category, and can avoid rendering all other victim cells. This reduction in the number of cells rendered is the reason we've implemented culprit tracking.

4.1. Method

We now present details of our method of culprit tracking. First, we discuss the extra work required to track the culprit, then we discuss the screen render portion of the system where the benefits of this extra work are realized.

4.1.1. Tracking the Culprit

For the system to make use of the knowledge that a user event has occurred and can only affect victims of that culprit, it must keep track of which cells are clients of the culprit's primary victim. The system has access to such information when it demands the value of a cell and evaluates the cell's formula. If, in evaluating the cell's formula, the system finds that any direct suppliers are victims of a PV, it marks this cell as also being a victim of that

PV. This propagates culprit dependencies. The PV itself is designated a victim when the system activates the PV, providing the initial value that is propagated.

Cells in the temporal category are tracked the same way, with the initial value set by the temporal primitive operators.

4.1.2. Rendering the Screen

The system next renders the screen; here it makes use of the information maintained during the demanding of cells/evaluating of formulas. The system distinguishes the 3 cases of logical time moving: 1) a user event moves time forward, 2) a temporal operator moves time forward, and 3) otherwise, the user has manipulated the time slider to move time either forward or backwards. For each form on the screen the screen, the system checks:

Case 1: Logical Time progressed because of an event:

> select for rendering all temporal cells on the form and all cells on the form that are in the victim subcategory of the currently active PV.

Case 2: Logical Time progressed because of a temporal operator,

> select for rendering all temporal cells on the form.

Case 3: (default) Logical Time progressed because the user is moving time backwards, or manipulating the time slider:

> select for rendering all temporal and victim cells on the form.

End Case.

Demand and render those cells selected.

This concludes our description of the culprit tracking technique. To summarize, CT stores, during evaluation, time dependency information, and uses this information during screen rendering to demand fewer cells, yet still maintains correctness, as we now show.

4.2. Correctness

Here we show that lazy marking with culprit tracking preserves correctness of output, showing that the values that it allows to become out of date will never be seen by the user, nor will those out-of-date values be used for computing output.

Recall the 3 cases of changing logical time: 1) a user event moves time forward, 2) a temporal operator moves time forward, 3) the user manipulates the time slider to position time either backwards or forwards. A proof by contradiction follows:

Hypothesis: There exists a cell on-screen whose value is out-of-date.

Case 1: A user event moves time forward.

As a result:

1. The culprit defines a new value for the primary victim cell at a new time.

2. Logical time advances 1 time step.

3. For on-screen cells in the temporal category, each cell (and its suppliers) is re-demanded and rendered. For on-screen cells in the victim category, those in the subcategory of the primary victim that received the user event are re-demanded and rendered.

4. Any un-rendered cells are in the static category or in victim subcategories other than that of the currently active primary victim.

5. By our initial assumption, there exists a cell on-screen whose value is out-of-date. This cell must be one of the un-rendered cells in the previous paragraph. However, these un-rendered cells cannot be out-of-date, because:

a) The cells of the static category are, by definition, unchanging.

b) Cells in non-active victim subcategories have received no new event, so they cannot be out-of-date unless they were time dependent for some other reason, in which case they would also have been partitioned into either the temporal category or the active victim subcategory, and rendered.

This contradicts our initial assumption that there exists a cell on-screen whose value is out-of-date. Therefore, culprit tracking preserves the correctness of the output for Case 1.

Case 2: A temporal operator moves time forward.

As a result:

1. The system evaluates a temporal operator which causes time to move forward.
2. Logical time advances 1 time step.
3. For on-screen cells in the temporal category, each cell (and its suppliers) is re-demanded and rendered.
4. Any un-rendered cells are in the static category or in the victim category.
5. By our initial assumption, there exists a cell on-screen whose value is out-of-date. This cell must be one of the un-rendered cells in the previous paragraph. However, these un-rendered cells cannot be out-of-date, because:

a) The cells of the static category are, by definition, unchanging.

b) Cells in the victim category have received no new event, so they cannot be out-of-date unless they were time dependent for some other reason, in which case they would also have been partitioned into the temporal category and rendered.

This contradicts our initial assumption that there exists a cell on-screen whose value is out-of-date. Therefore, culprit tracking preserves correctness of the output for Case 2.

Case 3: The user moves time by manipulating the time slider.

As a result:

1. The system has a new current time.
2. This time may be greater or less than the previous current time.
3. All victim and temporal cells (and their suppliers) are re-demanded and rendered.
4. Any un-rendered cells are in the static category.

5. By our initial assumption, there exists a cell on-screen whose value is out-of-date. This cell must be one of the un-rendered cells in the previous paragraph. However, these un-rendered cells cannot be out-of-date, because:

a) The cells of the static category are, by definition, unchanging.

This contradicts our initial assumption that there exists a cell on-screen whose value is out-of-date. Therefore, culprit tracking preserves correctness of the output for Case 3. Therefore culprit tracking preserves correctness of output for all cases.

4.3. Implementation

To implement culprit tracking, we modified six areas of the Forms/3 implementation, approximately 50 functions and methods. These areas are: 1) data structures, 2) the form display/iconify routine, 3) event receptor routines 4) language primitives, 5) the demand subsystem, and 6) the screen rendering routine. See appendix A for samples of the modified code.

4.3.1. Data Structures

We added a global array which holds the Primary Victims (PVs) active in a program. The zeroth element is initialized to a reference to the cell System:time, which is used as a hash key for the temporal category. We also added to each cell an array indicating of which Primary Victims this cell is a client. This array is a cell's TD (timeDependent) array. To each form, we added a TD hashtable, keyed by PV. Each value in the hashtable is a pointer to another hashtable of clients of the PV key. The sub-hashtable was used so that the system need not check whether a cell is already a client of a PV; the hashing efficiently eliminates duplicates.

4.3.2. Form Display/Iconify Routine

The form display/iconify routine was altered to take advantage of the user's moving a form off-screen. Formerly, the system did not distinguish between on-screen and iconified forms. All were treated as being displayed. The system now regards an iconified form as being off-screen. To effect this, we added a daemon to the Garnet window system which

notifies the runtime system when a form is iconified and when a form is remapped (de-iconified). Values of cells on iconified forms are not computed unless they supply a value to a cell that is on-screen.

4.3.3. EventReceptor Routines

When the user loads or copies the eventReceptor primitive form, the system assigns an event queue to that receptor. We added code that adds the newly assigned event queue to a global array of Primary Victims. The event queue's TD array is also initialized to reflect that this event queue cell is a Primary Victim so this information can be passed to any client cells.

4.3.4. Language Primitives

The language primitive routines (approx. 30) were examined and the primitives that are inherently time dependent were modified. This includes such primitives as accessing the system clock, the functions that implement the *earlier, prev, and fby* primitives, and functions that map events to event queues and event Receptors.

The modifications were:

For the temporal operators, the zeroth element of the cell's TD array is set to True.

For the event routines, the nth element of the TD is set to True, where n is an index given an event queue when it was assigned.

4.3.5. The Demand Subsystem

The demand subsystem consists of about 15 functions and methods that compute the value of a cell at a given time. Here is where lazy marking occurs. Marks are placed indicating the time range over which this value is valid. Code was added so that a cell *or*'s the TD array of any supplier cells into it's TD array. Code was also added to check the TD array and, if the cell is time dependent, add a (PV, cell) tuple to the hashtable of the cell's form. All time dependencies are propagated to clients via the demand subsystem.

4.3.6. Screen Rendering Routine

The screen rendering routine was rewritten to use the information stored by the demand subsystem in each form's hashtable. We also altered the code which calls the screen rendering routine to pass the primary victim parameter, if applicable. As stated earlier, the screen rendering routine distinguishes three cases of logical time moving:

For case 1, Logical Time progressed because of an event:
select cells for rendering by accessing the form's TD hashtable twice: once to select clients of the active PV, using the active event's EQ as the key, and once to get temporals using SysTime as the key.

For case 2, Logical Time progressed because of a time dependent operator,
select cells for rendering by accessing the form's TD hashtable once, with SysTime as the key.

For case 3, (default) Logical Time progressed because the user is manipulating the time slider:
select for rendering all cells in the form's TD hashtable.

Finally, having selected which cells to render, demand and render those cells selected.

4.3.7. Recursion Unstacking

In implementing culprit tracking, we encountered an additional problem. Because of lazy evaluation, some cells are safely allowed to remain un-rendered as time moves forward. The user may demand these cells, as in the case of the user deiconifying an iconified form or moving time forward many time steps. This may cause a significant demand for computation in the case where values depend on earlier values. Requesting a cell at time 100 when the cell was last demanded at time 1 generates a request for some values at time 99, 98, 97, etc. Such heavily nested calls caused the system to run out of stack space. We solved this problem by adding a check in the demand subsystem, which, rather than recursing on such demands, iterates up from the last known computed value.

4.4. Complexity Analysis (Worst Case)

In order to compare our technique to other techniques, we now analyze the worst case complexity of CT. We treat the 2 parts of CT (tracking and rendering) separately, because the tracking part builds on the prior LM algorithm, adding an incremental cost, while the rendering part is an entirely different algorithm.

4.4.1. Cost Increment of Adding Culprit Tracking to the Demand Algorithm

We will show that culprit tracking adds, for each demand for cell X in a program, O(#directSuppliers * #eventQueues) where #directSuppliers is the number of cells that cell X references in its formula, and #eventQueues is the number of event queues active in the program[1]. That is, if the cost of demanding a cell X under the LM algorithm is D, then the cost under CT is D + (#directSuppliers * #eventQueues). This is a local picture of the cost, at the level of each demand, and does not include the cumulative cost of traversing the tree of suppliers. In the next section, we include these cumulative costs, but first we look at the incremental cost.

