Amortizing 3D Graphics Optimization Across Multiple Frames

Jim Durbin
Naval Research Lab[1]
Code 5580, 455 Overlook Avenue SW
Washington, D.C. 20375
+1-202-767-6025
durbin@ait.nrl.navy.mil

Rich Gossweiler, Randy Pausch
University of Virginia, School of Engineering
Department of Computer Science
Charlottesville, VA 22903
+1-804-982-2289
[rich | pausch]@Virginia.edu

ABSTRACT

This paper describes a mechanism for improving rendering rates *dynamically* during runtime in an interactive three-dimensional graphics application. Well-known techniques such as transforming hierarchical geometry into a flat list and removing redundant graphics primitives are often performed off-line on static databases, or continuously every rendering frame. In addition, these optimizations are usually performed over the whole database. We observe that much of the database remains static for a fixed period of time, while other portions are modified continuously (e.g. the camera position), or are repeatedly modified during some finite interval (e.g. during user interaction). We have implemented a runtime optimization mechanism which is sensitive to repeated, local database changes. This mechanism employs timing strategies which optimize only when the cost of optimization will be amortized over a sufficient number of frames. Using this optimization scheme, we observe a rendering speedup of roughly 2.5 in existing applications. We discuss our initial implementation of this mechanism, the improved timing measurements, the issues and assumptions we made, and future improvements.

KEYWORDS

three-dimensional graphics, interactive graphics, real-time, optimization, rendering, virtual reality

INTRODUCTION

In 1976 Clarke suggested that a hierarchical data structure would have several characteristics which are useful for manipulating and rendering graphics objects [4]. One powerful advantage of a tree structure for the application programmer is that matrices at each node in the tree can represent individual coordinate systems. When the programmer manipulates a graphics object, its children will inherit the changes implicitly [13].

However, for the rendering engine, a hierarchical data structure is not optimal. To keep the highly pipelined architecture of modern graphics hardware from stalling, we would prefer to make few, if any calculations while traversing the graphics database. To that end, a *flat list* of graphics primitives is preferable, because it requires no combination of transformations (e.g. matrix multiplications). Flat lists also offer the opportunity to perform compiler-like *peephole optimizations* (e.g. removing redundancies), and is efficient for pipelining graphics.

Existing Solutions

The use of optimization techniques to improve rendering speeds is well-established, in both research (e.g. the Berkeley and UNC Walkthrough projects [7][1][2]) and commercial systems (e.g. SGI Performer [11]). These two systems exemplify both ends on the spectrum of *when* to optimize. The Berkeley Walkthrough assumes that the database is static, and can perform aggressive off-line optimization to restructure the database for improved runtime performance. Performer, on the other hand, supports very dynamic environments and cannot make assumptions about the static nature of the database. Performer employs global database optimization *for every frame*. Since optimization takes time, Performer users prefer to execute these mechanisms on an auxiliary CPU to minimize impact on the rendering CPU. Both techniques apply optimization globally over the database.

In this paper, we present optimizations that occurs *after* higher-level techniques such as culling objects which are not in view, or using multiple representations of objects stored at various levels of detail [12].

Our Approach

We have observed that in many applications some portions of the database remain unchanged during the entire run, and other portions change repeatedly, but during brief intervals (e.g. when the user is directly manipulating the object), and still other portions change continuously (e.g. the camera location).

Since we have constructed a rendering system which is application independent [8], we, like Performer, cannot make assumptions that a given portion of the database will remain static. We must analyze the database during runtime. Instead of attempting to transform the entire database during each frame, our mechanism records the frequency with which portions of the database change, and uses this infor-

1 This work predominately completed at the University of Virginia

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
UIST 95 Pittsburgh PA USA
© 1995 ACM 0-89791-709-x/95/11..$3.50

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **NOV 1995**	2. REPORT TYPE	3. DATES COVERED **14-11-1995 to 17-11-1995**
4. TITLE AND SUBTITLE **Amortizing 3D Graphics Optimization Across Multiple Frames**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5580,4555 Overlook Avenue SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**7**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

mation to predict how frequently the object will be changed in the near future. The mechanism determines or predicts the cost of performing an optimization and uses a simple utility function to determine the value of optimizing those portions of the database. The algorithm collects data about the average frequency for which each object gets altered If the object does not get altered frequently, then the cost to optimize the object may be worth the savings recouped over later frames. However, if the object is constantly changing, then there is less incentive to perform the optimization, as the benefits will be fleeting. Our approach is a hybrid, as shown in Figure 1:

Graphics System	*Optimizes* *When*	*How Often*
Walkthrough	off-line	once
Performer	run time	each frame
Alice	run time	when it pays

Figure 1: When to optimize

The Tree Structure

We implemented our optimization mechanism as part of the Alice graphics system [10]. Alice supports a variety of applications ranging from immersed virtual environments, to rapid exploration of three-dimensional interfaces, to a teaching tool for graphics classes. The rendering subsystem [8] is based on a hierarchical data structure. All graphical objects in the rendering sub-system are represented as nodes in the tree. Nodes contain transformation matrices and may also contain geometry consisting of polygons, polylines or polypoints.

