DATA-BASE CONSIDERATIONS FOR ANIMATION SYSTEMS

J. Barenholtz
Simon Fraser University

ABSTRACT

Trees have become a standard representation scheme for graphic data. The author's animation system, GRAX, uses trees to represent not only static organization within frames, but also time-dependent animation parameters (rotation, position, etc.). This scheme works as well as long as all parameters are globally available to all data structures when they are interpreted. However, if one data structure contains information necessary to the interpretation of another (e.g., Item A refers to a position defined dynamically within item B), then it is necessary to create an interpretive procedure for extracting the needed data.

This paper describes the data structures employed in GRAX, and discusses several possible techniques for communication of data between different trees. Amongst the techniques discussed are: (1) methods involving no pre-computation, (2) pre-interpretation of selected data under interpreter control, (3) re-ordering of the branches in the data tree, and (4) abandonment of the interpretive data structures in favour of compiled, optimized representations.

CONSIDÉRATIONS SUR LES STRUCTURES INFORMATIONNELLES APPLIQUÉES A DES SYSTÈMES NUMÉRIQUES D'ANIMATION

RÉSUMÉ

Les structures arbre sont maintenant un méthode de représentation acceptée pour l'information graphique. Le système d'animation GRAX mis sur pied par l'auteur, utilise structures en arbre pour représenter non seulement l'organisation d'un cadre simple, mais aussi pour le contrôle des paramètres variables en animation. Cette approche fonctionne très bien à condition que tous les paramètres soient accessibles à toutes les structures en arbre lorsque celles-ci sont interprétées. Toutefois, si une structure contient de l'information essentielle à l'interprétation d'une seconde structure (par exemple l'élément A réfère à une position définie dynamiquement dans l'élément B), il est alors nécessaire d'établir une procédure d'interprétation pour extraire l'information désirée.

Cet exposé décrit les structures informationnelles utilisées dans les programmes GRAX et discute différentes techniques de communication informationelle entre les arbres différents. Entre autres on parlera de (1) méthodes ne requérant pas de pré-calculation, (2) pré-interprétation d'information sous le contrôle de l'interprète, (3) restructuration des branches dans les structures en arbre, et (4) l'abandon des structures informationnelles interprétative en faveur de représentations compilées et efficaces.

DATA BASE CONSIDERATIONS FOR ANIMATION SYSTEMS

Jerry Barenholtz

The use of heirarchical data structures for representing graphic data has become quite widespread. In the author's animation system, GRAX [1], as in many others, each object is represented as a rooted, directed graph with no loops. Such graphs are called trees by those who do not mind leaves being shared by various branches. Those trees which represent static graphic entities may be analysed in terms of three types of nodes, which we call SKETCHs, MODifys, and GROUPs.

SKETCHs are the simplest type of node, used to represent simple, static drawings. A SKETCH is a terminal node, and takes no display parameters. In GRAX, a SKETCH is just a collection of the X, Y, endpoints of a set of line segments, and a partition indicating where "pen-up" and "pen-down" instructions are to be executed. The MODify nodes have exactly one arc out. The several constants which MOD also accepts are parameters which determine some part of the graphic context (e.g. scale, position) for the display of the subtree on the arc out. The GROUP nodes have any number of arcs out. They accept no parameters, and have no internal structure. They simply cause all the subtrees on out arcs to be displayed in the same frame.

The data structures of many graphics systems are similar to the GROUP-MOD-SKETCH trees used in GRAX. This is a useful representation scheme because it is apt to the organization of scenes, easy to edit and manipulate, and very flexible. In addition, trees lend themselves well to real-time interpretation by a task which sends data to the display unit. These features lead directly to the program organization used in GRAX and other systems. The animator communicates with the system via a foreground task, which contains commands for creating, editing, storing and fetching, etc., graphic items. There is also a background task which displays the items in the current animation. To add animation capabilities to a system designed for the display of still images, it is necessary to cause the background task to reinterpret each tree twelve or fifteen times per second, while some other tasks change parameters in various nodes. Such tasks may be part of the response routines associated with various editing commands (such as MOD), or may be associated with the data trees themselves. These self updating data forms are the animation data types.