We derive this complexity term as follows. From sections 4.3.4 and 4.3.5, the system must perform these additional operations:

In primitive operators:

Operation	Incremental Cost
In each Temporal operator: Set the zeroth element of the cell's TD array to True.	O(1)
In the EventReceptor operator, FormsMakeEventReceptorDycon: Set the nth position of the cells TD array to True, where n is an index stored by each EventQueue when the system activates the EQ.	O(1)

Figure 4-1. The Additional Operations of Time Dependent Operations of the CT algorithm.

In demanding a cell (italics indicate operations added for culprit tracking; non-italics indicate pre-existing code from the LM algorithm):

---

[1] (Our current demonstation system has a maximum of 12 event queues; This may or may not suffice in a "real", industrial strength version.)

Operation	Incremental Cost
For each direct supplier,	
For i = 1 to #EventQueues+ 1	O(#eventQueues)
"Or" the TD array [i] of each direct supplier into the cell's TD array[i].	O(1)
If TD array[i] is True,	
hash (PV,(cell,cell)) into the form.	O(1)

Figure 4-2. The Demand Function for the CT algorithm.

The only non-constant term is that of the "or" loop, this is O(#eventQueues). This term appears inside a pre-existing loop that is O(#directSuppliers). Therefore the incremental complexity of the tracking part of CT is O(#directSuppliers * #eventQueues).

4.4.2. How Bad is the Additional Term?

How does the increment compare to the total cost of demanding a cell? Having determined the incremental cost of each demand, we can now determine the total cost of demanding a cell, including the cost of demanding all of its suppliers. We derive the demand-cost-function for each of the two techniques (DCF-OneCell$_{LM}$ and DCF-OneCell$_{CT}$) by finding the recurrence relation for each function. To simplify the discussion, we use the Basic Formula Model of Forms/3 [Burnett 1991]. The textual shortcuts used for actual programming are syntactic sugar for the Basic Formula Model. In the Basic Formula Model, textual short cuts are eliminated so that all formulas consist solely of a cell reference. For example, a formula of A + B is transformed, in the Basic Formula Model, to a reference to the result cell on a copy of the primitiveAddition form whose argument cells point to cells A and B, respectively. Using this model increases the number of forms required to program, so programmers do not use this model in actual programming, but it simplifies formal reasoning about the language.

Let k_i be the costs O(1) operations for a demand (see Figure 3-1).
Let k be a constant greater than the sum of all k_i's.
Let #DSs be the number of direct suppliers, which in the Basic Formula Model is 0 or 1.
Let n be the number of indirect suppliers of a cell.

For LM, the DCF-OneCell$_{LM}$, the cost of demanding a cell and its direct and indirect suppliers is:

$$
\begin{aligned}
T(0) &= k_1 + k_2 * \#DSs. \\
&= k_1 + k_2. \\
&\leq k. \\
\\
T(1) &= T(0) + k_1 + k_2 * \#DSs. \\
&\leq k. \\
\\
T(n) &= T(n\text{-}1) + T(1). \\
&\leq T(n\text{-}1) + k. \\
&\leq T(n\text{-}2) + k + k. \\
&\leq T(n\text{-}2) + 2k. \\
&\leq T(n\text{-}3) + 3k. \\
&\ldots \\
&\leq T(n\text{-}n) + n\,k. \\
&\leq k + n\,k. \\
&\leq O(n).
\end{aligned}
$$

This simply shows that the cost is directly proportional to the number of indirect suppliers, and (since in the Basic Formula Model the number of Direct Suppliers is 1) therefore also directly proportional to the number of suppliers (direct and indirect). This cost can be, worst case, the number of cells in the program.

For CT, the DCF-OneCell$_{CT}$, the cost of demanding a cell with n indirect suppliers is:

$$
\begin{aligned}
T(0) &= k_1 + k_2 * (\#DSs * \#EQs). \\
&= k_1 + k_2 * \#EQs. \\
&\leq k\ \#EQs. \\
\\
T(1) &= T(0) + k_1 + k_2 * (\#DSs * \#EQs). \\
&\leq k\ \#EQs. \\
&\leq k\ \#EQs. \\
\\
T(n) &= T(n\text{-}1) + T(1) + k_1 + k_2 * (\#DSs * \#EQs). \\
&\leq T(n\text{-}1) + T(1) + k * \#EQs. \\
&\leq T(n\text{-}2) + (k * \#EQs) + (k * \#EQs). \\
&\leq T(n\text{-}3) + 3\ (k * \#EQs). \\
&\ldots \\
&\leq T(n\text{-}n) + n(k * \#EQs). \\
&\leq k + n(k * \#EQs). \\
&\leq O(n * \#EQs).
\end{aligned}
$$

Is this closer to O(n) or O(n^2)? The relative magnitudes of n and #EQs depend on what program the programmer writes. For most real-world programs, we believe that #EQs will be much smaller than n. That is, the number of event queues needed to implement the GUI will be much smaller than the number of a cell's indirect suppliers, which we noted can

equal, worst case, the total number of cells in the program. (We noted earlier that 12 EQs has sufficed for our research prototype). Therefore, we believe that the complexity of DCF-OneCell$_{CT}$ tends more toward DCF-OneCell$_{LM}$'s O(n) than the possible worst case O(n^2) that occurs when every cell in the program is an event queue.

4.4.3. Cost of Rendering Cells On-Screen

The payoff for CT comes when the system renders the screen. Since CT uses a different algorithm than LM, we examine the total cost of rendering the screen. In Figure 4-3, we label the cells of a program as being Static or Changeable. Those cells that are on-screen (constitute the output) we label Onscreen. Those cells that supply on-screen cells we label Suppliers. Changeables we divide into Temporals and Victims, and we divide Victims into subcategories (one for each PV). For simplicity, we draw this diagram as though different victim categories were disjoint, although overlap is actually allowed. We label AV (Active Victims), the clients of the active Culprit.

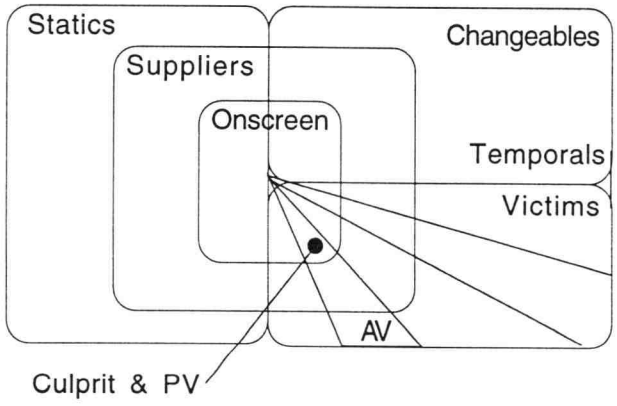


Figure 4-3. Partitioning of cells into Statics, Changeables, Temporals, Victims, Suppliers, Onscreen, and Active Victims (AV).

To determine the worst case complexity for rendering the screen, we must consider the number of cells selected for rendering, the number of cells selected for marking, and the number of cells that must be demanded. For the first two sets of cells, a cell may be accessed more than once. For the third set, we must factor in the demand-cost function (DCF), for not just one cell, but every cell selected for rendering. We define DCF to be a generalization of DCF-OneCell that gives the demand cost function for a set of cells. We

have the rendering cost function (RCF):

RCF = O(#cells selected for rendering * #times accessed +
#cells accessed for marking * #times accessed) +
DCF(#cells selected for rendering).

We can simplify this equation a bit for the CT technique. In the first term, #times accessed is equal to 1, because the system eliminates duplicates via hashtables. In the middle term, the cell is only marked once, so the #times accessed term also equals 1. The equation therefore simplifies to:

RCF_{CT} = O(#cells selected for rendering + #cells accessed for marking +
DCF_{CT}(#cells selected for rendering)).

We have 3 cases of time moving forward. Figure 4-4 shows case 1, an event pushing time forward. CT renders the shaded set of cells ((Onscreen ∩ Temporals) ∪ (Onscreen ∩ AV)), and marks the bolded set of cells (Suppliers ∩ Temporals) ∪ (Suppliers ∩ AV). The cost of the screen rendering part of CT is:

RCF_{CT} = O(|shaded| + |bolded| + DCF_{CT}(shaded)).
= O(|((Onscreen ∩ Temporals) ∪ (Onscreen ∩ AV))| +
|((Suppliers ∩ Temporals) ∪ (Suppliers ∩ AV))| +
DCF_{CT}((Onscreen ∩ Temporals) ∪ (Onscreen ∩ AV)).

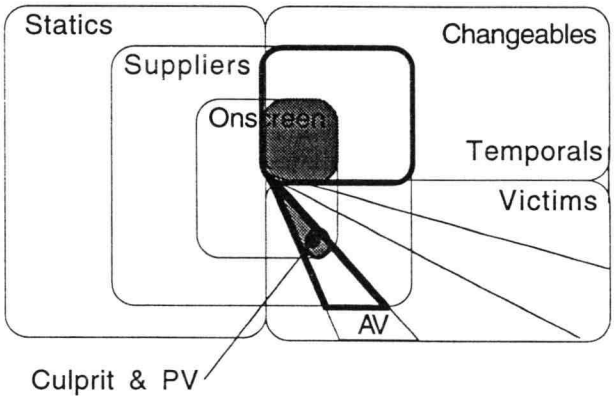


Figure 4-4. Partitioning of cells in CT technique. The shaded areas represent those cells selected for rendering, and the areas in bold represent those cells selected for marking.