When rendering from this data structure, there are two basic forms of inefficiency:

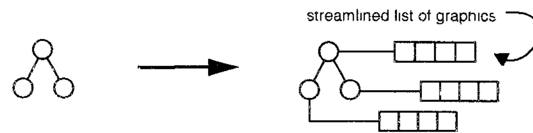
1) matrix multiplications to combine the implicitly chained transformations (in order to render a leaf node, we must first apply all the transformations from the root node to that leaf).

2) redundant calls to set state in the graphics pipeline; for example, in most objects, a large number of polygons are the same color. Repeatedly calling the graphics `setColor()` subroutine (the OpenGL `glColor3f()` call) [9] disrupts the pipeline. In addition, even using a local `if` statement in the inner rendering loop also disrupts the hardware pipeline [5], degrading rendering performance. This is partially because the `if` statement to compare with, for example, the current normal vector, needs to compare three floating point values The most efficient mechanism is to produce the list of graphics calls which will be streamlined into the graphics pipeline, and to remove the redundant calls. This is exactly what Performer does on an extra CPU for each frame.

OUR TECHNIQUE

Our technique involves several optimization phases and a separate mechanism for analyzing the benefit of performing the optimizations. The phases are:

- streamlining each individual node in the tree by creating an array of graphics calls without any intervening computations. This is done for each node in the tree, so that if the node is altered, only that particular node's streamlined array needs to be re-constructed



- peephole optimization to the streamlined list, removing redundancies

- flattening the hierarchical tree structure -- coalescing nodes so that each subtree node has a single cache list representing all of its children at one transformed level. We call this list the *streamlined array.*

- peephole optimization of the flattened list for the entire subtree

Since invoking these optimization mechanisms requires time, we need a guideline to determine if it is worth performing these operations. The algorithm must evaluate the time needed to perform the optimizations, how long the object has remained unaltered and how much of a predicted improvement will be achieved.

Optimization Phases

Converting Each Node into a Rendering List

Each node in the tree may be thought of as a container holding a set of properties describing the object or subobject at that level. For example, in a hierarchical model of a bunny, the bunny's foot may be a child of the bunny's leg. As a node, the foot contains geometry, color information and inherits the transformation matrix from its parent (the leg). When traversing the foot's node, the rendering engine may have to perform several conditional statements to determine how to draw the foot. The color may be inherited from the parent (the leg), set for all of the polygons, be specified for each individual polygon, or specified for each individual vertex of each polygon. Evaluating these conditional state-

ments every frame disrupts the pipeline. If the graphics calls are the same from frame to frame, these calls can be cached into an array. Then the rendering engine can iterate through the array rather than making repeated conditional evaluations. While it is surprising that seemingly minor conditional tests have a strong impact on rendering performance, this is due to the highly pipelined nature of high-end graphics hardware. To quote the authors of SGI Performer:

"The data structures used to represent geometry and the code which transfers that data to the graphics hardware very often make or break an immediate-mode graphics application." [11]

Peephole Optimization

When rendering an object, there are many polygons which are the same color Instead of resetting the color for every vertex or for every polygon, the color call can be factored out and made only once.

This redundancy factoring can occur for many of the properties which characterize the object (e.g. the color, the vertex normal vectors, and the textures). For the initial implementation, our optimization removes redundant color and normal information This operation is performed during the construction of the streamlined array.

Early peephole compilers used a similar method to track the source of a register's value and remove redundant load commands [6] More sophisticated peephole optimizers perform increasingly complex pattern matching and even rearrange code in an attempt to minimize the number of instructions and register manipulations; we hope to use a number of these techniques to further optimize rendering.

Flattening the Tree

Since the optimization mechanism is traversing the hierarchical structure and creating cache arrays for each node, the cost to accumulate the transformation matrices is trivial. This allows the optimization mechanism to effectively flatten subtrees into a single node with a longer cache array.