At present, GRAX contains two animation types, p-curves and keyframes. In p-curve animations, the successive endpoints which define a sketch are analysed to produce the value of some parameter in successive frames of an animation. In this sense, p-curves are a generalization of the MOD commands. The P-CURVE nodes in GRAX have two arcs out. One of these, called the "path", can only accept a SKETCH, while the other arc is general. P-curve animation was first extensively developed by Baecker [2].

In keyframe animations, a set of key frames and the relative timing for the pairs of key frames is specified. Linear interpolation produces frames between two key frames. Each line segment in one key frame is transformed into a corresponding line in the other key frame. The

KEYFRAME nodes in GRAX may have any number of arcs out. Each arc out must take a SKETCH. Keyframe animation was extenively investigated by Burtnik and Wein [3].

When various parameters are being set or edited, as in the MOD or P-CURVE commands, the command response routines write the changing values directly into the node's data fields, so that as far as the interpreter is concerned, each node is self contained with respect to parameters.

This scheme works admirably so long as the nodes remain self contained with respect to their parameters. However, allowing complex definitions in which one item refers to another requires that nodes use external references for their parameters. Currently in GRAX, this situation occurrs only when one animation take its timing values from another. This permits synchronization of animations, and the application of p-curve animations to other p-curve or keyframe animations. In this way, animation effects can be built up in layers.

It is desirable to allow more complex cross references in time and in space between animations, since this will increase the expressiveness of the animation language. First, however, it is necessary to examine how the interpreter might handle data tress which result from the more expressive commands.

The strategies for complex references to be discussed are:

1- Straight interpretation of trees as they appear, with no preinterpretation of values.

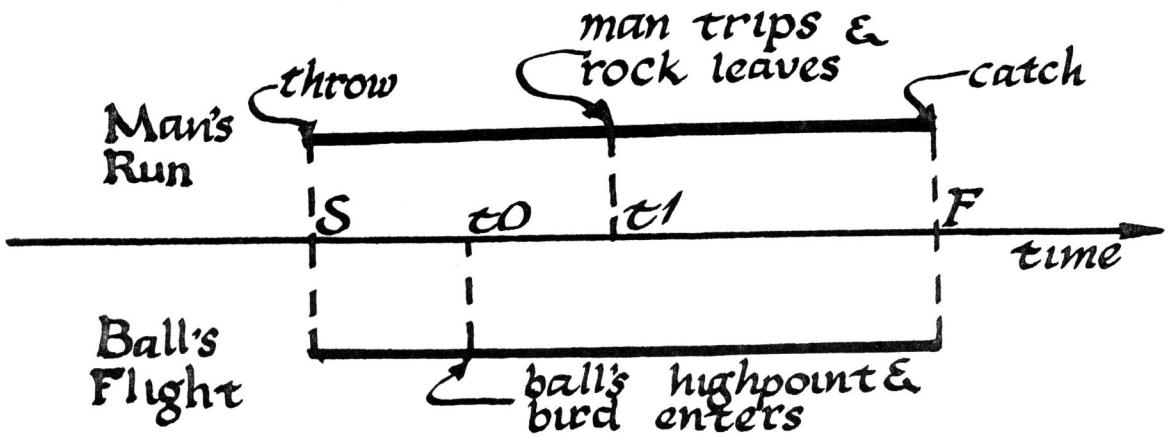
2- Pre-interpretation of selected things, making the results globally available.

3- Re-arrangement of the graph in order to get all the needed values available on one pass through the interpreter.

4- Compilation of the data structure representing an animation into an optimized program, and abandonment of the interpretive structure.

Before entering into the discussion of the various possibilities, we present as an example a bit of animation which demonstrates the typical timing problems we are trying to solve. A man throws a ball into the air, and runs across a field to the point at which the ball comes down, where he catches it. As he runs along, he trips over a rock, which slows him down.

FIG 1: A Representation of a Complex Animation



This animation might be described in terms of two paths, one of which is the ball's flight, and the other of which is the man's run. To insure the proper synchronization of the two sequences, they are tied in time at the instant when the man throws the ball, and at the instant the man catches the ball.