In Case 2, when a temporal operator moves time forward, AV is empty; the situation is otherwise identical to Case 1. In Case 3, when the user manipulates the time slider, CT reverts to the algorithm of LM, which we discuss in the next section.

4.4.4. Comparison of CT with Other Techniques

Figure 4-5 shows the corresponding diagrams for LM, and EM. For the LM technique, the complexity equation also simplifies to:

$$RCF_{LM} = O(\#\text{cells selected for rendering} + \#\text{cells accessed for marking} + DCF_{LM}(\#\text{cells selected for rendering})).$$

because LM selects a cell for rendering only once as it iterates through a form, and marks a cell only once. The cost for RCF_{LM} is therefore:

$$\begin{aligned} RCF_{LM} &= O(\,|\text{shaded}| + |\text{bolded}| + RCF_{LM}(\text{shaded})). \\ &= O(\,|\text{Onscreen}| + |((\text{Suppliers} \cap \text{Temporals}) \cup (\text{Suppliers} \cap \text{AV}))| + \\ &\quad DCF_{LM}(\text{Onscreen})). \end{aligned}$$

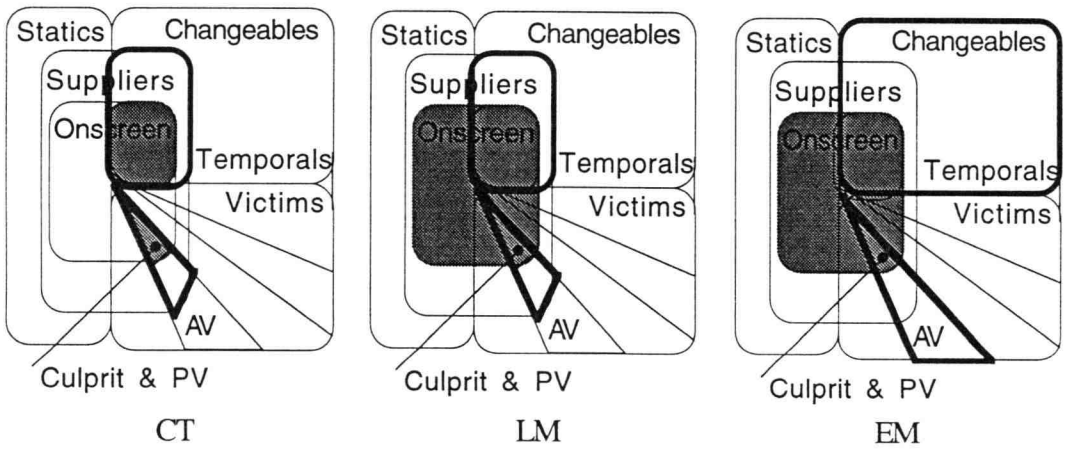


Figure 4-5. Comparison of 3 Lazy Techniques: CT, LM, and EM.

For EM, the demand-cost-function does not simplify as nicely because the marking is done in a separate traversal. The complexity of the middle term is a function of the number of clients of the cells selected for marking, and can be, worst case, $(n^2-n)/2$ where n is the number of cells that are affected by demands, when a program's dependency graph is dense, or $O(n^2)$. The equation for EM is:

$$RCF_{EM} = O(\text{ \#cells selected for rendering + (\#cells accessed for marking * \#times accessed) + } DCF_{EM}(\text{Onscreen})).$$
$$= O(\ |\text{Onscreen}| + |\text{Temporals} \cup AV|^2 + DCF_{EM}(\text{Onscreen})).$$

Now consider how CT compares to the remaining techniques, EE and NM. For EE, the first term simplifies as before. This technique does no marking, so the middle term of its RCF drops out. However, EE eagerly propagates changes. So the number of cells demanded is expanded to include all cells affected by changes. The complexity of the DCF_{EE} can be $O(n^2)$ when the program has many dependencies.

$$RCF_{EE} = O(\text{\#cells selected for rendering + } DCF_{EE}\text{ (\#cells selected for rendering and propagating changes)}).$$

$$= O(\ |((\text{Onscreen} \cap \text{Temporals}) \cup (\text{Onscreen} \cap AV))| + DCF_{EE}((\text{Onscreen} \cap \text{Temporals}) \cup (\text{Onscreen} \cap AV)).$$

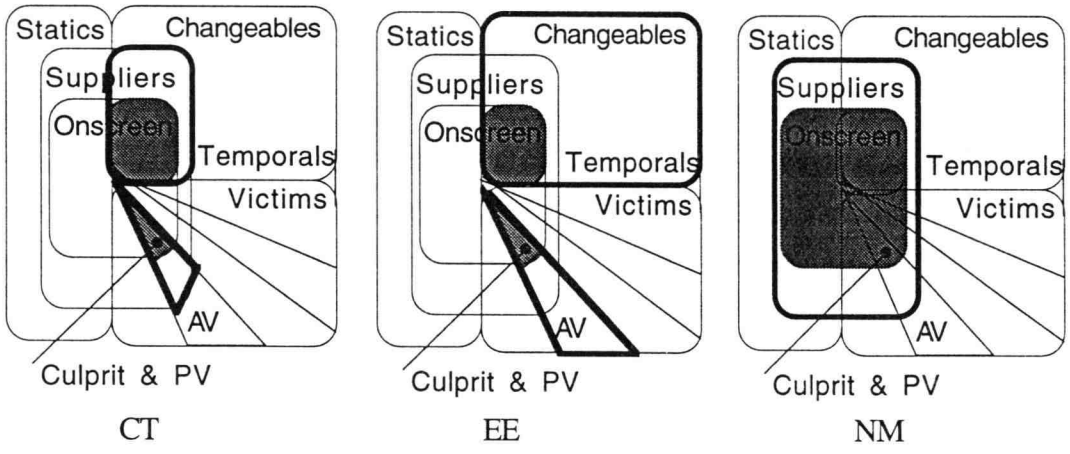


Figure 4-6. Comparison of CT with EE and NM. For EE the bolded areas are redefined to be client cells changed by eager propagation. For NM the bolded areas represent the additional work due to lack of memoization.

For the NM technique, again the first term simplifies. As for the second term, lazy evaluation with no memoization means that no marking occurs, so the second term drops out, but it also means that every access to a cell must traverse its entire supplier subgraph, so DCF_{NM} can be $O(n^2)$.

$$RCF_{NM} = O(\text{ \#cells selected for rendering + } DCF_{NM}(\text{\#cells selected for rendering})$$
$$= O(\ |\text{Onscreen}| + DCF_{NM}(\text{Onscreen})).$$

Figure 4-7 gathers the RCFs for all 5 techniques:

$$RCF_{CT} = O(\,|((Onscreen \cap Temporals) \cup (Onscreen \cap AV))| + |((Suppliers \cap Temporals) \cup (Suppliers \cap AV))| + DCF_{CT}((Onscreen \cap Temporals) \cup (Onscreen \cap AV)).$$

$$RCF_{LM} = O(\,|Onscreen| + |((Suppliers \cap Temporals) \cup (Suppliers \cap AV))| + DCF_{LM}(Onscreen)).$$

$$RCF_{EM} = O(\,|Onscreen| + |Temporals \cup AV|^2 + DCF_{EM}(Onscreen)).$$

$$RCF_{EE} = O(\,|((Onscreen \cap Temporals) \cup (Onscreen \cap AV))| + DCF_{EE}((Onscreen \cap Temporals) \cup (Onscreen \cap AV)).$$

$$RCF_{NM} = O(\,|Onscreen| + DCF_{NM}(Onscreen)).$$

Figure 4-7. Complexities of the Rendering Cost Function of the 5 Techniques.

Comparing CT with LM, we note that CT's DCF is slightly costlier than LM's, but CT renders and demands a smaller set of cells. EM has the squared middle term, which CT and LM both avoid. Although the cost of DCF_{CT} is slightly higher than the DCFs of EE and NM, DCF_{CT} demands fewer cells, and RCF_{CT} renders fewer cells, resulting in an overall lower cost (when #EQs is small compared the number of suppliers).

4.4.5. Meeting the Cost Distribution Requirement

As we noted in Chapter 1, our goal was to develop an evaluation technique that better distributed the costs of evaluating expressions in a program to meet the cost distribution requirement. EE fails this requirement because it eagerly re-computes all clients of changed values, even those that do not contribute to the output. NM fails this requirement because, without marking, non-changing and even static values must be re-computed. EM also fails this requirement because it eagerly marks cells that are not Suppliers. LM fails the requirement because it renders some cells on-screen that cannot have changed at a given time step. CT does meet the cost distribution requirement, rendering the cells needed for output, accessing the necessary Suppliers, and accessing no other cells. The way that CT achieves this characteristics is by combining the best aspects of EE and LM. CT matches EE in which cells are rendered, and matches LM in which cells it marks. EE's eager propagation of changes to all clients can make it do computation that has no affect on the output, but EE is optimal in the cells it selects to render, because it only renders those cells

that do change. LM renders some cells that cannot have changed, but only marks those that do change. CT matches the rendering of EE, and the marking of LM.

5. Empirical Performance Results

In the previous chapter, we showed how CT always satisfies the Cost Distribution Requirement as well as or better than other techniques. We also showed that overall performance will vary -- CT may show a significant performance gain (due to the more efficient RCF), but may show a slight performance loss (due to the slightly more expensive DCF). In this chapter, we validate our earlier analysis with timings of actual programs, We first look at the thermometer example in 3 screen configurations. We then isolate the incremental cost that CT adds to LM in the demand subsystem. We then look at several other programs that have different characteristics than the thermometer program. All timings were done on an HP 9000/715 with 1 user; the research system runs under Lucid Common Lisp 4.1.4 and the Garnet UI 3.0.