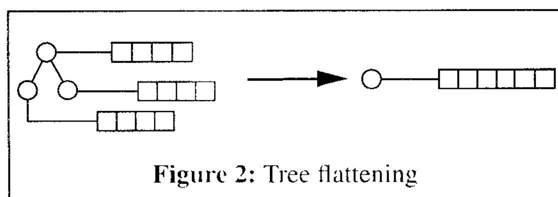


Figure 2: Tree flattening

Note that the original hierarchy is not discarded, since alterations to the nodes in the tree require that the cache be invalidated and the original hierarchy be available.

Flattening the hierarchy into a rendering list is an explicit function call in Performer, and therefore makes it the responsibility of the application programmer to perform this operation. Our optimization mechanism engages automatically when optimization is cost-effective.

Analysis Mechanism

The important distinction of our mechanism is how it determines when optimization should be done, and to which portions of the database. Performer does optimization globally to the entire hierarchy every frame. Our approach is to perform a utility measurement, comparing the cost/benefit of flattening subtrees and factoring out redundant calls. We measure how long an object remains unaltered, how long it takes to optimize, and the value of performing the optimizations.

Wall-clock Time v. Frame Time

Because we are interested in how many times an object is modified versus how many times it is rendered, it is more appropriate to use rendering frame counts rather than wall-clock time. For example, if we used wall-clock time, and the system pauses momentarily (perhaps due to other users on the machine) a single frame could take arbitrarily long to render. The ratio of how frequently the object is modified to how frequently it is rendered is what is important.

While frame-counting is necessary to determine *when* to optimize, wall-clock time is necessary to compute the *effectiveness* of the optimizations. We measure the rendering times of objects in both their unoptimized and optimized states.

Important Information for the Algorithm

AUT -- Average Unaltered Time: the ratio of the number of frames where an object is not altered to the total number of frames (This is a running total average, but other options are discussed later).

TTRU -- Time to Render Unoptimized: the amount of time to render an object without any optimizations.

TTRO -- Time To Render Optimized: the amount of time to render an object once it has been optimized.

OT -- Optimization Time: the time it takes to perform the optimization on the object.

Computing Variables

Since there already is a penalty for loading an object from the disk we expend slightly more time to obtain timing information. We load the object and render it several times without performing a `swapbuffer()` call. During this time we gather the TTRU, the TTRO and the OT. We also create the streamlined array for each node. The performance cost of gathering this information is not observable, as it is dominated by the time to read the object from the disk.

Dynamically Created Objects

During the execution of the program, subtrees may be reparented, effectively creating new objects, and the timing variables cease to be representative. In this case, we use a simple, coarse predictive function based on the number of polygons in a subtree

Based on timings taken on a large number of subtrees on an SGI Reality Engine[2] [3], we find for an N polygon subtree, the representative times are:

EstimatedTTRU = (0.0000127 * N) seconds

EstimatedTTRO = (0.0000038 * N) seconds

EstimatedOT = (0.0000879 * N) seconds

The Utility Function

For each node, we store the TTRU, TTRO, and OT. The utility function computes the amount of savings we will obtain over the average unaltered time (AUT) and adds the cost of performing the optimization. The total is compared to the cost of rendering unoptimized over the same AUT:

if (TTRO*AUT + OT < TTRU*AUT) then optimize...

This simple algorithm does not prevent arbitrarily bad hitches. For example, if the object has not changed for a very long time, but the optimization time (OT) is five seconds, the algorithm would perform the optimization, and the system would stall for five seconds. This problem can be solved by placing a hard upper limit on the allowable OT.

Invalidating the Optimizations

Since our system supports a dynamic tree, we must invalidate the caches when an object's characteristics change (e.g. color, translucency, texture), when the object's matrix undergoes a transformation, and when reparenting occurs The caches are marked dirty and the rendering engine traverses the unoptimized hierarchy instead. Since each node maintains its own streamlined array, any unaltered node keeps its streamlined array:

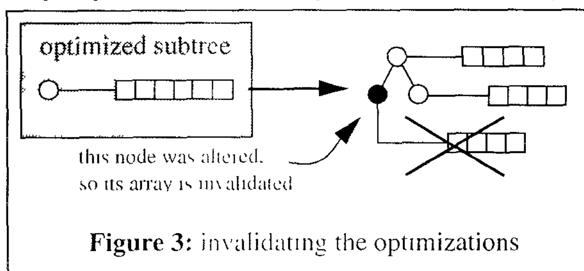


Figure 3: invalidating the optimizations

MEASUREMENTS

The effectiveness of this optimization mechanism depends on how frequently an application alters an object. If the object is altered continuously, then no optimizations are performed. If an object is never altered, then it is optimized once and remains in the optimized state for the duration of the application.