Having tied the two animations together, we have yet to give any absolute time information about starting time or duration of the sequence. The needs of the overall animation may provide some constraints in this regard. Suppose for example, that the ball's high point is to be synchronized with some other event, like a bird's entering the scene, and that the man's tripping over the rock is a cue to the rock to walk away. Then we may tie the ball's highpoint to some time t0 and the man's tripping to some time t1 in the absolute time framework, intending to tie the other events to these times later.

Assuming that only linear scaling is used in resolving the timing, there is now enough information to unambiguously determine S and F, the start and finish times, and any other necessary timing information.

THE NEED FOR SOME PRE-COMPUTATION

The advantages of using no precomputation on the data are simplicity of implementation and economy of core memory. One is presumably willing to pay the price in time of recomputing unchanging values. But the gains are not forthcoming, so the cost is not worth paying.

The major problem with attempting to interpret complex time mappings without any precomputation is the amount of flexibility and power which must be coded into the interpreter. As it tries to evaluate the timing parameters in one tree, the interpreter finds itself chasing through other trees, trying to dig out various values. In the example of the man and the ball, the interpretation of the man's run requires an evaluation of timing information included in the description of the ball's path. More generally, if we assume that the interpreter will always be writing for itself a set of simultaneous equations or inequalities, then the interpreter must know what values to look for in a given data form, how to reduce the form to an equation in the variables that are already in use, how to recognize the relevance or irrelevance of a given datum to the computation at hand, how to resolve variable linkages and finally, how to go about solving sets of linear constraints.

It must exercise all this knowledge a dozen or two times each second, and be able to do so no matter what order the relevant information is presented to it.

A program might be written which could perform all the desired tasks, and solve the resulting sets of equations. Indeed, such a program will have to be written, since one way or another, all the constraints will have to be evaluated. Solving constraint problems is not a new or experimental area in computer science. There is no doubt that the required programs can be made to work. The question is whether they can be made to work in the environment of real-time animation. In such an environment, tables and scratch areas will have to be allocated before the interpreter has a chance to evaluate the needs of the particular problem. The routines in standard software packages for the various mathematical operations involved were never intended to run in real time, and will have to be rewritten accordingly. Attempting such a project would commit a programmer to a probable failure involving months or years of the most brutal type of coding.

Moreover, since the interpreter, using whatever algorithms, may run into any type of situation at any time in the interpretation of a tree, the entire interpreter must be core resident. There simply isn't time to swap in an arbitrary number of subroutines on demand in a frame by frame refresh system.

One concludes that some type of precomputation is warranted.

RE-ORDERING OF DATA TREES

Perhaps the simplest class of precomputations which we may examine is the re-arrangement of the tree, or equivalently, the computation of an optimal touring order. In some cases, a value is developed in the interpretation of one branch of a tree and used in another. As long as the value can be developed without reference to some other branch, it suffices to assure that the branch containing the value is interpreted before any references to it are encountered. In the example of the previous section, the position of the rock along the runner's path may be determined by some external graphic process. If that process has been interpreted before the runner gets to the rock, then the interpreter need not reinterpret the branch which defines the rock's position. There are likely to be many simple cross references like this in a complex animation, and therefore it is worth the trouble to optimize the touring order so as to minimize the number of reinterpretations required. The actual optimization is not all that difficult a task. Assuming that the interpreter has access to off line programs with enough capabilities to track down and extract the needed values, it is a simple process to keep records as to what branches of the tree were visited in developing a value for a given variable. After one pass through the tree, the interpreter will have a list of all the values needed to compute a given value, and all the references to a given value. This list can then be used as input to an optimization program, which will produce the optimal ordering of nodes, subject to the constraints imposed by the graphic context and the needs of specific computations.

Optimizers are not generally known for their speed of execution, so this technique could not be applied on an every frame basis. But that is not necessary, since most changes to a graphic data base will have no effect on the optimal order. Indeed, only those changes which actually restructure the tree, rather than just changing a parameter in some node, would require the optimizer to be called. The only time that this happens is when the animator issues a command which explicitly changes something. The optimizer can be called as part of the command response routine, when the animator is expecting a delay or interruption in the animation.