Recall that the thermometer example of Chapter 1 has the forms, Input, Analog output, and Digital output. We assume 3 scenario's: 1) A programmer is debugging the program and wants to see all the forms in the program. 2) An end user is using the program and wants to see only the Analog output. 3) An end user is using the program and wants to see only the Digital output. The RCF columns of Figure 5-1 shows that CT shows a performance improvement in all three scenarios, with the most marked improvement in the third scenario where CT saves the (relatively expensive) cost of rendering (and therefore demanding) the cells of the Analog output. For this particular program, the performance improvement ranges from 50% to 1482%, depending on which forms are on-screen.

	RCF time in seconds		Number of cells demanded		Number of cells marked	
	pre-CT	CT	pre-CT	CT	pre-CT	CT
Debugging, 8 forms						
F<->C	19.27	12.89	7910	2550	1420	120
Color	19.66	12.60	7980	2500	1420	1020
Analog only						
F<->C	19.64	5.67	7910	1470	1420	660
Color	20.09	5.84	7980	1490	1420	660
Digital only						
F<->C	19.93	1.26	7910	470	1420	220
Color	-	-	-	-	-	-

Figure 5-1. Performance of the Thermometer program under 3 screen configurations and 2 evaluation techniques, pre-CT and CT. Averages of 5 runs of 10 clicks of each button.

CT's DCF is incrementally more costly, and CT's RCF is less costly. To find out more about the DCF cost alone, we decided to isolate the incremental cost of CT's DCF. A direct comparison of techniques would not give us this information because the different techniques render different numbers of cells. We therefore created a modified version of CT that allowed us to isolate the DCF. In the modified version, tracking calls are simply commented out. This algorithm will, in general, fail to propagate time dependencies, but does work correctly for simple programs that have no indirect time dependencies. We therefore used a simple program consisting of a single cell whose formula computes the Fibonacci sequence. We ran the program with 3 levels of eventReceptors loaded into the system. Figure 5-2 shows that, for this program, the DCF of CT is 7% to 15% more costly that the DCF of LM. This is consistent with our earlier complexity analysis.

Number of eventReceptors	RCF of CT	RCF of CT w/out tracking overhead	% degradation due to CT tracking overhead
0	3.78	3.45	9.56
5	3.80	3.44	10.50
10	3.77	3.48	8.33

Figure 5-2. Isolating CT's demand increment. Timings for Fib-iterative(51) under CT. Averages of 5 runs.

Figure 5-3 shows CT's performance on several other programs. We selected an output intensive program (Animated Selection Sort), a computation intensive program (Fibonacci-Recursive), and 2 configurations of a highly interactive program (Pong). In these programs, CT's performance ranges from a 6% degradation to an improvement of 287%. These programs are listed in Appendix B.

	RCF time in seconds		Number of cells demanded		Number of cells marked	
	pre-CT	CT	pre-CT	CT	pre-CT	CT
Animated Selection Sort	78.84	57.67	26,023	14,509	2,006	1,835
Fibonacci-Recursive(10)	29.38	29.187	1307	1307	0	0
Pong Game: Play	5.65	5.96	1,882	1,617	542	432
Pong Game: Debug	23.42	6.05	7,347	1,617	1,872	432

Figure 5-3. Performance of CT for several programs.

In general, the total performance of CT will depend on the program and the screen configuration, but CT always meets the Cost Distribution Requirement.

6. Future Work

We see several possible areas for further improve GUI performance: one extension of culprit tracking and several other strategies that, while not related to culprit tracking, will dovetail nicely with it.

6.1. Leg Tracking

In the demand subsystem, culprit tracking propagates time dependency information from suppliers to clients whenever a formula is evaluated. The lazy nature of the language means that a formula may be partially evaluated and a cell may have some as yet undiscovered time dependencies in the unevaluated portion of its formula. For example, if the condition predicate of an *if* expression has never gone false, the else "leg" of the expression has not been evaluated, and may contain a reference to a Temporal or Victim. This does not impair the correctness of culprit tracking because, should the predicate become false, the else leg propagates the dependency back to the cell so that the cell is added to the hashtable appropriately. However, once the formula has been completely evaluated, no new information can be discovered, so the propagation could cease. By having each cell keep track of whether all legs of its formula have been evaluated, and passing that information on to its clients, it may be possible to reduce the cost of the demand subsystem portion of culprit tracking to O(1) for cells whose formulas (and suppliers formulas) have been evaluated along every leg. As there is no guarantee that a formula is ever fully evaluated, the complexity reduction of leg tracking would be stochastic.

6.2. Refining the Term, On-screen

Currently, the user has a single way of moving a cell off-screen to signal that he's not interested in that cell: the user iconifies the form on which that cell resides. The granularity of the term on-screen is therefore at the form level. One can easily think of ways to offer multiple levels of granularity, such as allowing the user to specify interest at the cell, rather

than form, level. We are also studying whether it is feasible to query the Garnet window system so that the system can determine whether a cell (and its value) is obscured by another cell or form. This would add another method of specifying interest and allow the system to be more consistent with the user's perspective that any value not visible is not of interest. Problems arise in that a cell's value may appear anywhere on the form (window), not just inside the cell itself.

6.3. Look Ahead Computation

A commonly mentioned scheme for improving performance that would coexist nicely with culprit tracking is look-ahead computation. Because the system allows values of cells to become safely out-of-date, bringing a Temporal or Victim cell to the screen may force the system to demand the value for that cell, not an just the current time, but at past logical times also. One possible improvement to the system would be to pre-compute some values while the system is idly waiting for input. This would have to be done only during idle periods so as to not impair response time. This is an eager approach, even overly eager, as the system may well compute values never needed. However, the response time cost of these speculative computations will be zero to the user, since the machine cycles used to do these computations are taken while the system is idling, waiting for user input. Possible heuristics (for demanding cells not currently on screen) might be to compute those with the greatest number of client cells, or to compute those most recently on-screen.

6.4. Memoization Heuristics

The saving of prior values in temporal vectors is space intensive. The system currently stores all prior values. A useful improvement would be to develop a heuristic algorithm to bound the space usage. An initial strategy might be to maintain only three prior values for each temporal vector: the first value (at time 1), the latest value (at time = the greatest time to which the system has advanced), and the most recently accessed value (often the current time, but different if the user moves time backwards, a cell is off-screen, or a temporal operator, such as prev, causes a reference to at times other than the current time). Other heuristics might be to cache values on an LRU, client-count, or recomputation-cost basis.

6.5. Recycling Screen Objects

Another implementation optimization that would coexist nicely with culprit tracking would be to reuse GUI objects. Each screen object is composed of sub-objects. When the current implementation needs to change an object's sub-object, it destroys that sub-object and creates a new one. Our preliminary timing tests indicate that we could see performance gains by having the system reuse existing sub-objects where possible. This could be done by specializing multi-methods dispatched on two arguments, the old sub-object (currently destroyed) and the new sub-object (to be created). If they are of the same type, the appropriate variables could be copied over from new to old, rather than destroying and creating sub-objects.

7. Conclusion

In this thesis, we have introduced culprit tracking, a technique to improve the performance of screen rendering in graphical user interfaces. We noted that a mouse or keyboard event can occur at only one location on the screen at a given time; this observation allowed us to implement culprit tracking, which tracks the culprit that is the new input to a program, and enables the system to examine fewer cells during the rendering task that keeps the screen up-to-date. The cost that CT adds to the demand process is reasonably small, which means that a reduction in total cost will often result.

We stated a cost distribution requirement that, when a user manipulates a GUI screen object, he should pay only for active values that are of interest, and not pay for inactive values or values not of interest. Eager evaluation's strength is that the user pays for only active (changing) values, but its weakness it that some of those active values may not contribute to the output on the screen. Lazy evaluation with lazy marking is the opposite: the user pays only for values that contribute to the output on the screen, but some of those values may be unaffected by the active user input, and therefore be needlessly recomputed. Culprit tracking combines the desirable properties of two other techniques, eager evaluation and lazy marking to meet the cost distribution requirement, the first such technique to do so.

Bibliography

[Achten et al. 1992] Achten, P., van Gronigen, J., and Plasmeijer, M. High level specification of I/O in functional languages, *Fifth Annual Glasgow Workshop on Functional Programming, Springer Verlag Lecture Notes in Computer Science*, (1992).

[Ashcroft and Wadge 1976] Ashcroft, E. and Wadge, W. Lucid: A Formal System for Writing and Proving Programs *SIAM Journal on Computing* , Vol. 5, No. 3, (Sept. 1976), 519-526.

[Ashcroft and Wadge 1985] Ashcroft, E. and Wadge, W. *Lucid, the Dataflow Programming Language,* Academic Press, London, England, (1985).

[Burnett 1991] Burnett, M. *Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model*, Ph.D. Thesis, Univ. Kansas Comp. Sci. Dept., (1991).

[Burnett and Ambler 1990] Burnett, M. and Ambler, A. Efficiency issues in a class of visual languages. *1990 IEEE Workshop. on Visual Languages*, Skokie, IL, (Oct. 1990), 209-214.

[Burnett and Ambler 1992] Burnett, M. and Ambler, A. A declarative approach to event-handling in visual programming languages. *1992 IEEE Workshop on Visual Languages*, Seattle, Washington, (September 1992), 34-40.