To establish that these optimizations produce a worthwhile savings, we measured the time to render a variety of objects when they are optimized and when they are unoptimized. The results are shown in Table 1. The graphical objects in the table represent several contrived and actual objects to explore variations on tree configurations and redundancy removal. Note that these speedups are greater than 2.5; actual day-to-day use achieves roughly a 2.5 speedup because of constant cost operations per frame such as clearing and swapping the frame buffer, which dilutes the speedup somewhat.

Note to the reviewers: the enclosed video tape shows the amusement park simulation rendering at 12 frames per second without optimizations and 29 frames per second with the optimizations engaged.

As future work we would like to gather statistics about how frequently objects are optimized and how long they remain optimized.

CONCLUSION

The graphics database is accessed by both the application program and the rendering engine, but the usage patterns and frequency of access dictate very different internal representations. Transforming a subtree into a render-optimized list steals rendering time which may be recouped over future frames. The question of utility becomes, "Is it worth the time to optimize now, to speed up the application for the future?" By observing that some parts of the tree are active for discrete periods of time, we have implemented a runtime optimization mechanism which optimizes only local portions when the utility of optimization appears worthwhile. The results are dependent on usage patterns, but initial timing experiments indicate that this mechanism is useful for improving runtime efficiency. Our initial measurements indicate a factor of 2.5 increase in rendering speeds when the optimization mechanism is employed.

HIGH LEVEL OBSERVATIONS

We have two high level observations based on this implementation. The first is that preliminary results of employing this mechanism look promising. Alice was recently used in a graphics class project where ten students built an amusement park (each student built individual rides). This was a fair test, involving an application written by a ten person team, unaware of the underlying optimization techniques. The optimization mechanism improves the rendering by a factor of 2.5.

Our second observation is that trees are typically constructed with very little depth -- on the order of five levels or less.

FUTURE WORK

The following is a brief list of issues we intend to consider as we continue this work.

Spatial Coherency

The rendering engine, by computing the bounding volumes of subtrees, knows which objects in the tree are located near each other after the matrix transformations. This may not be identical to the way the user has constructed the parent-child relationships of the tree.

For portions of the database which do not change over long periods of time, the rendering engine could flatten the application tree and then reconstruct a different tree based on spatial locality. This would allow for more efficient high-level culling

Timing Across Runs

We currently compute all variables for the optimizations at runtime. There may be some benefit in storing the timing data across runs This information might be useful when determining the utility of flattening and may serve as a second-order statistic about the nature of the object (e.g. the "lamp" object is used by many applications as static decoration)

Averaging Differently

We currently compute the Average Unaltered Time (AUT) over the entire run. An alternate approach is to weight the timings, so that over time the more recent alterations influence the average more than alterations having occurred a long time ago. The general problem is similar to the page replacement problem in operating systems.

Application Hints and/or Explicit Control

It might be prudent in some cases to allow the application to explicitly express good moments to perform optimizations. For example, if the application knows that the user has paused or gone into another mode, then that might be a good time to optimize.

Progressive Flattening

For example, if we flatten a bunny object which has children nodes of head, body and arms, and then we rotate the head, the entire head tree will be unflattened. If, instead, we progressively flatten each subtree into larger and larger lists, we can unflatten only the portions which have been altered, rather than the whole subtree.

Combine with Other Optimization Techniques

Using this optimization mechanism does not preclude off-line optimization For example, with a static database such as the Berkeley Walkthrough, off-line optimization can provide a great deal of effective visibility culling. During runtime a given "cell of visibility" may still have a large number of objects. Our mechanism may then be effective for improving the cell.

ACKNOWLEDGEMENTS

We would like to thank all of the members of the User Interface Group at the University of Virginia for their valuable help and ideas during the completion of this work.

REFERENCES

1. John M. Airey, John H. Rholf and Frederick Brooks Jr., *Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments*, **ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics**, 24 (2), 1990, pp. 41-50.

2. Frederick Brooks Jr, *Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings*, **Proceedings of the 1986 Workshop on Interactive 3D Graphics**.

3. Kurt Akeley, *Reality Engine Graphics*, **ACM Annual Proceedings of SIGGRAPH '93 (Anaheim California),** August 1-6, 1993, Addison-Wesley, pp. 109-116.

4. James H. Clark, *Hierarchical Geometric Models for Visible Surface Algorithms*, **Communications of the ACM**, 19(10), October 1976, pp. 547-554.