PRE-INTERPRETATION OF SELECTED VALUES

Whenever some process derives one of its parameters from a knob, the question arises whether the knob should be read once per frame, and the result stored in the process requiring it, or should be read on demand by the interpreter when it is required. As long as each knob occurs only once in a given frame, there is no practical difference in the two techniques. But since it is possible that some node will appear in several places in a tree, it is possible that the knob will be read several times per frame for the same value.

The alternative to reading knobs on demand is to commission a task which will execute once per cycle, before the interpreter is called on any data forms, which provides the needed values to all the data forms which require it. The commissioning and decommissioning of tasks is not difficult to control. Whenever a knob is called for, it is called for by some particular process, such as an editing command. The command's initialization routine includes commissioning the knob tasks, and the exit routine consists of decommissioning these tasks. The data tree using the knob, and the interpreter proper need not be concerned with where the knob task is, or who owns it, etc. That is, the existence of knobs and their differences from static parameters is completely transparent to the interpreter. The fact that a knob is used in editing need not be a part of the data tree.

In the above discussion, a knob has been used as an example of a data extractor whose value can be determined independently of the environment in which it is evaluated. The same comments apply with much more force in the case of complex software extractors. For example, GRAX's p-curve extractor is far more than a few lines of code, so the time it takes to interpret a given frame would increase as some function of the number of occurrences of a given tree in that frame if the extractor were called on demand, rather than once per display cycle. Smooth animations and synchronization between animations would not be achievable, nor would it be possible to get the maximum performance in terms of image complexity in a given frame. This difficulty is avoided by the same mechanism which was applied to knobs, namely moving the extraction process out of the interpretation of the data forms.

The techniques of the above paragraphs discuss ways of dealing with self-contained data sources, such as data extractors. But the example of the man and the ball shows that these techniques are not adequate.

As mentioned in the discussion of non-preinterpretive schemes, it ought to be possible in a program which is not expected to run at animation speed to set up and compute the equations necessary to complexly interlocked timing problems. Given this, we may separate the setting up of the equations from their evaluation, leaving the former task to run as a once-only process whenever a new data structure is commissioned, and accepting the task of evaluation of the equations as a once-per-cycle task.

Once the equations have been set up, the matrices triangulated, the scratch memory allocated, etc., the actual evaluation of the equations will probably be a small enough task to run at real-time speed. The parameters of the equations will be updated only once per frame, as was discussed in the previous section, and the results of the evaluation of the constraints will be made available before the interpreter is called in any given frame.

Prior to actually implementing this scheme, it is not possible to make any good estimates of the complexity of mapping which can be resolved in real time. But there is reason to at least hope that very good results could be achieved, since the actual equations resulting from most graphic situations would probably not be very involved.

The techniques discussed in the next paragraphs are specific to the Evans and Sutherland Picture System I, in that they refer to some details of that machine's hardware and programming strategy. Most of these comments would not, for example, apply to the DEC GT-40 graphics system, because it uses the same core memory for its graphics processor and its CPU. On the other hand, many machines, such as the Vector

General and the Adage have some similarities to the Evans and Sutherland in the way data is handled, and these comments with appropriate modification will apply to them.

The basic storage mechanism in GRAX for static graphic data is lists of X and Y endpoints. The Picture System I, on the other hand, uses homogeneous coordinates, thus requiring an X,Y,Z,W vector for each point. Within GRAX, the partitioning of endpoints into discrete pen strokes (i.e. the handling of "pen up" and "pen down" commands) is stored as an explicit partition at the head the list of endpoint data. The brightness of a given picture is stored in a global parameter. Within the Picture System, brightness is identified with the Z coordinate in the homogeneous vector representing a point, and the pen-up, pen-down functions are handled by command words interspersed within the data.

Conversion between the GRAX format and the Picture System format probably accounts for 75-90% of the total CPU cycles in GRAX, so it is worth the effort to try to cut the number of conversions necessary. To achieve this, when the conversions are first performed, the resulting data could be saved in a file in core, rather than being sent directly to the display hardware. That is, the picture is precompiled into a data structure which needs no computation before being shipped via the DMA interface to the display processor. All references to the original sketch are converted to references to the precompiled display file, and the interpreter deals only with the latter, never having to do any of the conversions at frame rates.