[Burnett and Ambler 1994] Burnett, M. and Ambler, A. Interactive visual data abstraction in a declarative visual programming language, *Journal of Visual Languages and Computing* 5, 1, (March 1994), 29-60.

[Carlsson and Hallgren 1993] Carlsson, M. and Hallgren, T. Fudgets - a graphical user interface in a lazy functional language, in *Proceedings ACM FPCA '93,* Copenhagen, Denmark, (1993).

[Denbaum 1983] Denbaum, C. *A Demand-Driven, Coroutine-Based Implementation of a Nonprocedural Language,* TR 83-01, Dept. Comp. Sci., Univ. Iowa, (1983).

[Du and Wadge 1988] Du, W. and Wadge, W., An intensional language as the basis of a 3-D spreadsheet design, 1988 *International Conference on Computer Languages,* Miami Beach, FL, October 9-13, 1988, 2-9.

[Du and Wadge 1990] Du, W. and Wadge, W., A 3D spreadsheet based on intensional logic, *IEEE Software*, 7(3), May 1990, 78-89.

[Faustini 1982] Faustini, A. *The Equivalence of a Denotational and an Operational Semantics for Pure Dataflow,* Ph.D. Dissertation, Univ. Warwick, Coventry, UK, (1982).

[Faustini and Lewis 1986] Faustini, A. and Lewis, E. Toward a real-time dataflow language, *IEEE Software*, (January 1986), 29-35.

[Gordon 1993] Gordon, A. An operational semantics for I/O in a lazy functional language, in *Proceedings ACM FPCA '93,* Copenhagen, Denmark, (1993).

[Haeberli 1988] Haeberli, P. ConMan: A visual programming environment for interactive graphics, *Computer Graphics* 22, 4, (August 1988), 103-111.

[Hudak and Sundaresh 1988] Hudak, P. and Sundaresh, R. On the expressiveness of purely functional I/O systems, Research Report YALEU/DCS/RR-665, Yale Univ. Dept. of Computer Science, (Dec. 1988).

[Hughes 1985] Hughes, J. Lazy memo-functions. In *Lecture Notes in Computer Science #201*, (J. Jouannaud, ed.), *FPCA*, Nancy, France, (Sept. 1985), 129-146.

[Ingalls et al. 1988] Ingalls, D. et al. Fabrik: a visual programming environment, *OOPSLA '88 Proceedings*, San Diego, CA, (Sept 1988) 176-190.

[Lau-Kee et al. 1991] Lau-Kee, D. et al. VPL: an active, declarative visual programming system, *1991 IEEE Workshop. on Visual Languages*, Kobe, Japan, (Oct. 1991), 40-46.

[Launchbury and Peyton-Jones 1994] Launchbury, J. and Peyton-Jones, S. Lazy functional state threads, *SIGPLAN '94 Conf. on Programming Language Design and Implementation*, Orlando, FL, (June 1994), 24-35.

[Myers et al. 1990] Myers, B. et al. Garnet: comprehensive support for graphical, highly interactive user interfaces, *Computer*, (Nov. 1990), 71-85.

[Ostrum 1981] Ostrum, C. *The Luthid 1.0 Manual*, Univ. of Waterloo, Waterloo, Ontario, Canada, 1981.

[Shu 1988] N. Shu, *Visual Programming*, Van Nostrand Reinhold, New York, 1988

[Smith 1988] Smith, D. Visual programming in the interface construction set, *IEEE Workshop. on Visual Languages,* Pittsburgh, PA, (Oct. 1988), 109-120.

[Tanimoto 1990] Tanimoto, S. VIVA: A visual language for image processing, *Journal of Visual Languages and Computing* 1(2) 127-139, June 1990.

[Wilde and Lewis 1990] Wilde, N. and Lewis, C. Spreadsheet-based interactive graphics: from prototype to tool, *CHI '90 Proceedings*, (April 1990), 153-159.

Appendices

Appendix A. Source Code Listings

```
(defclass RO (displayable)
 ((id :accessor displayable-id :initarg :id :initform nil)
  (formula :accessor hidden-displayable-formula :initarg :formula  :initform nil)
  (object :accessor displayable-object :initarg :object :initform nil)
  (parentFormID :accessor displayable-parentFormID :initarg :parentFormID  :initform nil)
  (parentForm :accessor hidden-displayable-parentForm :initarg :parentForm  :initform nil)
  (latestComp :accessor displayable-latestComp  :initarg :latestComp :initform nil)
  (earliestComp :accessor displayable-earliestComp  :initarg :earliestComp :initform nil)
  (tvTerminated?  :accessor displayable-tvTerminated? :initarg :tvTerminated? :initform nil)
  (timeDependent  :accessor displayable-timeDependent :initarg :timeDependent :initform
            (make-array (1+ $EventQueues-number) :initial-element nil))
                  ;had to hardcode $EventQueues-number 'cuz RO3 loads before
                  ;$EventQueuesAvailable is set in eventReceptor3
 )
 (:documentation "Referenceable Object class.") )


(defclass form (displayable)
 (
  (id         :accessor displayable-id :initarg :id :initform nil)
  (name       :accessor displayable-name :initarg :name :initform "no-name")
  (win        :accessor displayable-win :initarg :win :initform nil)
  (constr-name :accessor  displayable-constr-name :initarg :constr-name :initform nil)
  (subformIdList :accessor displayable-subformIDList :initarg :subformIDList
     :initform nil)
  (subformp    :accessor displayable-subformp :initarg :subformp :initform nil)
  (cellTable     :accessor displayable-cellTable :initarg :cellTable  :initform nil)
  (nameTable   :accessor displayable-nameTable :initarg :nameTable  :initform nil)
  (attributeList  :accessor displayable-attributeList   :initarg :attributeList :initform nil)
  (cellsTimeDependent :accessor displayable-cellsTimeDependent
    :initarg :cellsTimeDependent
    :initform (make-hash-table :test #'equal))
  (updateTime  :accessor displayable-updateTime  :initarg :updateTime :initform nil)
  )
 (:documentation "Forms class")
)

(defvar          $PrimaryVictims (make-array (1+ $EventQueues-number)))


;;;-------------------------------------------------------------------------
;;; AssignEventQueue is what assigns an eventQueue
;;;   to this partic eventReceptor if it doesn't already have one.
;;;
;;; Updates the system's info about this assignment, and
;;; returns the string version of the id of the eventQueue assigned.
;;;
```

```
(defun displayable-assignEventQueue (name)
 (with-scheduling-inhibited
  (let (eq eqid strname td-index)
    (setf strname (intrinsic-value name))
    (setf eqid (cdr (assoc strname $EventQueuesAssigned :test #'equalp)))
    (if eqid
        (setf eq nil)
        (progn
            (setf eqid (car $EventQueuesAvailable))
            (if eqid
                (progn
                    (setf eq (displayable-getcell $SystemForm eqid))
                    (setf $EventQueuesAvailable (cdr $EventQueuesAvailable))
                    (setf $EventQueuesAssigned (cons `(,strname . ,eqid)
                            $EventQueuesAssigned))
                    (setf td-index  (1+  ; 1+ 'cuz Sys:Time is zeroth
                        (position eqid $EventQueuesStatic :test #'equal)))
                    (setf (aref (displayable-timeDependent eq) td-index) T)
                    (setf (aref $PrimaryVictims td-index) eq)
                    (setf (displayable-td-index eq) td-index)
                    (displayable-add-cellsTimeDependent $SystemForm eq)
          )))) ;; end if

   (cond ((and (stringp $SavedSystem) (not $AllEventQueuesLoaded)))
      ((null eq) nil)
      ( t (progn
           (setf (displayable-latestComp eq) nil)
           (setf (displayable-earliestComp eq) nil)
            (if (= (car $CurrentTime) 1)
            (progn
                (make-TVnode eq
               :time '(1)
               :value (make-eventDycon :no-event 0 0 nil nil '(1)
                           (displayable-demand $SystemForm
                        $TimeCellID
                           :seeTime '(1))))
                (make-TVNode eq :time '(2)))
            (progn
                (make-TVnode eq
               :time '(1)
               :value (make-eventDycon :no-history 0 0 nil nil '(1)
                           (displayable-demand $SystemForm
                        $TimeCellID
                           :seeTime '(1))))
                (make-TVnode eq
               :time $CurrentTime
               :value (make-eventDycon :no-event 0 0 nil nil
                        $CurrentTime
                           (displayable-demand $SystemForm
                        $TimeCellID
                           :seeTime $CurrentTime)))
                (make-TVNode eq :time (time-addOne $CurrentTime))))
          ))
```

```
   ) ;; end cond

   (setf eq (displayable-getcell $SystemForm eqid))
     (roobj-update-value (displayable-object eq))

   eqid
)))

; ------------------------------------------------------
; formsMake-eventReceptorDycon
; exists soley because we need a hook to set timeDependent stuff.
; The fmla for eventReceptors is make-eventReceptorDycon. I made it
; a forms operator because if we call make-eventReceptorDycon directly
; we've lost which cell the dycon's going into
; This intercepts the call, allowing us the say that the cell is dependent
; on it's EventReceptor
;
(defun formsMake-EventReceptorDycon (arg-list form cell seeTime &key debug)
     (let* (ans defTime expireTime termTime)
       (multiple-value-setq (ans defTime expireTime termTime)
          (formsLispEval 'make-eventReceptorDycon arg-list form cell seeTime :debug
                debug))
     (setf (aref (displayable-timeDependent cell) (displayable-td-index
          (displayable-getcell $SystemForm (displayable-queueAddress ans)))) T)
     (displayable-add-cellsTimeDependent form cell)
     (values ans defTime expireTime termTime)
))


;;; ------------------------------------------------------
;;; formsTimer -- support for the clock function -- an implementation function,
;;;  not a primitive form.
;;;
;;; formsTimer - the easier to way to handle formsTimer, given the new
;;; approach to time (using the Garnet animation interactor to drive time)
;;;
(defun formsTimer (arg-list form cell seeTime &key debug)
  (let ((ans nil) (Aans nil) (defTime nil)
     (Aref (car arg-list))
     )
     (if (null $TimeCellId)
          (progn (setf $timeCellId (displayable-id cell))
                 (setf $TimeCell cell)
                 (setf (aref $PrimaryVictims 0) cell)))
     (setf Aans (arg-process Aref cell form seeTime :debug debug))
     (setf (aref (displayable-timeDependent cell) 0) T)
     (displayable-add-cellsTimeDependent form cell)

    (cond
     ((eql Aans :universal)
      (setf ans (get-universal-time))
      (setf defTime seeTime))

     ((eql Aans :decoded)
```

```
    (setf ans (actualTimeDecode (get-universal-time)))
    (setf defTime seeTime))
   )
  (values ans defTime (time-addOne defTime) nil)
)) ;end formsTimer

;;; ------------------------------------------------------
;;; formsFby
;;;
;;; Given args A, B, C returns A || B until C.
;;;  eg: given < 1 >, < _ 3 _ 5 > ,< false > returns < 1 3 _ 5 >
;;; fby needs its previous components to figure out where the next one
;;;  is defined, which means it & everything it uses end up being built
;;;  sequentially.
;;; We accomplish this by calc'ing the node exactly located at the "mark"
;;;  rather than the one visible at seeTime.
;;;
;;; precondition -- A is a single-component temporal vector.

;;; here's the textual equivalent:
;;;
(defun formsFby (arg-list form cell seeTime &key debug)
 (declare (ignore cell))
 (if (not (formtable-find 'fby)) (fby-example))
 (let* ((ans nil) (defTime nil) (expireTime nil) (termTime nil)
    (Aref (car arg-list))
    (Bref (cadr arg-list))
    (Cref (caddr arg-list))
    (fbyName nil)
   )
  (if (typep aref 'cellref)
   (setf aref (make-cellRef (unravel-self-refs (cellRef-form aref) form)
                (cellRef-cellid aref))))
  (if (typep aref 'list)
   (setf aref (map-bottom-up #'(lambda (x)
          (if (typep x 'cellRef)
           (make-cellRef (unravel-self-refs (cellRef-form x) form)
                (cellRef-cellid x))
           x))
        aref)))
  (if (typep bref 'cellref)
   (setf bref (make-cellRef (unravel-self-refs (cellRef-form bref) form)
                (cellRef-cellid bref))))
  (if (typep bref 'list)
   (setf bref (map-bottom-up #'(lambda (x)
          (if (typep x 'cellRef)
           (make-cellRef (unravel-self-refs (cellRef-form x) form)
                (cellRef-cellid x))
           x))
        bref)))
  (if (typep cref 'cellref)
   (setf cref (make-cellRef (unravel-self-refs (cellRef-form cref) form)
                (cellRef-cellid cref))))
  (if (typep cref 'list)
```

```
    (setf cref (map-bottom-up #'(lambda (x)
            (if (typep x 'cellRef)
              (make-cellRef (unravel-self-refs (cellRef-form x) form)
                    (cellRef-cellid x))
              x))
          cref)))
   (setf fbyName (make-constr-name :name "fby"
                    :list `((a <- ,aref) (b <- ,bref)
                        (until <- ,cref))))
   (multiple-value-setq (ans deftime expireTime termTime)
   (displayable-demand fbyName 'D :seetime seeTime :debug debug))
    (setf (aref (displayable-timeDependent cell) 0) T)
    (displayable-add-cellsTimeDependent form cell)
   (values ans defTime expireTime termTime)
))

;;;---------------------------------------------------------------------
;;; Displayable-add-cellsTimeDependent
;;;  called from displayable-demand,  add a timeDependent cell to the form's
;;;  cellsTimeDependent hashtable, keyed on timeCell, which will be either
;;;  System:Time or an EventQueue.
(defmethod displayable-add-cellsTimeDependent ((aForm form) aCell)
 (let* ((hashtable (displayable-cellsTimeDependent aForm))
        (tdarray (displayable-timeDependent aCell))
        subHash PV bit
       )
    (dotimes (i $EventQueues-number)
       (setf bit (aref tdArray i))
       (if (and bit (not (matrixIDp (displayable-id aCell))))    ;don't add
                                                                 ;parts of matrices
       (progn
            (setf PV (aref $PrimaryVictims i))
            (setf subHash (gethash PV hashtable))
            (if (null subHash) (setf subHash (make-hash-table)))
            (setf (gethash aCell subHash) aCell)
            (setf (gethash PV hashtable) subHash)
     )));progn, if, dotimes
))

(defun displayable-orSet-timeDependent (cell cell2)
 (let* ((td1 (displayable-timeDependent cell))
        (td2 (displayable-timeDependent cell2))
        )
    (dotimes (i (length td1))
      (setf (aref td1 i) (or (aref td1 i) (aref td2 i))))
))

;;;---------------------------------------------------------------------
;;; Displayable-remove-cellsTimeDependent
;;; called when you cut a cell,
;;; or by force-update, when you're tossing the TV
;;;
(defmethod displayable-remove-cellsTimeDependent ( acell)
 (let* ((hashtable (displayable-cellsTimeDependent
```