5. Sharon Clay, member of SGI Perfomer Group. Personal conversation, April 25, 1995.

6. Jack W. Davidson and Christopher W. Fraser, *Register Allocation and Exhaustive Peephole Optimization*, **Software -- Practice and Experience**, 14(9), John Wiley & Sons, September 1984, pp. 857-865.

7. Thomas A. Funkhouser, Carlo Séquin and Seth J. Teller, *Management of Large Amounts of Data in Interactive Building Walkthroughs*, **1992 Symposium on Interactive 3D Graphics, ACM SIGGRAPH**, Cambridge Mass. April 1992, pp. 11-20.

8. Rich Gossweiler, Chris Long, Shuichi Koga, Randy Pausch, *DIVER: A Distributed Virtual Environment Research Platform*, **IEEE Symposium on Research Frontiers in Virtual Reality**, October 25-26, 1993, San Jose, CA, pp. 10-15.

9. Jackie Neider, Tom Davis and Mason Woo, *Open GL Programming Guide*, Addison-Wesley, Reading, Mass, 1993.

10. Randy Pausch, Tommy Burnette, A.C. Capehart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, Jeff White, *Alice: A Rapid Prototyping System for 3D Graphics*, **IEEE Computer Graphics and Applications**, 15(3), May 1995, pp. 8-11. http://www.cs.virginia.edu/~alice/

11. John Rohlf and James Helman, *IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics*, **ACM Annual Proceedings of SIGGRAPH '94**, (Orlando Florida), July 24-29, 1994, Addison-Wesley, pp. 381-393

12. Bruce Schachter (Ed.), *Computer Image Generation*, John Wiley and Sons, New York, NY, 1983.

13. Andries van Dam, et al *PHIGS+ Functional Description 3.0*. **Computer Graphics**, 22 (3), July, 1988, pp 124-218.

Table 1: Optimization Timing Measurements

	pre-optimized tree characteristics			pre-optimized			optimized				
object	Max Tree Depth	nodes	poly-gons	calls to Set Color	calls to Set Normal	time	calls to Set Color	calls to Set Normal	time	time to optimize	ratio
bunny	3	13	389	389	1683	0.00500	36	389	0.00144	0.02964	3.47
cow	1	8	3263	3263	12330	0.04557	1	12255	0.01626	0 41885	2.80
old demo room	2	7	609	609	2462	0.00744	82	609	0.00213	0.04448	3.50
room	2	12	196	196	792	0 00240	38	254	0.00080	0 01691	3.00
new demo room	1	9	55	55	220	0.00067	7	55	0.00016	0 00417	4.19
10 levels deep	10	10	20	20	80	0.00027	1	20	0.00008	0.00187	3.38
20 levels deep	20	20	40	40	160	0.00064	1	40	0.00013	0 00371	4.92
30 levels deep	30	30	60	60	240	0.00093	1	60	0.00019	0 00536	4.89
10 levels wide	1	10	20	20	80	0.00027	1	20	0.00008	0.00183	3.38
20 levels wide	1	20	40	40	160	0.00054	1	40	0.00012	0.00361	4.50
30 levels wide	1	30	60	60	240	0 00085	1	60	0.00019	0.00545	4 47
8 full binary	8	256	512	512	2048	0.00978	1	512	0.00198	0 04649	4 94
10 full binary	10	1024	2048	2048	8192	0 03285	1	2048	0.00834	0.20341	3.94
12 full binary	12	4096	8192	8192	32768	0.16171	1	8192	0.04178	0 84584	3.87
200 node color	1	200	200	200	800	0.00370	200	1	0.00096	0.02164	3.85
300 node color	1	300	300	300	1200	0.00563	300	1	0.00133	0.03298	4.23
200 node normal	1	200	200	200	800	0.00385	1	200	0.00082	0.02219	4.70
300 node normal	1	300	300	300	1200	0.00596	1	300	0.00120	0.03381	4.97
static amusement park	32	385	3490	3490	14596	0.06366	1842	2269	0.01696	0.00728	3.75
animated amusement park	32	385	3490	3490	14596	0.06366	vari-able	vari-able	0.02232	0 40113	2.85

"opt. time" is the Optimization Time, OT

"ratio" is the Time To Render Unoptimized (TTRU) / Time To Render Optimized(TTRO)

"# levels deep" is a 1 node wide, # levels deep tree

"# levels wide" is a 1 node deep, # levels wide tree

"# full binary" is a # level deep, full binary tree

"# node color" is a 1 level deep, # level wide tree, with every polygon set to a different color

"# node normal" is a 1 level deep, # level wide tree, with every polygon rotated to produce a different normal