Pre-compilation of sketches has two drawbacks, neither of which outweigh the advantages in performance which accrue from implementing the approach. The first drawback is that the compiled data form is twice as large as the uncompiled one. Using the Picture System in double-buffer mode to ensure smooth animations, it has the capacity for handling about 4000 endpoints per frame. In compiled form, each endpoint takes four words of the computer's memory, and there are a few words required for each pen-up, pen-donw operation. Thus, 16K of memory is required to insure an adequate core buffer. Since it may be desirable to have the original data tree available for editing, this 16K is additional to the 8k buffer needed to contain uncompiled data forms. So, in total, a 24K buffer is required to insure adequate space in almost all conditions. The PDP 11 has a logical address space of 32K. The operating system and the GRAX programs need some space, and it is not desirable or even possible to swap everything often enough to make the needed space available.

On the other hand, when a picture is very complex it is usually made up of several instances of one picture. Only one compiled version of the master image need be stored. That is, a picture made up of 40 one-hundred point circles takes about the same buffer as a picture with one or two one-hundred point circles. So, in many instances buffers much smaller than 24K will suffice.

By allowing the interpreter to display either compiled or uncompiled pictures, the choice of whether or not to compile a given picture may be made when the picture is first defined, with due regard to the amount of buffer space currently available. A simple scheme can be effected which will optimize for speed subject to the availability of space, and resort to the slower method of interpretation as necessary to allow maximum complexity.

In the same spirit, the various matrices which define graphic context information may be precomputed and stored in core, rather than re-computed each frame. The space penalty here is not as great as it is with the basic picture files, since an average picture will contain only a dozen or two matrices, at sixteen words each. The time savings on a percentage basis is greater in this case, since the conversion of a few values to the proper representing matrix is often not computationally trivial. But since there are relatively few matrices in a given display, the absolute savings will not be great.

The second drawback to precompilation of display files for basic graphic entities is that it becomes difficult to maintain good communications between the graphics processor and the CPU. In particular, it is difficult to figure out what the user is pointing at with the stylus. Using the interpreter with its slow, point-by-point method of displaying data, each point can be checked as it is drawn to see if it is within some defined distance of the tablet. (There is a mere 100+ % time overhead for this service, since each point is being drawn twice--once for display, and once for hit-testing.)

Using the fast-as-can-be DMA strategy, no hit testing is performed. There are two modifications possible to the fast-as-can-be approach which would allow it to support hit testing. The first approach would DMA out the display file twice, once for display and once for hit testing. This approach would return only the information that a hit had been detected somewhere, with no clue as to which pen stroke or individual line segment had been hit. The second modification would DMA out a penstroke (or if needed, an individual line segment) at a time, twice. With this approach, full sensitivity to hits could be achieved, but the advantage of DMA would be significantly reduced, since it takes very little longer over a DMA interface to send out 256 words than it takes to send out two words. But stroke-by-stroke or point-by-point hit testing would require that the time sacrifice be made. On the other hand, the advantage of not having to convert the GRAX representation to the Picture System representation would be retained.

The complete implementation of this strategy would require four modes of operation. The first, straight non-precompiled interpretation of a sketch, would be used when the sketch was first being defined, since it would be more time-consuming to recompile the sketch each frame than to re-interpret it. Once the sketch is completed, the full frame, no hit test mode would be entered. This would be the usual mode of operation, and is the fastest of the four modes in execution. When the system is expecting the animator to select a picture, the full-frame hit test mode is entered. Although the full advtanges of DMA transfer and precompilation are achieved, each sketch is shipped out twice, so there is some loss of time. In editing operations, the fourth mode, which is single penstroke at a time with hit testing, would be used. This is the slowest of the precompiled modes, but gives the selectivity in hit testing necessary to the editing of a sketch.

The mode-switching would be handled by the command response routines. Whenever full-frame hit testing is required, it is required for every sketch in the display, so those commands requiring this service would simply set a flag in the interpreter. Stroke-by-stroke hit testing is only required in specific editing commands like ERASE, so these command would be able to apply the appropriate indicator to the item being edited, and switch modes when the edit is complete.

COMPLETE COMPILATION OF DATA TREES

If all the strategies described above were implemented, there would be little work left for the interpreter to do. Basically, the only tasks left to it would be the actual touring of the tree, setting the context for any hit testing which was currently in effect, and sending already computed matrices and display files to the picture processor in the right order.