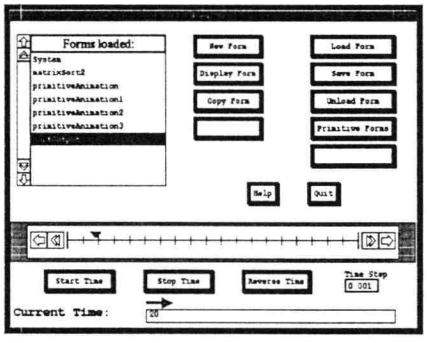
```
            (displayable-parentForm aCell))))
    (maphash #'(lambda (key subhash) (declare (ignore key))
        (remhash aCell subHash)) hashtable)
))

;;; ------------------------------------------------------
;;; update-as-appropriate
;;;  this functions iterates over all Garnet Windows, and if they're
;;;  not iconified, select appro cell and call roobj-update-value.
;;;  always do primitive & system time-dependents, and if cell is passed,
;;;  the cell that caused time to go forward, an eventQueue, do cells
;;;  dependent on it.
;;;
(defun update-as-appropriate ( &optional cell (deIconify nil))
(declare (ignore deIconify))
 (if (not $InUpdate-As-Appropriate)
  (progn
   (setf $InUpdate-As-Appropriate T)
   (dolist (tempFormWindow (displayable-allWindows $FormList))
    (if (and (not (equal (g-value tempformWindow :visible) :iconified))
          (not (equal $CurrentTime (displayable-UpdateTime
                (displayable-formData tempFormWindow)))))
      (let* ((temphash (clrhash $UpAsAppHash))
          (hashtable (displayable-cellsTimeDependent
                (displayable-formdata tempformwindow))))
        (cond
         ((typep cell 'eventQueue)                                    ;a culprit
          (set-tempHash (getHash $TimeCell hashtable) tempHash)   ;do Sys:time
          (set-tempHash (getHash cell hashtable) tempHash))          ;do EQ
         ((time= $CurrentTime $LastTime)                              ;new$lastTime
          (set-tempHash (getHash $TimeCell hashtable) tempHash))
         (t (maphash #'(lambda (key val) (declare (ignore key));default
                (set-tempHash val tempHash)) hashtable)));do 'em all
                                                    ;now update all the gathered cells
        (maphash #'(lambda (key val) (declare (ignore key))
              (roobj-update-value (displayable-object val))) tempHash)
        (setf (displayable-updateTime
              (displayable-formdata tempFormWindow)) $CurrentTime)
  ))) ;end let, if visible, dolist
(setf $InUpdate-As-Appropriate nil)
)))

(setf kr::*pre-set-demon* 'demon-for-update-as-appropriate)

(setf $TimeCellId nil) ;don't execute demon til everything's loaded
(defun demon-for-update-as-appropriate (schema slot v)
 (if $TimeCellId
 (if (and (and (equal slot :visible) (equal v T))
     (equal (g-value schema :visible) :IconiFied))
   (progn
;    (format T "Demon: slot ~a val: ~a" slot v)
    (with-demon-disabled 'demon-for-update-as-appropriate
     (progn (let (oldval)
      (setf oldVal (g-value schema :visible));save old value, to reset
```

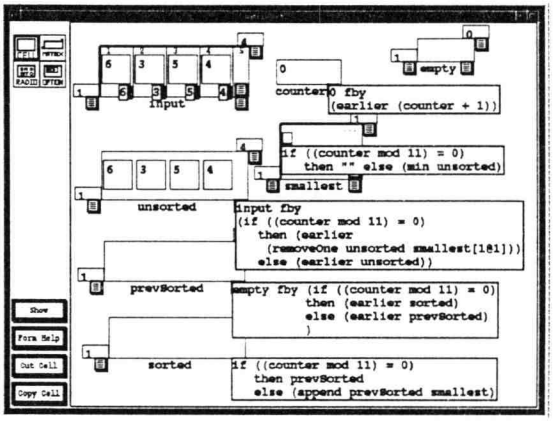
```
    (s-value schema :visible T)
    (update-as-appropriate nil T) ;pass deIconify Flag
    (s-value schema :visible oldVal) ;reset
)))))))

;? included only for recursion unstacking.

(defun displayable-demand1 (formParam cellParam &key
                            (seeTime $CurrentTime)
                            (debug nil))
 (let* ((formName (if (typep formParam 'form)
            (displayable-id formParam) formParam))
    (form (if (typep formParam 'form)
          formParam
        (formTable-find formName)))
    (cellID (if (typep cellParam 'RO)
          (displayable-id cellParam) cellParam))
    (cell (if (typep cellParam 'RO) cellParam nil))
    (ans nil) (defTime nil) (termTime nil) (expireTime nil)
   )

  ;; Find the form and cell.  If form or cell don't
  ;;  yet exist, create them.

  (if (null form)
   (setf form (form-create-from-abs-ref formName )))

  (if (null cell)
   (setf cell (displayable-getCell form cellID )))
  ;; 07/11/95 rebecca
  ;; Note that it is possible for cell to be nil at this point.
  ;; This happens when a formula (notably matrix formulas) creates an
  ;; invalid cell reference.

  ;; 02/10/95 JWA
  ;; this do loop below is to handle the case where a cell's being demanded
  ;; at a time far in the future of when it last was.  The danger is of
  ;; recursive demands causing a stack overflow.  It's handled by, rather
  ;;  than recursing back in time, iterating forward in time

  (if (and cell (aref (displayable-timeDependent cell) 0)
             (displayable-seeTimeInAdvance cell seeTime))
        (do ((i (time-addOne (TvNode-time cell
           (displayable-latestKnownComp cell))) (time-addOne i))
          )
         ((time> (displayable-timeNodeExpired cell
                 (displayable-latestKnownComp cell)) seetime))
         (displayable-demand1 form cell :seetime i :debug debug)
         ;(format T "loop: i: ~a seetime: ~a~%" i seetime)
         (format T ".")
         ))

  (if debug (format t "~%~a ~a:~a seetime=~a ~%"
            "Demand "
```

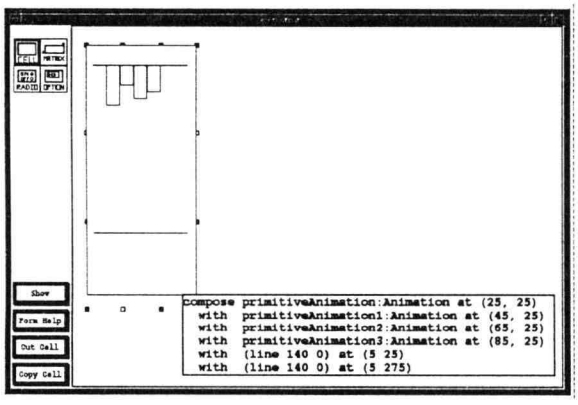
```
            (displayable-id form)
            cellID
            seeTime))

  (cond ((null cell)
      (if $EvalDebug (format t "CellID ~a does not exist~%" cellID))
      ;; This is not necessarily an error and
      ;; we can still return intelligent values.
      (multiple-value-setq (ans deftime expiretime termtime)
                (displayable-nonexistent-demand cellID form
                      :seeTime seeTime :debug debug))
      (values ans deftime expiretime termtime))
     ((time< seeTime (car $TimeLine))
      (values $aNoValueDycon seeTime (car $TimeLine) nil))
     ( t (multiple-value-setq (ans deftime expiretime termtime)
                  (displayable-cell-demand cell form
                        :seeTime seeTime :debug debug))
        (if (and (typep ans 'noValueDycon)
             (TVNode-see cell (time-subtractOne deftime)))
          (let ((oldDeftime deftime) (newExpireTime expiretime))
           (multiple-value-setq (ans deftime expiretime termtime)
             (displayable-cell-demand cell form
               :seeTime (time-subtractOne deftime) :debug debug))
           (TVNode-delete cell (TVNode-see cell oldDefTime))
           (values ans deftime newExpiretime termtime)
           )
          (values ans deftime expiretime termtime))
        ))
    ;; Do not add stuff here.  The return value of the cond statement
    ;; above is the function's return value.
))
```

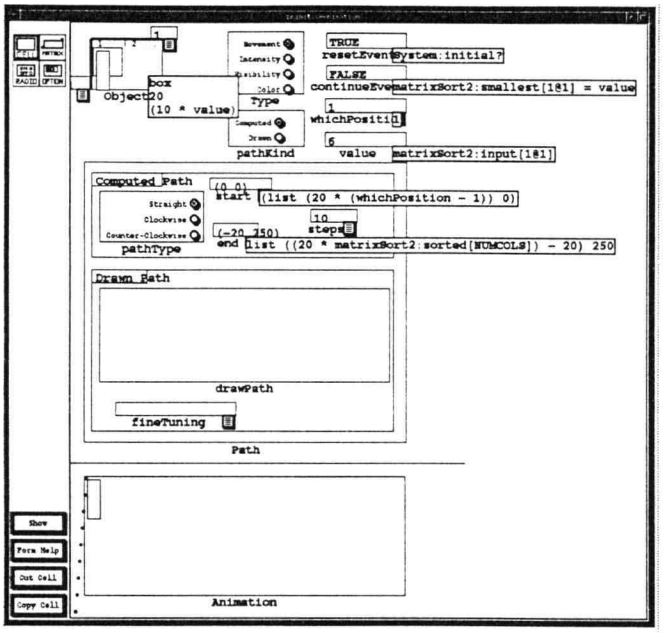
Appendix B Program Listings

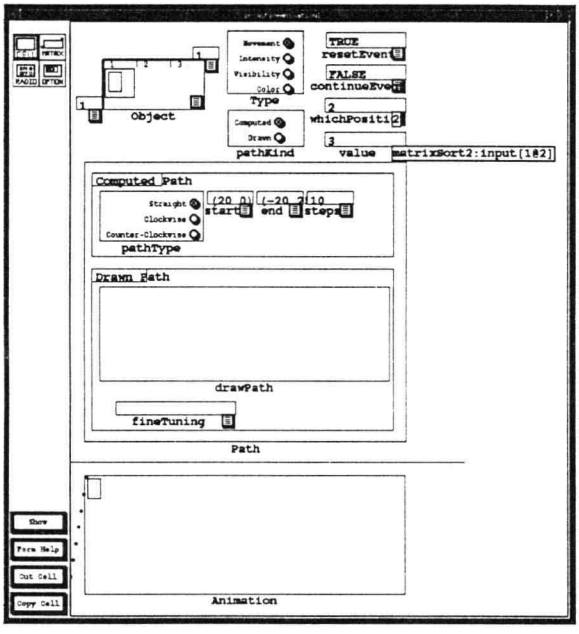
Animated Matrix Sort

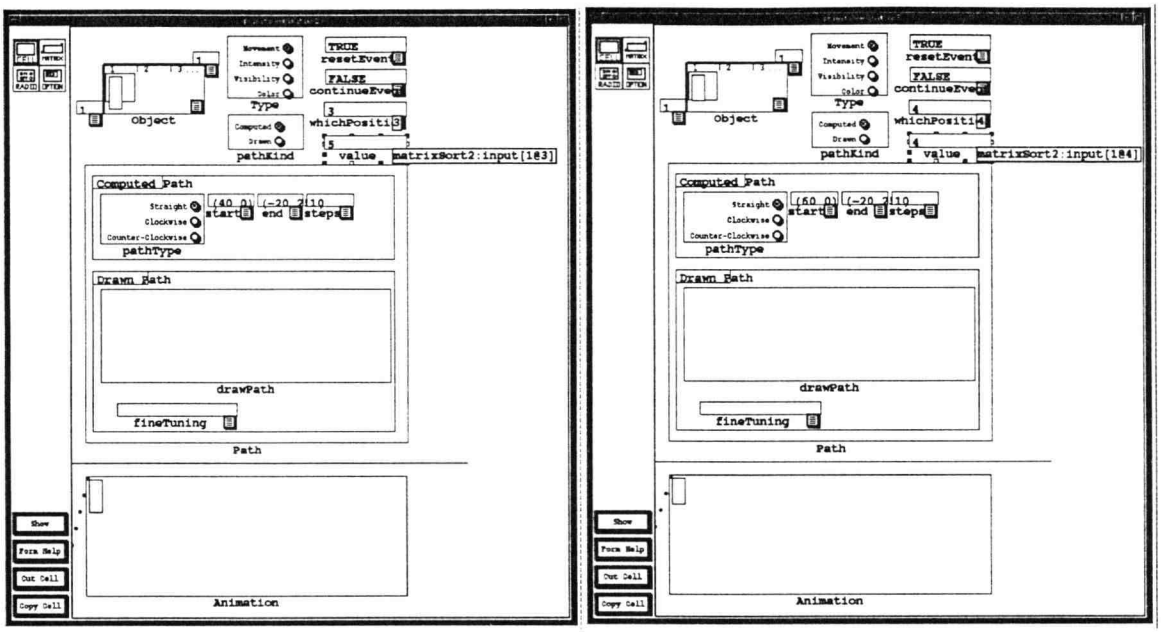




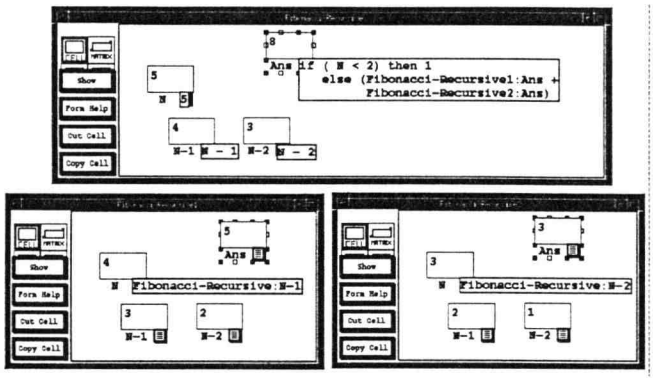



Object
box
(10 * value)
Type
pathKind
TRUE
resetEvent
FALSE
continueEve
matrixSort2:smallest[1@1] = value
whichPositi
value
matrixSort2:input[1@1]
Computed Path
Straight
Clockwise
Counter-Clockwise
pathType
start
(list (20 * (whichPosition - 1)) 0)
steps
end
list ((20 * matrixSort2:sorted[NUMCOLS]) - 20) 250
Drawn Path
drawPath
fineTuning
Path
Show
Form Help
Cut Cell
Copy Cell
Animation