One view of the situation is that the various optimization routines have built a program schema, one instance of which is executed each frame. The overall structure of this program schema is:

1- Extract all static values from knobs, etc.
2- Extract all dynamic values from p-curves, etc.
3- Use values developed in (1) and (2) to evaluate current constraints.
4- Run through data tree using values developed in (1) through (3) to display the data files in the correct order.

The tree could be pretoured, so that instead of using the interpreter's data linkages and node-to-node linkages, there would be jumps and subroutine calls in Step (4). The result would be a completely compiled representation of the data tree.

The main reason that one might want to do this last compile step is to gain the few instructions difference between the interpreter's search for the next node and the compiled program's subroutine jump. The interpreter is efficient enough so that the probable gains in time would be negligable. There are, however, some fairly significant grounds for avoiding the last phase of the compile operation. These comments apply with equal force to other schemes which would compile the data tree out of existance.

First, the difficulties involving hit testing which were mentioned in previous sections are amplified in a totally compiled scheme. In the compiled program, there is no explicit representation of the tree, so if any subroutine needs to know where it is in the tree, that information will have to be compiled in as an argument vector for the subroutine. In the interpretive scheme, the top level call to the interpreter, each call to a GROUP instruction, and each call to a SKETCH instruction causes an entry to be made into a trace tree, so that if a hit is detected in a sketch, the line segment hit, the SKETCH of which the segment was a part, the arc in a GROUP of which the sketch was a part,...back to the top-level name of the item being interpreted is available for use by whatever program turned on the hit testing.

To maintain this level of specificity in hit-testing, the compiler would have to not only list the graphics which are to be sent to the picture processor, but would also have to write a call to the hit-trace routine, and include the current context as an argument to the call. That is, the compiler would have to preface every call to a graphic routine with a representation of that call's position in the data tree, so that if a hit were detected, its position in the tree could be deduced. In the interpreter, this same information is efficiently maintained in real time, with little space penalty.

As the hit-testing modes changed in response to commands entered by the animator, the program representing the data structure would have to be apprised of this fact. Presumably, the program would just check some

flag somewhere in a global area, rather than be re-compiled. So, we find that short of recompiling our program every time anything changes, we still have to refer to a global data vector to find out what the program is to do. That is, the compiled program is still no more than a program schema. The difference between this and the optimized interpretive scheme is primarily that the latter takes the tree as data, while the compiled program wastes space distributing information which can be as well computed on the fly.

Likewise, we may look at how a compiled representation might handle a sketch which is being added to the current animation. The animator is adding a new point every so often. (It cannot be assumed that the user will add exactly one new point each frame.) Therefore some process external to the program representing the data will have to be called to decide when and how to update the file representing the sketch. That is, the program can call as a subroutine exactly the same subroutine already called by the interpreter, which performs just this update. Again assuming that we do not recompile every frame, we find that the compiled program, like the optimized interpretive scheme, is running program schema rather than self-contained programs.

Given that the difference between the two styles is one of degree only, and that the interpretive scheme takes much less nit-picking distribution of data, and that writing programs which write programs is not the easiest type of task, it seems advisable to retain the interepretive style of execution, and the explicit tree representation of data.

CONCLUSION

Complexly interlocking graphic data trees can best be interpreted in an environment which precomputes selected graphic values and partially precomputes the constraint conditions which govern the display of the graphic data.

The requirements of good hit-testing dictate that the original data tree be retained, and used as input to the graphic interpreter. Given these strategies, it seems likely that good performance can be achieved in a real time animation system which is expected to handle complex timing and positioning scenarios. This optimistic result spurs the author to development of such capabilities for GRAX.

REFERENCES

[1] Barenholtz, J., TOWARDS A NEW ANIMATION TECHNOLOGY, M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1978

[2] Baecker, R. M., INTERACTIVE COMPUTER-MEDIATED ANIMATION, Ph.D. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1969

[3] Burtnyk, N. and M. Wein, "Computer Generated Keyframe Animation", J. Society of Motion Picture and Television Engineers, Vol. 80, Num. 3, pp149-153, March 1971