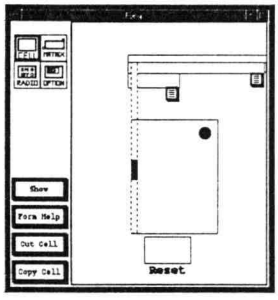

Object
Type
pathKind
TRUE
resetEvent
FALSE
continueEve
whichPositi
value
matrixSort2:input[1@2]
Computed Path
Straight
Clockwise
Counter-Clockwise
pathType
start
end
steps
Drawn Path
drawPath
fineTuning
Path
Show
Form Help
Cut Cell
Copy Cell
Animation

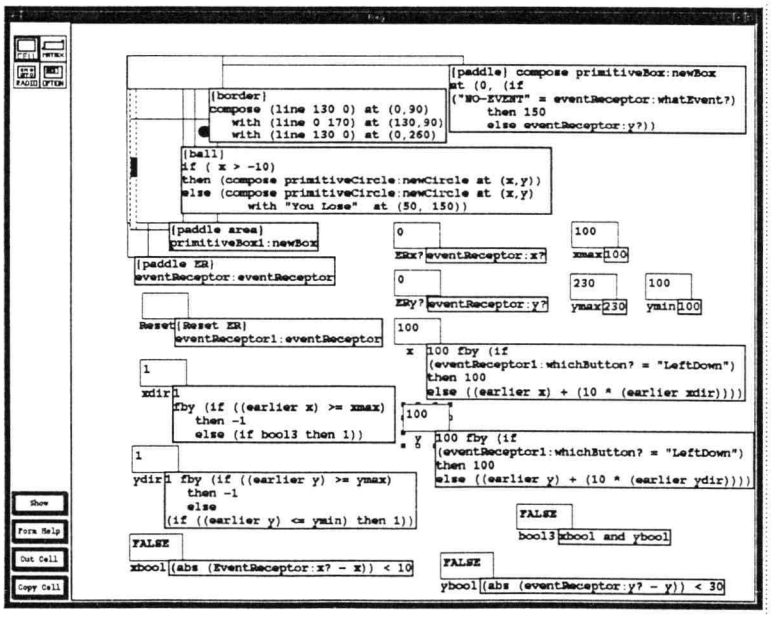

Object
Type
pathKind
TRUE
resetEven
FALSE
continueEve
whichPositi
value
matrixSort2:input[183]
Computed Path
Straight
Clockwise
Counter-Clockwise
pathType
start
end
steps
Drawn Path
drawPath
fineTuning
Path
Show
Form Help
Out Cell
Copy Cell
Animation
matrixSort2:input[184]

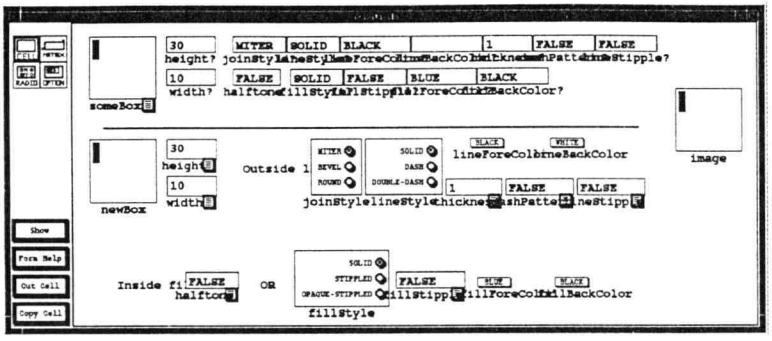
Fibonacci-Recursive

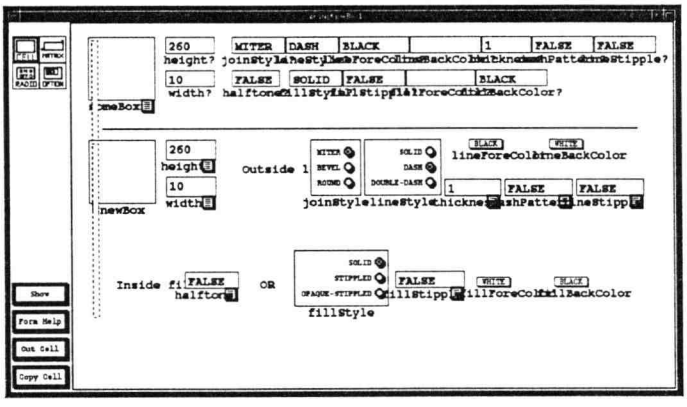


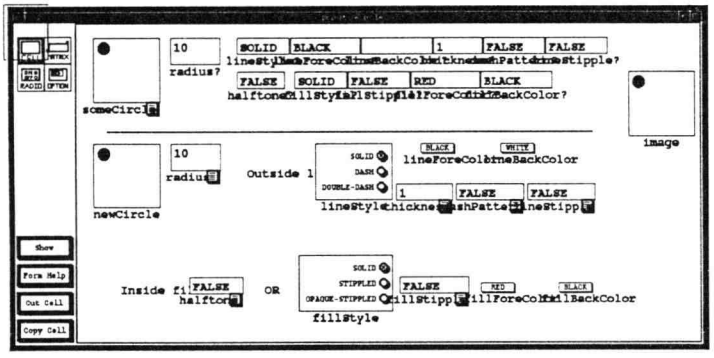
Pong

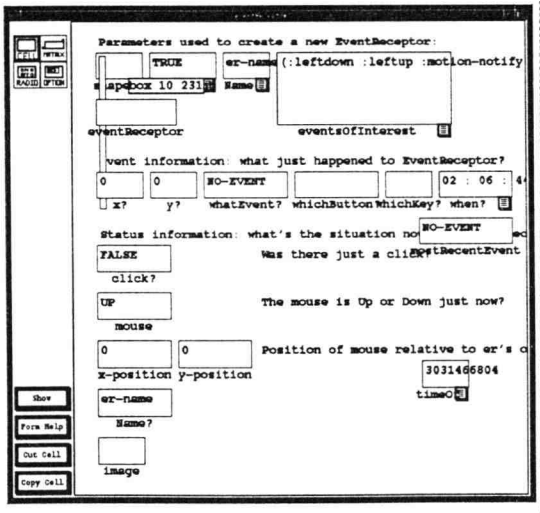


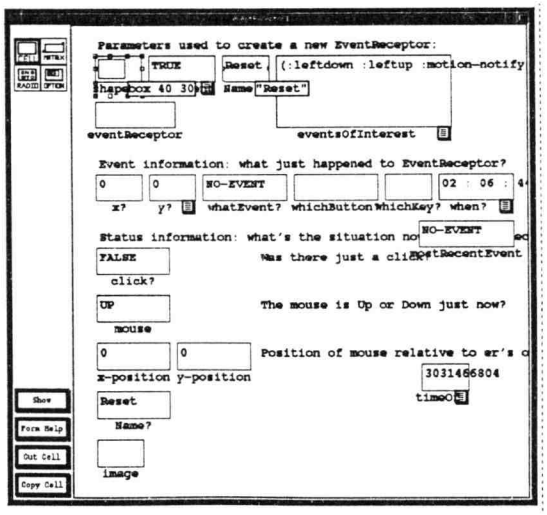



30
height?
10
width?
someBox
newBox
image
Show
Form Help
Out Cell
Copy Cell
FALSE
halftone
OR
fillStyle


260
height?
10
width?
newBox
Show
Form Help
Out Cell
Copy Cell
FALSE
halftone
OR
fillStyle


10
radius?
someCircle
newCircle
image
Show
Form Help
Out Cell
Copy Cell
FALSE
halftone
OR
fillStyle


Parameters used to create a new EventReceptor:
TRUE
Name
eventReceptor
eventsOfInterest
Event information: what just happened to EventReceptor?
0
0
NO-EVENT
x?
y?
whatEvent?
whichButton
whichKey?
when?
Status information: what's the situation no
FALSE
click?
Was there just a click?
UP
mouse
The mouse is Up or Down just now?
Position of mouse relative to er's o
x-position
y-position
3031466804
time0
er-name
Name?
image
Show
Form Help
Out Cell
Copy Cell


Parameters used to create a new EventReceptor:
TRUE
Reset
Name "Reset"
eventReceptor
eventsOfInterest
Event information: what just happened to EventReceptor?
0
0
NO-EVENT
x?
y?
whatEvent?
whichButton
whichKey?
when?
Status information: what's the situation no
FALSE
click?
Was there just a click?
UP
mouse
The mouse is Up or Down just now?
Position of mouse relative to er's o
x-position
y-position
3031466804
time0
Reset
Name?
image
Show
Form Help
Out Cell
Copy Cell